Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
Journal of Computer and System Sciences 69 (2004) 330–353
JOURNAL OF COMPUTER AND SYSTEM SCIENCES
http://www.elsevier.com/locate/jcss

Integer priority queues with decrease key in constant time and the single source shortest paths problem

Mikkel Thorup

AT&T Labs Research, Shannon Laboratory, 180 Park Avenue, Florham Park, NJ 07932, USA

Received 22 July 2003; revised 8 March 2004

Available online 20 July 2004

Abstract

We consider Fibonacci heap style integer priority queues supporting find-min, insert, and decrease key operations in constant time. We present a deterministic linear space solution that with n integer keys supports delete in $O(\log\log n)$ time. If the integers are in the range $[0, N)$, we can also support delete in $O(\log\log N)$ time.

Even for the special case of monotone priority queues, where the minimum has to be non-decreasing, the best previous bounds on delete were $O((\log n)^{1/(3-\varepsilon)})$ and $O((\log N)^{1/(4-\varepsilon)})$. These previous bounds used both randomization and amortization. Our new bounds are deterministic, worst-case, with no restriction to monotonicity, and exponentially faster.

As a classical application, for a directed graph with n nodes and m edges with non-negative integer weights, we get single source shortest paths in $O(m + n\log\log n)$ time, or $O(m + n\log\log C)$ if C is the maximal edge weight. The latter solves an open problem of Ahuja, Mehlhorn, Orlin, and Tarjan from 1990.

Keywords: Integer priority queues; Decrease key; Fibonacci heaps; Single source shortest paths

1. Introduction

In 1984, Fredman and Tarjan introduced Fibonacci heaps [15], which is a priority queue over a dynamic ordered set H supporting the following operations:

find-min(H) Returns an element from H with minimum key value in constant time.

insert(H, a) Adds the element a to H in constant time.

dec-key(H, a, x) Reduces the key value of element a to x in constant time. If the current key value of a was smaller than x, it is an error.

E-mail address: mthorup@research.att.com.

0022-0000/$ - see front matter
doi:10.1016/j.jcss.2004.04.003

delete(H, a) Deletes element a from H in $\tau = O(\log n)$ time where n is the current size of H.

It is important to note that we distinguish between an element and its key value. The Fibonacci heap stores information with the elements that it can benefit from in connection with a dec-key operation.

The classic application of Fibonacci heaps is inside Dijkstra algorithm [12]. For a weighted graph with n nodes and m edges they solve the single source shortest path problem in time $O(m + n\tau) = O(m + n \log n)$.

For general ordered sets supporting only comparisons, Fredman and Tarjan's delete time of $O(\log n)$ is optimal. However, here we show

Theorem 1. *We can implement a priority queue in linear space that with n integer keys in the range* $[0, N)$ *supports* find-min, insert, *and* dec-key *in constant time, and* delete *in* $O(\log \log \min\{n, N\})$ *time.*

Even for the special case of monotone priority queues, where the minimum has to be non-decreasing, as in Dijkstra's single source shortest path algorithm, the best previous bounds on delete were $O((\log n)^{1/(3-\varepsilon)})$ and $O((\log N)^{1/(4-\varepsilon)})$ due to Raman [25]. These previous bounds used both randomization and amortization. Our new bounds a deterministic, worst-case, with no restriction to monotonicity, and most importantly, exponentially faster.

The bounds in Theorem 1 match the current best bounds of Thorup [28] for a basic priority queue, that is, a priority queue as above but without a constant-time dec-key operation. In fact, any improvement of the $O(\log \log n)$ bound would lead to a better sorting algorithm. More precisely, we can sort n keys by first inserting them all in a priority queue and second extract them in sorted order. Using the priority queue of Theorem 1, this takes $O(n \log \log n)$ time, matching the best current sorting bound due to Han [19]. With randomization, however, Han and Thorup have shown how to sort in $O(n\sqrt{\log \log n})$ expected time, and with a general reduction from basic priority queues to sorting, Thorup [28] has shown that this implies a basic priority queue with delete in $O(\sqrt{\log \log n})$ expected time. Combining the latter with dec-key in constant time is left as a major open problem.

From Theorem 1, we immediately get

Corollary 2. *We can solve the single source shortest path problem for a directed graph with n nodes and m edges with weights in the range* $[0, C)$ *in linear space and* $O(m + n \log \log \min\{n, C\})$ *time.*

Proof. We use the priority queue from Theorem 1 in Dijkstra's algorithm. A small point to observe, however, is that the priority queue in Dijkstra's algorithm only needs to deal with integers in the range $[0, C)$ even though the distances may be up to nC. We just have to bucket distances δ according to $\lfloor \delta / C \rfloor$. Then, for $i = 0, \ldots, n$, we deal with distances in bucket i in the range $[iC, (i+1)C)$. The overhead from the bucketing is constant time per operation. This idea essentially goes back to Dial in 1969 [11]. □

The $O(m + n \log \log C)$ time bound from Corollary 2 solves an open problem of Ahuja et al. [1] from 1990.

1.1. The integer key model

Our computational model is word RAM, modeling what we program in a standard programming language such as C [22]. We have a processor determined word-size W, limiting how big integers we can operate on in constant time. On the other hand, we assume that each input integer fits in a single word. Also, we assume that $W \geqslant \log n$ so that we can address the different keys with single words. The machine instructions are those available via C. Besides direct and indirect addressing and conditional jumps, we have functions, such as addition and multiplication, operating on a constant number of words. The memory is divided into words, addressed $0, 1, 2, \ldots$ The time is the number of instructions performed and the space is the maximal address used.

The essential advantage of dealing with integer keys is that we can use them to compute addresses in bucket-based algorithms. A classic example of this is the folklore algorithm radix sort, which according to Knuth [23] is referenced as far back as in 1929 by Comrie in a document describing punched-card equipment [8]. Another classic example is multiplicative hashing which dates back at least to 1956 [14]. Finally, for the SSSP problem itself, the use of buckets go back at least to Dial's algorithm from 1969 [11].

We note that our integer priority queue also works for floating point numbers; for the IEEE 754 floating-point standard [21] is designed so that the ordering of floating point numbers can be deduced by perceiving their representations as integers. Thus the $O(m + n \log\log n)$ time bound from Corollary 2 holds equally well for floating point numbers. However, for floating point numbers, we do not get bounds in terms of the maximal weight. Similarly, we note that our priority queues can be applied to signed negative and positive integers. More precisely, if we flip the signbit of signed integers and view them as unsigned, we preserve their ordering. Applying this transformation to signed integers, we can use our priority queue for non-negative integers. However, Dijkstra's algorithm only works for non-negative integers, so this has no impact on Corollary 2.

Our algorithms use multiplication, which is not an AC^0 operation [6]. However, if we restrict ourselves to AC^0 operations such as addition, shift, and bit-wise boolean operations, we can solve the SSSP problem in $O(m + n(\log\log \min\{n, C\})^{1+\varepsilon})$ time using a monotone AC^0 variant of the priority queue from Theorem 1.

1.2. History

The history behind our result is best cast in terms of its implications for the single source shortest path problem (SSSP).

1.2.1. Bounds in terms of C

Back in 1969, Dial [11] developed an $O(m + nC)$ time SSSP algorithm for integer weights in the range $[0, C)$. For contrast, if we use the data structure of van Emde Boas, Kaas, and Zilstra [30,31] from 1977, supporting both insert and delete in $O(\log\log N)$ time, we get a solution in $O(m \log\log C)$ time. As commented in [1]:

> *Based on the existence of the Van Emde Boas-Kaas-Zilstra data structure, one might hope for a bound of $O(m + n \log\log C)$. Obtaining such a bound is an open problem.*
>
> Ahuja, Mehlhorn, Orlin, and Tarjan, 1990.

As stated in Corollary 2, we obtain in this paper exactly the bound hoped by Ahuja et al. [1].

Ahuja et al. [1] proceeded to provide a SSSP algorithm with running time $O(m+n\sqrt{\log C})$. In 1997, using randomization, this bound was improved by Cherkassky, Goldberg, and Silverstein [7] to $O(m+n\sqrt[3]{\log C}^{1+\varepsilon})$ expected time for any fixed $\varepsilon>0$. Later in 1997, this was further improved by Raman [25] to $O(m+n\sqrt[4]{\log C}^{1+\varepsilon})$ expected time. We note that the per node cost in all the above bounds is $(\log C)^{\Omega(1)}$, hence that our per node cost of $O(\log\log C)$ from Corollary 2 is an exponential improvement.

Of other related results, Thorup [26] has shown that SSSP with non-negative integer weights can be solved in linear time if the graph is undirected. Hagerup [18] generalized Thorup's algorithm for directed graphs, solving SSSP in $O(m\log\log C)$ time and linear space. For contrast, the previously mentioned $O(m\log\log C)$ time bound with van Emde Boas' data structure needs either randomization or $O(m+C^{\varepsilon})$ space. Here ε may be any positive constant. Yet C^{ε} is not bounded in terms of m. As mentioned, our new $O(m+n\log\log C)$ time bound is achieved deterministically in linear space, that is, space linear in m.

1.2.2. Bounds in terms of n

Shortly after the development of their $O(n\log n/\log\log n)$ time integer sorting algorithm in 1990, Fredman and Willard [16] showed that integer SSSP could be solved in $O(m+n\log n/\log\log n)$ time. In 1995, using randomization, Thorup [27] improved this to $O(m+n\sqrt{\log n}^{1+\varepsilon})$ expected time. He also solved the SSSP problem in $O(m\log\log n)$ expected time, thus getting bounds in terms of n matching those we had in terms of C after the above mentioned results of Ahuja et al. [1] in 1990.

In 1996, Raman solved the SSSP deterministically in $O(m+n\sqrt{\log n\log\log n})$ time and linear space [24]. Later, in 1997, using randomization, he solved SSSP in $O(m+n\sqrt[3]{\log n}^{1+\varepsilon})$ expected time [25]. However, as stated in Corollary 2, we solve the SSSP problem deterministically in linear space and $O(m+n\log\log n)$ time. Thus, we improve the per node cost with respect to n exponentially, as we did it with respect to C in Section 1.2.1.

We note that the recent randomized $O(n\sqrt{\log\log n})$ sorting algorithm of Han and Thorup [20] together with Thorup's reduction from SSSP to sorting [27] imply that we can now solve SSSP in $O(m\sqrt{\log\log n})$ expected time. This leaves as a natural challenge, the problem of solving SSSP in $O(m+n\sqrt{\log\log n})$ expected time. However, from the stand point of deterministic algorithms, we note that since Dijkstra's SSSP algorithm [12] sorts the nodes in order of increasing distance from the source, our deterministic $O(m+n\log\log n)$ time Dijkstra implementation cannot be improved without improving the current best $O(n\log\log n)$ time bound of Han [19] for deterministic sorting.

1.3. Techniques

Our new data structure for priority queues with dec-key in constant time combines Andersson's exponential search trees [3] with the recent priority queues of Thorup [28]. The fundamental new idea is to shift keys between levels in a search tree so as to support dec-key in constant time.

1.4. Contents

First we present a simpler amortized version of the priority queue from Theorem 1. This conveys most of the new ideas, and the amortization is not a problem for the application inside Dijkstra's single source shortest path algorithm. In particular, our amortized version of Theorem 1 implies Corollary 2. Next we present a worst-case priority queue as claimed in Theorem 1. Finally, we present a monotone implementation using only AC^0 operations.

2. Preliminaries

In this section, we describe some basic terminology and results to be used in the rest of the paper.

When we say *list*, we think of a doubly linked list, that is, with predecessor and successor pointers between the elements. Also, we have direct access to the first and the last element, called the *head* and the *tail*, respectively. Thus, in constant time, we can insert and delete elements and concatenate and split lists. If the elements are ordered, we call it a *sorted list*. If we do not care at all about the ordering, we call it a *bucket*.

In 1990, Ahuja et al. [1, Section 2] showed that for integers in the range $[0, N)$, Fibonacci heaps can be implemented with deletes in $O(\log N)$ time, improving over the original $O(\log n)$ time if $N = n^{o(1)}$. The lemma below presents a completely general such reduction that works for any priority queue.

Lemma 3. *Let $\tau(n, N)$ be the* delete *time for a priority queue for up to n integers in the range $[0, N)$ supporting* insert *and* dec-key *in constant time. Then $\tau(n, N) \leqslant \tau(N, N)$. This holds whether τ is amortized or worst-case.*

Proof. If $N \geqslant n$, there is nothing to prove, so we assume $N < n$. First we consider the simpler case of a monotone (non-decreasing minimum) priority queue with amortized bounds. If we end up with more than N keys, we make an array, mapping key values into buckets of keys with that value. We can then find the right bucket for a key in constant time, thus implementing the updates themselves in constant time. We also need to maintain the minimum value μ. Since the priority queue is monotone, the minimum can only change when the last key with minimum value μ is deleted, and then we just step up μ till we reach a value with a non-empty bucket of keys. In total, we only step μ up $N < n$ times, so the cost per key is constant.

We will now avoid the assumption of monotonicity. If we end up with N keys in our priority queue, we make a bucket sort in $O(N)$ time, producing a sorted list of key values, each with an associated bucket of keys with that value. The priority queue itself is thereby emptied. From now, we need to operate both on a priority queue, and on the sorted list. We need to check both for the minimum. New keys are inserted in the priority queue. A key is deleted from wherever it resides. If a key in the priority queue is decreased, it is done within the priority queue. If a key in the list is decreased, it is removed from the list and inserted in the priority queue, all in constant time. Now, if the priority queue gets full again with N keys, and we already have a list, we just produce a new list in $O(N)$ time, and then merge the new list with the old list in $O(N)$ time. This includes uniting

buckets for the same key value so that the final list has only one bucket for each key value. Thus it only takes $O(N)$ time to empty the priority queue into the list, which is constant time per key transferred.

Finally, we observe that the above is easy to de-amortize. We just start emptying the priority queue when it gets up to $0.75N$ keys, and complete the emptying over the next $0.25N$ operations. □

Finally, we will need the concept of splitting: by *splitting an ordered set X over k splitters* $y_1 < y_2 < \cdots < y_k$, we mean that we partition X into sets $X_0, X_1, \ldots, X_k$ such that for $i = 1, \ldots, k$, $\max X_{i-1} < y_i \leqslant \min X_i$. We note that if X consists of a singleton element x, then this is the standard search problem of finding the maximal i such that $y_i \leqslant x$. We shall use the following lemma of Han and Thorup [20]:

Lemma 4. *For any constant $\varepsilon > 0$, we can split n integer keys over $n^{1-\varepsilon}$ splitters in linear time and space.*

3. A simpler amortized priority queue

In this section, we describe a simpler amortized version of the priority queue of Theorem 1. It is convenient to have all keys distinct and finite, leaving the value ∞ for special purposes. For a linear space construction like ours, the uniqueness can be achieved by augmenting each key with its address of $\log n + O(1)$ bits. Keys are still contained in a constant number of words, so this will not affect our asymptotic bounds.

We will now turn to our construction, which is illustrated in Fig. 1. We will maintain a list of $O(\log\log n)$ *level splitters* $s_0, s_1, \ldots, s_\ell$, $s_0 = 0$, so that the number of keys in the interval $I_i = [s_i, s_{i+1})$ is in the order of $n_i = A^{(5/4)^i}$ keys. Here $A = \Theta(\log n)$ and $s_{\ell+1} = \infty$, and we allow fewer keys in $[s_\ell, \infty)$. We think here of $A = n_0 = \Theta(\log n)$ as fixed. We shall return to the fixedness of A in the very end of this section.

The keys in interval I_i are said to be on *level i*. These keys are split into $\Theta(n_i^{1/3})$ *local sets* $X_{i0}, \ldots, X_{ik}$, each with $\Theta(n_i^{2/3})$ keys. For comparison, the number of keys in interval I_{i-1} is $\Theta(n_i^{4/5})$. If there is only one local set on the last level ℓ, it is allowed $O(n_\ell^{2/3})$ keys. Between the local sets, we have *local splitters* $s_{i1}, \ldots, s_{ik}$ such that for $j = 1, \ldots, k$, $\max X_{i(j-1)} < s_{ij} \leqslant \min X_{ij}$. It is convenient to define $s_{i0} = s_i$ and $s_{i(k+1)} = s_{i+1}$. Then $X_{ij} \subseteq [s_{ij}, s_{i(j+1)})$ for $j = 0, \ldots, k$. Finally, each level i has an associated buffer B_i of $O(n_i^{1/2})$ update keys, that is keys to be inserted or deleted, with values in I_i.

Above, the indices are used abstractly for clarity of exposition, not to indicate direct physical access using the indices. In our implementation, the level splitters are stored in a sorted list $(s_0, s_1, \ldots, s_q)$, $q = O(\log\log n)$. With each level splitter s_i, we store the number of keys at that level. Also, s_i is head of a sorted list of local splitters, $s_{i0}, \ldots, s_{ik}$. Together with each s_{ij} we store a bucket with the local set X_{ij} as well as the size of X_{ij}. With such a representation, given a pointer to a local set X_{ij}, we can, for example, join X_{ij} with its successor set $X_{i(j+1)}$ in constant time, setting $X_{ij} = X_{ij} \cup X_{i(j+1)}$, and deleting $s_{i(j+1)}$ from the local splitter list.

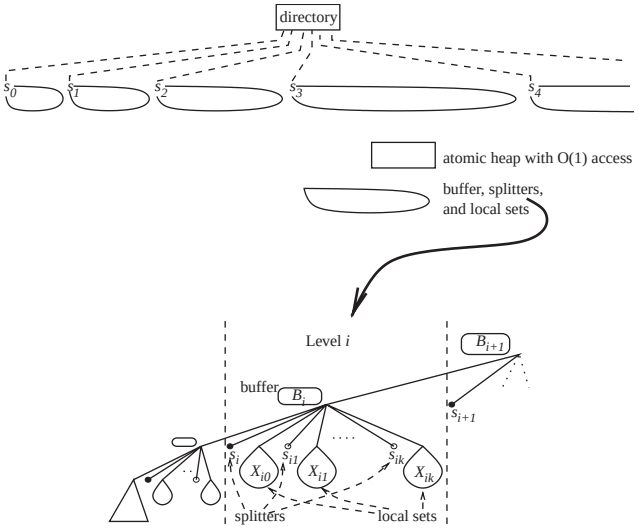

Fig. 1. The priority queue.

As a special case, the first local set X_{00} on level 0 is stored both as a bucket and in a Fibonacci heap [15]. Since this local set is of size $O((\log n)^{2/3})$, it supports deletes in $O(\log\log n)$ amortized time. We call X_{00} the *min-set* and the Fibonacci heap the *min-priority queue*. The local splitter s_{01} between the min-set and the next local set X_{01} is called the *min-splitter*. The level 0 buffer B_0 does not contain any keys below the min-splitter, so the global minimum is always found by querying the min-priority queue.

Finally, we have Fredman and Willard's atomic heap from [17] over the $O(\log\log n)$ level splitters. Then, given a key value x, in constant time, we can find its level i with $x \in I_i = [s_i, s_{i+1})$. Also, in constant amortized time, we can change the value of a level splitter, or add or delete a last level splitter s_{k+1}. We call this atomic heap the *directory*. As will be detailed in Section 5.1, it is possible to avoid the use of atomic heaps in a pure AC^0 construction.

3.1. The basic workings

We will now sketch the workings of our priority queue construction. First, on each level, the buffer size is designed so that it, when full, can be emptied over local set in constant time per update key using Lemma 4. Since the directory identifies the level in constant time, this essentially means that we can place keys in the data structure in constant time per key. The real cost of our data structure has to do with restructuring when the sizes of local sets or levels get out of wack. Redistributing keys between local sets is essentially standard, and is described in Section 3.2.

The interesting new stuff is the redistribution of keys between levels, as illustrated in Fig. 2. If a key is deleted from level i, for levels $j = i, i+1, \ldots$, we want to *shift in* a key from level $j+1$. Clearly this would preserve the sizes of the levels. However, to do the shifts in amortized constant

time, we shift in a whole local level $j+1$ set to level j at the time. With $O(\log\log n)$ levels, we pay $O(\log\log n)$ amortized time for the shifting in connection with a delete. This also includes the delete from the min-priority queue if $i=0$.

The most interesting thing is that we can decrease a key in constant amortized time. When a key is decreased from level k to level i, for $j=i,\ldots,k-1$, we want to *shift out* a key from level j to level $j+1$. This requires $O(\log\log n)$ shifts, but the point is that a shift out takes much less than constant time on the average. Shifting out from level j is done by joining enough of the last local level j sets so that they qualify as a local level $j+1$ set. We then shift that whole set out to level $j+1$. This only takes constant time per local level j set as the union is a simple concatenation of lists. Since local level j sets have size $\Theta(n_j^{2/3})$, the shift out cost per key is $O(1/n_j^{2/3})$, and summing over the levels, the decrease cost is $\sum_{j=i}^{k} O(1/n_j^{2/3}) = O(1)$. We note that an insert can be viewed as a decrease of an infinite key, and it is implemented as the decrease above, but with k the maximal current level. Hence insert takes constant time like decrease.

In the real shifting of keys between levels, we also have to worry about shifting update keys in buffers, but this does not affect the overall bounds. The details of the shifting are described in Section 3.3.

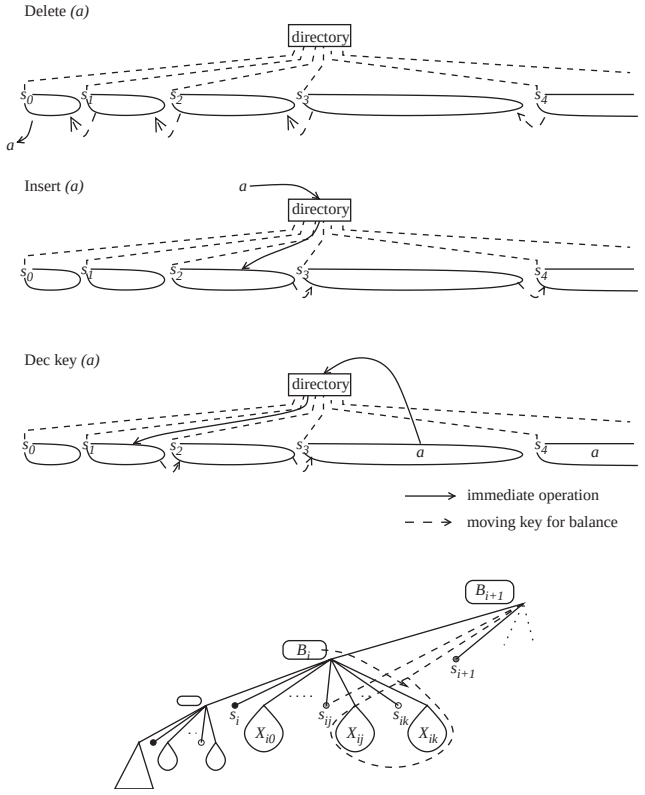

Fig. 2. Moving keys between levels.

3.2. Distributing keys in local sets

We will now describe how to insert and delete keys from level i in constant time per key. This includes keeping the local sets of sizes between $(0.5 - o(1))n_i^{2/3}$ and $(2.0 + o(1))n_i^{2/3}$. In this part, we ignore that the levels themselves may get too many or too few keys.

When keys are inserted or deleted, the directory first finds their level i, and they are then placed as update keys in the buffer B_i. Every time the buffer size reaches $n_i^{1/2}$, using Lemma 4, we split it over the $O(n_i^{1/3})$ local splitters in constant time per update key. For each update key, we now know its local set. We can then insert or delete the key from the local set and update the set counter, both in constant time.

With deletes, it may seem that we could bypass the buffers, and just deleted the key from whatever local set it was in. However, we need to know which local set we delete from in order to update the counter of the local set.

Each time we have emptied the buffer and updated the local sets, we check the counters of all the affected local sets. If a local subset has more than $2n_i^{2/3}$ keys, we split it in two. Since only $n_i^{1/2}$ buffer keys are placed at the time, the local subset being split has $(2 + o(1))n_i^{2/3}$ keys. In $O(n_i^{2/3})$ time, we identify and split the subset over a median, which is made a new local splitter. The resulting local subsets have $(1 + o(1))n_i^{2/3}$ keys. Similarly, if a subset has below $0.5n_i^{2/3}$ keys, we join it with a neighbor, getting between $n_i^{2/3} - o(1)$ and $2.5n_i^{2/3}$ keys, and split that in two if it is of size bigger than $1.5n_i^{2/3}$. We continue this way until each local set has size between $0.5n_i^{2/3}$ and $2n_i^{2/3}$.

To analyze the above, we note that whether a local subset is created from a join or a split, its *start size* is between $0.75n_i^{2/3}$ and $1.5n_i^{2/3}$. This means that it takes at least $0.25n_i^{2/3}$ new updates to the local subset before it initiates a new join or a split as above. Since join and split take $O(n_i^{2/3})$ time, they are done in constant time per update.

As an exception to the above is if an update is to the min-set on level 0, that is, if the value is below the min-splitter s_{01}. In this case, we bypass B_0 and update the min-set directly. If the min-set gets too many or too few keys, we make a split or join with its neighbor X_{02} as described above. In case of a join, we get a new bigger min-splitter $s'_{01} = s_{02}$, and we have to scan the buffer B_0 for keys below s'_{01} and place them in the new min-set X'_{01}. Finally, we construct a new min-priority queue, inserting each key from the new min-set in constant time. Even with the above extra work, splitting or joining the min-set takes $O(n_0^{2/3})$ time, just like any other local split or join on level 0. Hence the cost is constant per update to the min-set.

3.3. Redistributing keys between levels

Over time, we may get too many or too few keys in some of the levels. If level i gets more than n_i keys, we shift out some of the bigger keys to level $i + 1$. As a special case, if level i is the last level, we first have to create a new last level splitter s_{i+1} whose initial value is ∞.

Conversely, if level i is not last and gets less than $n_i/2$ keys, we shift in some keys from level $i + 1$. If level $i + 1$ is the last level, and we empty this level, we remove the level splitter s_{i+1}.

As sketched in Section 3.1, it is much faster shifting out than shifting in, and this asymmetry is the key to our constant time decrease-key operation.

Shift out First we empty the level i buffer B_i, in time $O(n_i^{1/2})$. Next, we collect local sets of I_i starting from the tail, that is, with the larger keys, until we have enough keys for a local level $i+1$ set S. More precisely, we stop collecting local level i sets when we have $\geqslant n_{i+1}^{2/3} = n_i^{5/6}$ keys. The local level i sets each have size $\Theta(n_i^{2/3})$, so S gets $n_i^{5/6} + O(n_i^{2/3}) = (1+o(1))n_{i+1}^{2/3}$ keys. This size is fine with our accounting from Section 3.2 which assumed start sizes anywhere between $0.75n_i^{2/3}$ and $1.5n_i^{2/3}$. The local sets are joined in constant time per set, so they are all found and collected in $O(n_i^{5/6}/n_i^{2/3}) = O(n_i^{1/6})$ time. Finally, the local splitter in front of the collected local sets is made the new splitter s_i whereas old splitter s_i is used as the local splitter after the new local level $i+1$ set S. The total time for shifting out is $O(n_i^{1/2} + n_i^{1/6}) = O(n_i^{1/2})$.

We note here that the min-set cannot be shifted out this way, for we only shift out a small fraction of the last local sets of a level, but the min-set is the first local set on level 0. More precisely, for $i=0$, out of the $\Theta(n_0)$ keys on level 0, the set S shifted out contains $\Theta(n_0^{5/6})$ largest keys whereas the min-set contains $\Theta(n_0^{2/3})$ smallest keys. Since $n_0 = \omega(1)$, it follows that the two sets are well separated. Consequently, when joining sets for S, we do not need to worry about the min-priority queue.

Shift in First we empty the level $i+1$ buffer B_{i+1}, in time $O(n_{i+1}^{1/2}) = O(n_i^{5/8})$. Next we take the first local level i set and insert its contents in level i, via the level i buffer B_i as described in Section 3.2 at constant cost per key. Since there are $O(n_{i+1}^{2/3}) = O(n_i^{5/6})$ keys, the total time for shifting in is $O(n_i^{5/6})$.

3.3.1. *Analysis*

First we consider how keys get distributed between levels in connection with an update. Before the update each level i has between $n_i/2$ and n_i keys, except that the last level ℓ may have fewer keys. The update itself can change these numbers by one, but that may start a cascade of shifts between levels. We will show that our data structure behaves nicely as long as no level performs more than one shift. In particular, we will show that no level can initiate a second shift.

A first important implication of each level having done at most one shift is that only the last level may run out of keys for the shifts. More precisely, consider any level $i<\ell$. With one shift on each level, we may lose $O(n_i^{2/3})$ keys to level $i-1$ and $O(n_i^{5/6})$ keys to level $i+1$. We had at least $n_i/2$ keys on level i before the update, so $n_i/2 - O(n_i^{5/6}) > 0$ keys remain. Concerning level ℓ, it cannot shift in as long as it is the last level, so if it makes a shift, it is a shift out. If level ℓ shifts out, creating level $\ell+1$, it is because it has more than n_ℓ keys, and then it cannot be emptied by a shift in on level $\ell-1$. If level ℓ does not shift out, it may run out of keys due to a shift in on level $\ell-1$, but that just implies that $\ell-1$ becomes the new last level.

We now want to show that no level i can be first to initiate a second shift. Suppose the first shift on level i was a shift out. The reason for the shift out was that level i had more than n_i keys. On the other hand, level i had at most n_i keys before the update. In the shift out, level i loses $\Theta(n_i^{5/6})$

keys to level $i+1$. It may also exchange $O(n_i^{2/3})$ keys with level $i-1$. Nevertheless, as long as no level performs more than one shift, after its shift out, level i will have $n_i - \Theta(n_i^{5/6}) \pm \Theta(n_i^{2/3}) = n_i - \Theta(n_i^{5/6})$ keys. Since this is between $n_i/2$ and n_i, we conclude that level i cannot be the first to initiate a second shift.

Now suppose the first shift on level i was a shift in. Since the last level does not shift in, this implies that i was not the last level, hence that it had at least $n_i/2$ keys before the update. On the other hand, the reason for the shift in was that level i had less than $n_i/2$ keys. Normally level i will shift in $\Theta(n_i^{5/6})$ keys from level $i+1$. Including an exchange of $O(n_i^{2/3})$ keys with level $i-1$, level i will then have $n_i/2 + \Theta(n_i^{5/6}) \pm \Theta(n_i^{2/3}) = n_i/2 + \Theta(n_i^{5/6})$ keys, preventing it from initiating a second shift. We note that level i may shift in less keys if it empties level $i+1$ as the last level. In that case, level i may end up with fewer keys, but then it is the last level, so it cannot shift in again. Since it needs n_i keys to shift out, it cannot be first to initiate a second shift.

Thus, we conclude that no level performs more than one shift, hence that all the analysis above is valid. In particular this implies that the redistribution between levels terminates after an update, leaving level i with between $n_i/2$ and n_i keys for the next update.

Amortization In general, each shift on level i will be amortized over updates done since that of the last activity on level i. Here an activity on level i is either a shift or the creation of level i as a new last level. Logically, it could also be the destruction of level i, but obviously, when we consider a shift on level i, the last activity on level i cannot be its destruction.

Shift out First, we consider a shift out on level i. We want to argue that there has been at least $\Theta(n_i^{5/6})$ keys inserted or decreased below s_{i+1} since the last activity. To this end, we first bound the number of keys below s_{i+1} just after the last activity.

If the last activity was the creation of level i, it left us with $\Theta(n_i^{2/3})$ keys on level i. Also, our previous analysis shows that a shift in would have left us with $n_i/2 + O(n_i^{5/6})$ keys and that a shift out would have left us with $n_i - \Theta(n_i^{5/6})$ keys on level i. Moreover, we had at most $\sum_{j=0}^{i-1} n_j = O(n_i^{4/5})$ keys smaller than s_i. Hence, after the last activity, we had at most $n_i - \Theta(n_i^{5/6})$ keys smaller than s_{i+1}.

To make a new shift out on level i, we need at least n_i keys on level i, hence at least that many keys smaller than s_{i+1}. Thus, there must be at least $n_i - (n_i - \Theta(n_i^{5/6})) = \Theta(n_i^{5/6})$ new keys smaller than s_{i+1}, and each of these is either due to an insert or dec-key operation. Including the time to empty buffer B_i, the shift out takes $O(n_i^{1/2})$ time. Hence the cost at level i associated with each of the new insert or dec-key operations is $O(n_i^{1/2}/n_i^{5/6}) = O(n_i^{-1/3})$. Summing over all levels, the cost of shifting out is $O(\sum_{i=0}^{O(\log\log n)} n_i^{-1/3}) = O(1)$ per insert or dec-key.

Shift in We now consider a shift in on level i. The analysis is very similar to that for shift out. The last activity cannot be a creation of level i as the last level, for then, to shift in, we first need a shift out to create a level $i+1$. Thus the last activity was a shift. If the last shift was a shift out, it leaves us with $n_i - \Theta(n_i^{5/6})$ keys on level i. If it was a shift in, since it did not empty level $i+1$, it leaves us with $0.5n_i + \Theta(n_i^{5/6})$ keys on level i. Thus we have at least $0.5n_i + \Theta(n_i^{5/6})$ keys on level i after the last activity. Also, to make a new shift in, including the keys smaller than s_i, we have at

most $0.5n_i + O(n_i^{4/5})$ keys smaller than s_{i+1}. Consequently, we have lost at least $0.5n_i + \Theta(n_i^{5/6}) - (0.5n_i + O(n_i^{4/5})) = \Theta(n_i^{5/6})$ keys smaller than s_{i+1}, and this loss is all due to deletes. Since the shift in takes $O(n_i^{5/6})$ time, the level i cost associated with each of these delete operations is constant. Summing over all $O(\log\log n)$ levels, the cost of shifting out is $O(\sum_{i=0}^{O(\log\log n)} 1) = O(\log\log n)$ per delete operation. This cost also includes a deletion from the min-priority queue if the key is in the min-set.

In fact, we can tighten the last analysis a bit based on the rank q of a deleted key x. Here the rank is the number of keys that are smaller than q. We note that deleting a key only costs on level i if the key is smaller than s_{i+1}. However, we know that s_{i+1} has rank $\Theta(n_i)$ except if i is the last level. Moreover, $n_{i+1} = n_i^{5/4}$. Hence, the number of levels i with $s_{i+1} > x$ is $O(1 + \log\frac{\log n}{\log q})$, bounding the shift out cost for deleting x. We note that this cost also includes an $O(\log\log n)$ time deletion from the min-priority queue if the key is in the min-set, for if x is in the min-set, we have $q = O(\log n)$, and then $(1 + \log\frac{\log n}{\log q}) = \Theta(\log\log n)$. The interesting point in the $O(1 + \log\frac{\log n}{\log q})$ bound is that we pay less deleting keys of higher rank. In particular, we pay constant amortized time for keys of rank $n^{\Omega(1)}$.

3.3.2. The size of the min-set

Finally, we make some remarks concerning out choice of $A = n_0 = \Theta(\log n)$. We said that we would think of A as fixed, but what happens if n varies? One standard solution is a complete linear time rebuilding of the priority queue every time n changes by a factor 2. However, the only reason that we picked A as growing with n is that we wanted it to be non-constant for simple asymptotic calculations like $O(n_i^{5/6}) < n_i$. However, with exact calculations, all these inequalities are satisfied for some sufficiently large constant A, and we can therefore just use this fixed constant value for A. Theoretically, this means that we could just use an unsorted list for the min-priority queue, but from a practical perspective this would be provokingly bad.

Concluding our amortized construction, we have shown:

Theorem 5. *We can implement a priority queue that with capacity for n integer keys in the range* $[0, N)$ *in linear space supporting* find-min, insert, *and* dec-key *in constant amortized time, and* delete *in* $O(1 + \log\frac{\log n}{\log q}) = O(\log\log n)$ *amortized time.*

Together with Lemma 3 this gives us an amortized version of Theorem 1. Also, plugged into Dijkstra's algorithm, Theorem 5 and Lemma 3 give us the single source shortest paths results in Corollary 2.

4. The worst-case efficient priority queue

In this section, we will show how to de-amortize the construction from the previous section, so as to get worst-case bounds of Theorem 1. A first simple change is to implement the min-priority queue with a relaxed heap [13] instead of a Fibonacci heaps [15] so as to get worst-case times for

the min-priority queue. Similarly, we get worst-case time for the directory using the q-heap from [17] instead of the atomic heap from [17].

Our fundamental problem for worst-case bounds is that we cannot take the priority queue down for repair so as to redistribute keys between local sets and levels. All this has to happen as back-ground processes while supporting find-min, insert, and dec-key in constant time and delete in $O(\log\log n)$ time.

4.1. Distributing between local sets

For each level i, we want a system so that update keys can be injected in constant time. The local sets will be kept at sizes between $n_i^{2/3}/4$ and $n_i^{2/3}$. Also, we will have a maximum of $n_i^{1/2}$ update keys outside the local sets in the buffer B_i.

The buffer B_i is divided into two chambers, B_i^0 and B_i^1, each with up to $n_i^{1/2}/2$ update keys. The buffer system works in rounds of $n_i^{1/2}/2$ *buffer steps*, each taking constant time. When a round begins, B_i^0 is empty while B_i^1 has up to $n_i^{1/2}/2$ update keys. During a round, we accept new update keys in B_i^0, at most one for each buffer step. Meanwhile, we scan the list of $\Theta(n^{1/3})$ local splitters, split the update keys from B_i^1 over the scanned splitters, and update the local sets with the update keys from B_i^1. As we shall discuss below, the whole emptying of B_i^1 into the local sets take $O(n_i^{1/2})$ time, so this can be done in constant time per buffer step. At the end of the round, B_i^0 has at most $n_i^{1/2}/2$ update keys while B_i^1 is empty, so to get ready for the next round, we simply swap the two.

We now need to argue that the emptying of B_i^1 into the local sets takes $O(n_i^{1/2})$ time. The scanning of local splitters only takes $O(n_i^{1/3})$ time. Also, since $|B_i^1| < n_i^{1/2}$, by Lemma 4, we can split B_i^1 over the scanned local splitters in $O(n_i^{1/2})$ time. The scanning and splitting is distributed over the first half of the buffer round, that is over $n_i^{1/2}/4$ constant time buffer steps. We now want to place two update keys in each of the remaining $n_i^{1/2}/4$ buffer steps. Having done the splitting, this would be trivial if the local splitters did not change. However, the splitting only distributes the buffer keys on the scanned local splitters, and meanwhile, the local splitter list may have changed.

We want to show how in constant time, we can place each update key from a scanned local splitter in a current local set while keeping the sizes of the local sets in balance. For this we will use a general balancing schedule from [5, Section 3.2]. The schedule tells us when we should start joining and splitting local sets. The joins and splits are processes that are implemented over multiple updates to the priority queue. During such a process, the involved local sets cannot participate in any other join or split processes.

Each time we place an update to a local set, if there is a join or split process involving the local set or one of its neighbors, we will spend constant time on a *balance step* in that process. We note here that each buffer step can place two update keys, hence perform two balance steps. Setting $b = n_i^{2/3}/60$, $\Delta = 1$, and $\mu = 15$ in Proposition 15 from [5], we get $b = n_i^{3/2}/60$ balance steps to complete each process while maintaining local set sizes between $\mu b = n_i^{2/3}/4$ and $(3\mu + \Delta + 14)b = n_i^{2/3}$.

Now, consider a join of two neighboring local sets X and X'. Let s and s' be the local splitters in front of them, hence with s' between X and X'. The join, setting $X = X \cup X'$, removes s' from the local splitter list. This takes constant time. However, s' may not be out of the picture yet, for the current buffer round may already have scanned s'. Then, when it splits $B_i^!$ over its scanned local splitters, some update keys may arrive s'. So far, these are just forwarded to s in constant time.

We need, however, to argue that a current buffer round involving s' will terminate before the current join process, hence that no updates will arrive at s' after the join is finished. As stated earlier, we have $n_i^{2/3}/60$ balance steps to complete a join process, and each balance step is caused by an update key to a nearby local set. Since each buffer round sends at most $n_i^{1/2}/2$ update keys to the local sets, there must be many buffer rounds during the current join process. More precisely, we will have at least $\lfloor (n_i^{2/3}/60)/(n_i^{1/2}/2) \rfloor = \Omega(n_i^{1/6}) = \omega(1)$ buffer rounds during the current join process. Thus we conclude that the current buffer rounds terminates before the current join process, hence that s' can be removed completely when the join process finishes.

For a split of a local set X, we first find a median s' to split X. While finding the median, we put new update keys to X aside. The median is found in $O(n_i^{2/3})$ time, and we divide this over the first half of the $n_i^{2/3}/60$ balance steps, spending constant time in each of these $n_i^{2/3}/120$ balance steps, as required. Now, s' is added to the local splitter list, to be included in subsequent scans. During the remaining $n_i^{2/3}/120$ balance steps, we take all keys in X and all update keys to X, including those we put aside, and split them over s' into X and X'. This is at most $n_i^{2/3} + n_i^{2/3}/60$ keys and update keys, so the splitting over s' can be done in constant time per balance step. Of importance for the scheduling from [5, Proposition 15], we note that the difference in size between X and X' is due to update keys, hence of size at most $n_i^{2/3}/60 = \Delta b$ (actually, it could be $1/2$ more if X was of odd size initial size, but, technically, we can always apply the first update before finding the median). As for joins, we have many buffer rounds during the last $n_i^{2/3}/120$ balance steps, and after the first one is finished, we will start receiving update keys at s' for X'. The split is thus reflected in the buffer scans before the end of the split process.

Summing up, the above is a buffer system that allows us support updates to level i in constant time while maintaining the sizes of the local sets between $n_i^{2/3}/4$ and $n_i^{2/3}$. Each update was implemented via a constant time buffer step, but we note that buffer steps can also be performed without an update. We know that any key currently in the buffer will be in the local sets by the end of the next buffer round, hence after at most $n_i^{1/2}$ buffer steps.

Finally, the scheduling from [5, Proposition 15] imposes some restrictions for the shifting between levels. When a local set created from level $i-1$ arrives the head of level i, it should be of size between $(\mu+3)b = \frac{3}{10} n_i^{2/3}$ and $(2\mu + \Delta + 9)b = \frac{2}{3} n_i^{2/3}$. Also, when we want to collect local sets to be shifted in or out, we cannot pick out arbitrary local sets. For example, we cannot pick out a local set in a join or split process. The schedule from [5, Proposition 15] combines local sets in uncuttable segments, each with a constant number of local sets, and with a total of between $(\mu+3)b = \frac{3}{10} n_i^{2/3}$ and $(5\mu + \Delta + 19)b = \frac{19}{12} n_i^{2/3}$ keys. The first or last uncuttable segment on level i can be identified and extracted in constant time.

4.2. Redistributing keys between levels

We are now going to describe how keys are shifted between level. The general, a shift process start by identifying the set S to be shifted with surrounding splitters. Then we perform buffer steps so as to flush buffers for updates to S. Finally, in constant time, we shift the set to the other level.

The shifting on each level will be divided into constant time *shift steps*. Every time we get an update to the priority queue, we do a shift step on some level, in a round-robin fashion: we have a level counter j, perform a shift step on level j, and increment j, or set j to 0 if j is the maximal level. Also, every time we delete a key below s_{i+1}, we do a shift step on level i. For example, a delete below s_1 causes a shift step on all levels.

When a shift step is performed on a level i, it goes to any current shift process on level i. If level i is not currently shifting, we check the number of keys. If it is above n_i, we start shifting out. If i is not the last level and the number is below $0.5n_i$, we start shifting in.

To describe the shifting, we will use auxiliary variables s_i^- and s_i^+ with $s_i^- \leqslant s_i \leqslant s_i^+$. When shifting in or out on level i, the affected keys will be between s_{i+1}^- and s_{i+1}^+. When done, we will have $s_{i+1}^- = s_{i+1} = s_{i+1}^+$. We will show that $s_i^+ \ll s_{i+1}^-$, hence that the shiftings on levels $i-1$ and i do interfere with each other.

4.2.1. Shifting out

Shifting keys out from level i to level $i+1$ is done as follows. First we collect uncuttable segments of local sets, starting from the tail, until we pass $0.5n_{i+1}^{2/3} = 0.5n_i^{5/6}$ keys. More precisely, in the first shift step, we set $S_i^{\text{out}} = \emptyset$ and $s_{i+1}^- = s_{i+1}$. In each subsequent shift step, we take the last uncuttable segment U, which is before s_i^-, and join it to S_i^{out}, setting s_{i+1}^- to be the local splitter before U. Recall here U consists of a constant number of local sets. Even if some of these are part of join and split processes, we can halt these processes and join them with S_i^{out} in constant time. We stop the expansion of S_i^{out} when it has $0.5n_i^{5/6}$ or more keys. This fixes s_{i+1}^-. Throughout the shift-out process, all new updates to level i and all updates coming out of B_i are compared with s_{i+1}^-, and if not smaller, transferred to S_i^{out}.

When the expansion of S_i^{out} is completed, we perform a buffer step on B_i in each of the subsequent $n_i^{1/2}$ shift steps. This flushes B_i for all updates to S_i^{out}. The priority queue contains no more updates in $[s_{i+1}^-, s_{i+1})$.

In a last shift step of the shift out, we set $(s_{i+1}, s_{(i+1)1}) = (s_{i+1}^-, s_{i+1})$, that is, we replace s_{i+1} with s_{i+1}^- in the level splitter list and turn the old s_{i+1} into the first local level $i+1$ splitter. Now, S_i^{out} is the local level $i+1$ set between s_{i+1} and $s_{(i+1)1}$. We also set $s_{i+1}^+ = s_{i+1}$. We note that if i was the last level, the above creates level $i+1$ with a single local set S_i^{out}. We then set s_{i+2}^-, s_{i+2}^-, and s_{i+2}^+ to ∞. This completes the last shift step in the shift out process.

4.2.2. Shifting in

The shift-in process on level i goes as follows. In the first shift step, we pick S_i^{in} as the join of the local sets in the first uncuttable segment on level $i+1$, and let s_{i+1}^+ be the subsequent local splitter.

From now on, all new updates to level $i+1$ and all updates coming out of B_{i+1} are compared with s^+_{i+1}, and if smaller, transferred to S^{in}_i. We note that if we are currently shifting out on level $i+1$, the same updates are also compared with s^-_{i+2}, but the overhead of these comparisons and possible transfers remains constant per buffer step and new update. In this cases both levels may work on flushing B_{i+1}, but this can only speed up the flushing.

During each of the subsequent $n_{i+1}^{1/2} = n_i^{5/8}$ shift steps on level i, we perform a buffer step on level $i+1$, thus flushing B_{i+1} for updates to S^{in}_i. Now S^{in}_i is completely updated with all keys in $[s_{i+1}, s^+_{i+1})$. In the next shift step, we set $s_{i+1} = s^+_{i+1}$ in constant time. The set S^{in}_i is put on the side on level i. If level $i+1$ was the last level, level i is the new last level.

The set S^{in}_i is now viewed as a kind of extra buffer on level i but which only has keys in $[s^-_{i+1}, s_{i+1})$. Physically, S^{in}_i is just a list. When a new update key x arrives level i, it is compared with s^-_{i+1}. If x is smaller, it is put directly in B_i. If $x \geqslant s^-_{i+1}$, we put it last in S^{in}_i and put the first key from S^{in}_i in B_i. Moreover, for each shift step on level i, we take the two first keys from S^{in}_i and put them in B_i, performing two buffer steps on B_i. When S^{in}_i is empty, we finish the shift in process setting $s^-_{i+1} = s_{i+1}$.

A few remarks are in place to appreciate the above design. At the moment we start emptying S^{in}_i, it contains all keys and updates in $[s^-_{i+1}, s_{i+1})$. As S^{in}_i gets emptied via B_i, we will get local sets with keys from $[s^-_{i+1}, s_{i+1})$. The reason why new updates in $[s^-_{i+1}, s_{i+1})$ are put behind the keys in S^{in}_i is that if they are deletes of keys in S^{in}_i, we need to make sure that they arrive the local sets after the keys they are supposed to delete.

4.3. Correctness

To argue correctness of the above system of shifts, we will prove a number of invariants for all i:

(i) At any time, the number of keys below any of $s^-_{i+1}, s_{i+1}, s^+_{i+1}$ is $\Theta(n_i)$, or lower if the splitter is last and infinite.
(ii) When level i finishes a shift process, the number of keys below s_{i+1} is between $0.5n_i + \Theta(n_i^{5/6})$ and $n_i - \Theta(n_i^{5/6})$, or anywhere below $n_i - \Theta(n_i^{5/6})$ if i becomes the last level.
(iii) Between shift processes or from the creation to the first shift, the number of keys below s_{i+1} is between $0.5n_i + \Theta(n_i^{4/5})$ and $n_i + \Theta(n_i^{4/5})$, except that the number may be lower if i is the last level.
(iv) Splitter s_{i+1} splits keys on or below level i from those above.
(v) We have multiple uncuttable segments between s^+_i and s^-_{i+1} unless i is the last level.

We will prove the invariants in parallel. We assume that the algorithm halts as soon as some invariant gets violated. Hence, while running, there has been no past violations. Also, when disproving a first violation of an invariant for level i, we assume that all preceding invariant are not violated, and that the invariant itself is not violated for smaller values of i.

We start by proving (iv). Since the algorithm carefully flushes buffers before moving s_{i+1}, the only worry is the situation where we increase s_{i+1} to s^+_{i+1} while having a set S^{in}_{i+1} of level $i+1$ keys

shifted in from level $i+2$. Since we do not flush S^{in}_{i+1}, it could potentially contain a key below s^+_{i+1}. However, while S^{in}_{i+1} is non-empty, it is contained in $[s^-_{i+2}, s_{i+2})$. Then $s^-_{i+2} \leqslant \min S^{\text{in}}_{i+1}$ is finite so by (i) we get $s^+_{i+1} < s^-_{i+2}$. Thus increasing s_{i+1} to s^+_{i+1} does not violate (iv).

Next we prove (v). From (i) it follows that there are $\Theta(n_i)$ keys in $[s^+_i, s^-_{i+1})$. By (iv), all buffer keys in $[s_i, s_{i+1})$ have to be in B_i, which has $O(n_i^{1/2})$ keys. Thus we have $\Theta(n_i)$ keys in the local sets in $[s^+_i, s^-_{i+1})$. There are $\Theta(n_i^{2/3})$ such keys per uncuttable segments, so there are $\Theta(n_i^{1/3})$ uncuttable segments in $[s^+_i, s^-_{i+1})$.

Between shifts We are now going to show (iii); namely, that between shift processes or from the creation to the first shift process, the number of keys below s_{i+1} is between $0.5n_i + \Theta(n_i^{4/5})$ and $n_i + \Theta(n_i^{4/5})$. For now, we assume that i is not the last level. By (ii), we had between $0.5n_i + \Theta(n_i^{5/6})$ and $n_i - \Theta(n_i^{5/6})$ keys below s_{i+1}, so (iii) is easily satisfied. There are only $O(\log\log n)$ updates till the next shift step on level i, and this is not enough to violate (iii).

Now, if the next shift step does start shifting in or out, it is because the number of keys in $[s_i, s_{i+1})$ is between $0.5n_i$ and n_i. By (i) on $i-1$, there are $\Theta(n_{i-1}) = \Theta(n_i^{4/5})$ keys below s_i. Including $O(\log\log n)$ updates till the next shift, the number of keys below s_{i+1} remains between $0.5n_i + \Theta(n_i^{4/5})$ and $n_i + \Theta(n_i^{4/5})$. This completes the proof of (iii) if i is not the last level. If i the last level, we note that it is created by a shift out from level $i-1$ with $\Theta(n_i^{2/3})$ keys. We can then apply the proof from when i is not last but ignoring the lower-bounds. This completes the proof of (iii).

Between shift processes, $s^-_{i+1} = s_{i+1} = s^+_{i+1}$ and then (iii) implies (i). However, since (i) is the first invariant, we cannot immediately assume any of the other invariants in its proof. For example, we cannot use (ii) to argue that (i) is satisfied when a shift process finish. However, after that, in the period till the next shift step, we can assume that (ii) was not previously violated, and then our argument for (iii) applies to (i). Also, for the period between shift steps, we only used (i) on $i-1$ to prove (iii), and this is OK for proving (i) on i. Thus we may conclude that (i) is satisfied from after a shift process finishes and till the next shift process starts.

Shifting out We will now analyze the numbers of the shift out process assuming that no invariant get violated. We know we start shifting because we have more than n_i keys in $[s_i, s_{i+1})$. Meanwhile, by (iii), we start with at most $n_i + \Theta(n_i^{4/5})$ keys below s_{i+1}. Also, we start with $s^-_{i+1} = s_{i+1} = s^+_{i+1}$.

We will argue that we cannot spend more than $2n_i^{1/6}$ shift steps on collecting S^{out}_i. Since (v) holds, each shift step successfully collects an uncuttable level i segment with at least $0.3n_i^{2/3}$ keys. In $2n_i^{1/6}$ shift steps, we collect $0.6n_i^{5/6}$ keys. Meanwhile, we may loose keys due to deletes, either from the buffer B_i or from new updates. The buffer has at most $n_i^{1/2}$ updates, and during $2n_i^{1/6}$ shift steps, we have at most $O(n_i^{1/6}\log\log n)$ new updates, so even if all these are deletes from S^{out}_i, it ends up with $(0.6 - o(1))n_i^{5/6}$ keys, contradicting $|S^{\text{out}}_i| < 0.5n_i^{5/6}$. Thus the collection of S^{out}_i stops within $2n_i^{1/6}$ shift steps. In this period, the number of keys below s_{i+1} can change by $O(n_i^{1/6}\log\log n)$.

For the collection of S_i^{out}, we collect an uncuttable segment of $O(n_i^{2/3})$ keys in each shift step. Moreover, we have $O(\log\log n)$ new updates to S_i^{in} between shift steps, so the collection ends up with $0.5n_i^{5/6} + O(n_i^{2/3})$ keys in S_i^{out}. During and after the collection, we have s_{i+1}^- in front of S_i^{out} and $s_{i+1} = s_{i+1}^+$ behind S_i^{out}. Including the $O(n_i^{1/2})$ potential updates in B_i, the number of keys in $[s_{i+1}^-, s_{i+1})$ is now $0.5n_i^{5/6} + O(n_i^{2/3}) \pm O(n_i^{1/2})$.

Subsequently, we flush B_i over at most $n_i^{1/2}$ shift steps. This gives $O(n_i^{1/2}\log\log n)$ new updates which may affect any interval. The number of keys in S_i^{out} is then $0.5n_i^{5/6} + O(n_i^{2/3}) \pm O(n_i^{1/2}\log\log n)$. This is within the legal size range of $[\frac{3}{10}n_{i+1}^{2/3}, \frac{19}{12}n_{i+1}^{2/3}]$ for shifting S_i^{out} out as a new local set on level $i+1$, setting s_{i+1} and s_{i+1}^+ to s_{i+1}^-.

Adding up all the numbers, we see that we finish with at most $n_i + O(n_i^{4/5}) + O(n_i^{1/6}\log\log n) - 0.5n_i^{5/6} + O(n_i^{1/2}) + O(n_i^{1/2}\log\log n) = n_i - \Theta(n_i^{5/6})$ keys below s_i, satisfying (ii). Also, we see that throughout the shifting, we have between $n_i - O(n_i^{5/6})$ and $n_i + O(n^{4/5})$ keys below each of s_{i+1}^-, s_{i+1}^-, and s_{i+1}^+, satisfying (i).

We now check that the above argument for (i) and (ii) is not cyclic. First, it is valid to assume that (i) and (iii) has not been violated by some action before we start shifting out, and the start of the shift cannot in itself violate these bounds. In particular, this implies that our initial use of (iii) is valid.

Next, concerning the use of (v), the argument goes as follows. We only used (v) to argue that there would always be an uncuttable segment to include in S_i^{out}. To reach a contradiction, consider the first time at which there is no such uncuttable segment. At this point, there has been no problems in the past, so our the accounting for (i) is valid. Thus, (i) is true. But then (v) is also true, contradicting that there is no uncuttable segment to collect for S_i^{out}.

Shifting in Next we analyze the numbers of the shift in process assuming that no invariant gets violated. We only start shifting in if level i is not last and has less than $0.5n_i$ keys in $[s_i, s_{i+1})$. By (i) on $i-1$, this means that we have at most $0.5n_i + \Theta(n_{i-1}) = 0.5n_i + \Theta(n_i^{4/5})$ keys below s_{i+1}. This matches the lower-bound in (iii), so we conclude that the number of keys below $s_{i+1}^- = s_{i+1} = s_{i+1}^+$ is $0.5n_i + \Theta(n_i^{4/5})$.

In the first shift step, we let S_i^{in} be the join of the first uncuttable segment on level $i+1$ and move s_{i+1}^+ to the other side of S_i^{in}. For now, we assume that level $i+1$ is not the last level. Then the uncuttable segment exists by (v) and contains $\Theta(n_{i+1}^{2/3}) = \Theta(n_i^{5/6})$ keys. Next we flush B_{i+1} over $\Theta(n_{i+1}^{1/2}) = \Theta(n_i^{5/8})$ shift steps, during which we may have $O(n_i^{5/8}\log\log n)$ new updates. Yet, we preserve $|S_i^{\text{in}}| = \Theta(n_{i+1}^{2/3}) = \Theta(n_i^{5/6})$.

In the next shift step, we set $s_{i+1} = s_{i+1}^+$, and the set $S_i^{\text{in}} \subseteq [s_{i+1}^-, s_{i+1})$ is put aside on level i. Let $t = \Theta(n_i^{5/6})$ be the current size of S_i^{in}. We know that S_i^{in} is emptied within the next $t/2$ shift steps on level i. During these, we can have $O(t\log\log n)$ global updates, and out of these, at most $t/2$ deletes of keys below s_{i+1}. Consequently, the minimum number of keys below s_{i+1} ends at $0.5n_i + \Theta(n_i^{4/5}) + t - O(n_i^{5/8}\log\log n) - t/2 = 0.5n_i + \Theta(n_i^{5/6})$. Similar calculations show that the number of keys below any of s_{i+1}^-, s_{i+1}, and s_{i+1}^+ remain between $0.5n_i + \Theta(n_i^{4/5})$ and $0.5n_i + \Theta(n_i^{5/6})$. Thus

we have shown the statements of (ii) and (i) assuming that level $i+1$ was not last. If level $i+1$ was last, it may consist of a single local set with $O(n_i^{5/6})$. In that case, this whole set gets shifted in, destroying level $i+1$ and setting $s_{i+1}^- = s_{i+1} = s_{i+1}^+ = \infty$. In that case, we only get an upper-bound of $0.5n_i + \Theta(n_i^{5/6})$ on the number of keys on level i, but that suffices for (ii) and (i) on the last level.

The argument that our usage of (iii) and (v) is valid is identical to the one used for shifting out. Thus, we have argued that all our invariants are preserved, hence the correctness of our priority queue algorithm.

Time bounds To analyze the time bounds, it is immediate that we only spend constant time on each of find-min, insert, and dec-key. The minimum key is found in constant time from the min-priority queue, and insert and dec-key only perform a constant number of buffer and shift steps on two levels.

Concerning delete(x), we only spend constant time on a shift step on each level i with $x \leqslant s_{i+1}$. From (i), we know that there are $\Theta(n_i)$ keys below s_{i+1}. This is the same bound we used in our amortized analysis, so again we conclude that if x has rank q, the number of levels with $x \leqslant s_{i+1}$ is $O(1 + \log\log n - \log\log q)$. This is hence the time we spend on shifting in connection with delete(x). This time bound also dominates the $O(\log\log n)$ time it takes to delete from the min-priority queue when $x \leqslant s_0$ and $q \leqslant A$.

Concerning the fixedness of $A = n_0 = \Theta(\log n)$, we have the same choices as in the amortized version: we can do complete rebuilding in the back-ground while n changes by a factor 2, or we can replace A with a large enough constant that our asymptotic calculations go through with exact numbers. This completes the proof of our worst-case result:

Theorem 6. *We can implement a priority queue that with n integer keys in the range* $[0, N)$ *in linear space supporting* find-min, insert, *and* dec-key *in constant time, and* delete *in* $O(1 + \log\log n - \log\log q)$ *time for a key of rank* q.

Theorem 6 and Lemma 3 together establish Theorem 1, which is the main result of this paper.

5. An AC^0 priority queue

In this section, we will develop an efficient priority queue where only standard AC^0 functions are supported on words. This includes functions like addition, shifts, and bit-wise Boolean operations but excludes multiplication. The two places where we currently use multiplication is in the atomic priority queues of Fredman and Willard [17] and in the linear time splitting of Han and Thorup [20].

First, we will show how to bypass the atomic heap. Next we point to a classic monotone priority queue construction using a non-monotone priority queue for short keys. Finally, we construct an AC^0 priority queue that works efficiently for these shorter keys.

5.1. Avoiding the atomic heap

Currently, we use an atomic heap in the directory, and Thorup [29] has shown that atomic heaps cannot be implemented with AC^0 operations only. However, in the directory, we can use

splitting instead of the atomic heap. We will use a buffer system similar to the one currently devised for each level. More precisely, we are going to divide updates in rounds of $\Theta((\log\log n)^2)$ updates were we work on keys at three different places. When a new updates is made, if the key is below s_1, it is sent directly to level 0. Otherwise, it is placed in a directory buffer B. At the end of a round, we switch B with an empty buffer. In a round, we also scan the current list of $O(\log\log n)$ level splitters, and use them to split the keys from the buffer of the last round. The splitting takes linear time, or constant time per update. The result is a set S of updates that are split between the scanned level splitters. Finally, in a round, every time a new update is made, we pick an update key x from the set S split it the last round, and inject it on the appropriate level. If the key x was on level i according to the scanned level splitters, since rounds only take $\Theta((\log\log n)^2)$ updates, the key x will be on level $i-1$, i, or $i+1$ with the current level splitters. Hence, to find the current level of x, we just need to compare x with s_i and s_{i+1}.

To see that the above works correctly, we have to argue that the directory cannot contain the smallest key in the priority queue. We know that when a key x was placed in the directory, it was as big as s_1, hence that there were $\Theta(\log n)$ keys below x. We know that x leaves the directory within $O(\log\log n)$ updates, and these updates include updates below s_1 even though these are not kept in the directory. Hence we will always have $\Omega(\log n - \log\log n) = \Omega(\log n)$ keys below x while x is in the directory.

We have now bypassed the atomic heap of Fredman and Willard. The only place left where we use multiplication is in the linear time splitting of Han and Thorup.

5.2. *Using a priority queue for short keys in a full-word monotone priority queue*

Our goal is to construct an efficient monotone priority queue Q, that is, a priority queue where the minimum is assumed to be non-decreasing. We construct such a monotone priority queue from a non-monotone priority queue for much shorter keys. The basic idea goes back to Denardo and Fox [10], and was further developed by Ahuja et al. [1]. Our simple variant of the construction has the advantage of being implemented in linear space using only AC^0 operations.

We view our full-word key x as consisting of q characters, enumerated $0,\ldots,q-1$ and with character $x[0]$ being the least significant.

Let μ be current minimum in the queue. For each key x, we consider a derived key x^μ whose value is a pair (i,a) where i is the most significant character in which x and μ differ and $a = x[i]$. If $x = \mu$, we set $i = 0$. Above, we call *i the index* and *a the character* of the derived key $x^\mu = (i,a)$ The derived keys are lexicographically ordered. Clearly

$$x^\mu < y^\mu \Rightarrow x < y.$$

By monotonicity, if μ changes, it increases, and this can only decrease x^μ. In more detail, let $\mu \leqslant \mu' \leqslant x$, $x^\mu = (i,a)$, $x^{\mu'} = (i',a')$. Then

$$x^{\mu'} \neq x^\mu \Leftrightarrow x^{\mu'} < x^\mu \Leftrightarrow \mu'[i] = x[i] \Leftrightarrow i' < i$$

Note that we can have at most q decreases to a derived key due to increases in the minimum μ; for each such decrease in the derived value decreases the index.

We will use a priority queue Q^μ over the derived keys. These derived keys have only $c = \log q + W/q$ bits. The priority queue Q^μ will not be monotone. We assume that Q^μ supports insert

and dec-key in amortized constant time. Finally, it is important that Q^μ supports an extended find-min that can list all keys of minimal value in constant time per key listed.

It is an observation of this paper that such a listing of keys of minimal value allow us to bypass the standard use of buckets for keys with the same value [1,10]. Such a general identification of keys with the same value would have given us space problems when restricted to AC^0 operations [4].

When a key x is inserted, we insert x^μ in Q^μ. When x is decreased, we decrease x^μ accordingly in Q^μ. Each of these operations take constant amortized time.

When a key x is deleted, the derived key x^μ is deleted from Q^μ in some amortized worst-case time t. If $x = \mu$, we need to find a new minimum key. First, let $x^\mu = (i, a)$ be the new minimum key in Q^μ.

If $i = 0$, then x is a new minimum key in Q. We then set $\mu = x$, and note that this does not affect the derived character of any key.

If $i > 0$, we identify the set X of all keys with minimum derived value (i, a). Among these we find our new smallest key μ'. The keys in X are exactly the keys whose derived value decreases with a smaller index. For each $x \in X$, we compute $x^{\mu'}$ for a decreased value in $Q^{\mu'}$.

The keys in X are found and decreased in constant time per key, and since this only happens q times per key, we conclude that we spend constant amortized time per dec-key operation, and $O(q + t)$ per key.

A general technique of Alstrup et al. [2] allow us to amortize the per key cost over deletes, even if most keys are not deleted. That is, we get insert in constant amortized time and delete in $O(q + t)$ amortized time. Summing up, we have

Proposition 7. *Let W be the word-length and q a parameter. Suppose we have a linear space priority queue for short keys of length $q + W/q$ supporting* insert *and* dec-key *in amortized constant time and* delete *in amortized time t, and which further can list all keys of minimum value in constant time per key listed. We can then construct a linear space monotone priority queue for full-word keys supporting* insert *and* dec-key *in amortized constant time and* delete *in amortized time $O(q + t)$ time. The reduction uses standard AC^0 operations only.*

5.3. A fast priority queue for short keys

We will now obtain an efficient priority queue for short keys. Han and Thorup [20] have shown that the splitting of Lemma 4 can be performed in linear time using only standard AC^0 operations if the keys are short with $\ell = O(W/(\log\log n)^{1+\varepsilon})$ bits. Here ε is a positive constant and W is the word-length of the computer. Since we have already bypassed the atomic heaps, this essentially gives us an AC^0 implementation of our priority queue from Theorem 6. We do, however, note that our construction assumed that each key is suffixed with a unique $O(\log n)$ bit identifier. If $\ell = \Omega(\log n)$, this does not affect the key length in the splitting. On the other hand, if $c = O(\log n)$, we can sort the keys and the splitters in linear time using radix sort. Thus we have an AC^0 implementation for short keys of the priority queue from Theorem 6.

Besides the operations supported in Theorem 6, we want to list all keys with minimum value. Settling for amortized bounds, first we consider the Fibonacci heaps in the min-queue. Referring

to the presentation of Fibonacci heaps in [9], each key in the heap-ordered trees should maintain the children with the same value. Also, we maintain a list of all roots with minimum key value. This list is easily maintained, for if a root is deleted, the logarithmic delete time allows us to scan all the remaining roots to produce a new minimum root list from scratch. It is now trivial to list all the minimum keys in the min-heap.

For the remaining keys in our priority queue, first for each local set, we store the members with the same value as the preceding local splitter. Then, to list the keys of minimum value in all the local sets, we scan the list of local splitters, stopping when we get to a local splitter of larger value. Finally, concerning keys in the dictionary and in the level buffers, we note that we only need to consider the dictionary if most of the $\Theta(\log n)$ keys below the first splitter s_1 are minimum, but then we can afford to scan all the $O((\log\log n)^2)$ keys in the directory. Similarly, we only need to consider the buffer B_{i+1} if most of the level i keys are minimum, and on level i we have $\Theta(n_i)$ keys, whereas the buffer B_{i+1} only has $O(n_{i+1}^{1/2}) = O(n_i^{5/8})$ keys, all of which we can afford to scan.

Thus we have

Lemma 8. *Using only standard AC^0 operations, we can implement a linear space priority queue that with $O(W/(\log\log n)^{1+\varepsilon})$-bit keys and $\log n$-bit identifiers in linear space supporting* find-min, insert, *and* dec-key *in constant time, and* delete *in $O(\log\log n)$. The* find-min *operation can also list all keys of minimum key value in linear time.*

We now combine Lemma 8 with Proposition 7. In Proposition 7, we set $q = (\log\log n)^{1+\varepsilon}$. Since $W \geqslant \log n$, this gives us a short key length of $\log q + W/q = O(W/(\log\log n)^{1+\varepsilon})$. Then Lemma 8 gives an amortized delete time of $O(\log\log n)$ for short keys. Back in Proposition 7, this gives us an overall amortized delete time of $O((\log\log n)^{1+\varepsilon})$. Combining this with Lemma 3, we get

Theorem 9. *Using only standard AC^0 operations, we can implement a monotone priority queue that with n integer keys in the range $[0, N)$ in linear space supporting* find-min, insert, *and* dec-key *in constant amortized time, and* delete *in $O((\log\log\min\{n, N\})^{1+\varepsilon})$ amortized time. In particular, we can solve the single source shortest path problem for a directed graph with n nodes and m edges with weights in the range $[0, C)$ in linear space and $O(m + n(\log\log\min\{n, C\})^{1+\varepsilon})$ time.*

6. Concluding remarks

We have presented a priority queue that maintains the minimum in a dynamic set of integer keys. Keys can be inserted and decreased in constant time. If the current number of keys is n, a key can be deleted in $O(\log\log n)$ time. If the integers are in the range $[0, N)$, we can also support delete in $O(\log\log N)$ time. As a classical application, for a directed graph with n nodes and m edges with non-negative integer weights, we get single source shortest paths in $O(m + n\log\log n)$ time, or $O(m + n\log\log C)$ if C is the maximal edge weight. The latter solves an open problem of Ahuja, Mehlhorn, Orlin, and Tarjan from 1990.

Our priority queue can be used to sort n integers deterministically: first we insert them all, and second we extract them in sorted order. This takes $O(n \log\log n)$ time and linear space, matching the current best deterministic shorting bound which is due to Han [19]. Improving our delete time would thus lead to a better deterministic sorting algorithm.

However, Han and Thorup have shown how to sort in $O(n\sqrt{\log\log n})$ expected time, and with a general reduction from basic priority queues to sorting, Thorup [28] has shown that this implies a basic priority queue with delete in $O(\sqrt{\log\log n})$ expected time. Combining the latter with dec-key in constant time is left as a major open problem.

References

[1] R.K. Ahuja, K. Mehlhorn, J.B. Orlin, R.E. Tarjan, Faster algorithms for the shortest path problem, J. ACM 37 (2) (1990) 213–223.
[2] S. Alstrup, T. Husfeldt, T. Rauhe, Marked ancestor problems, Technical Report 98/8, Department of Computer Science, University of Copenhagen, 1998 (short version from FOCS'98 does not contain the reduction needed here).
[3] A. Andersson, Faster deterministic sorting and searching in linear space, in: Proceedings of the 37th FOCS, 1996, pp. 135–141.
[4] A. Andersson, P.B. Miltersen, S. Riis, M. Thorup, Static dictionaries on AC^0 RAMs: Query time $\Theta(\sqrt{\log n/\log\log n})$ is necessary and sufficient, in: Proceedings of the 37th FOCS, 1996, pp. 441–450.
[5] A. Andersson, M. Thorup, Dynamic ordered sets with exponential search trees, Technical Report cs.DS/0210006, The Computing Research Repository (CoRR), 2002. http://arXiv.org/abs/cs.DS/0210006.
[6] P. Beame, J. Håstad, Optimal bounds for decision problems on the CRCW PRAM, J. ACM 36 (3) (1989) 643–670.
[7] B.V. Cherkassky, A.V. Goldberg, C. Silverstein, Buckets, heaps, lists, and monotone priority queues, SIAM J. Comput. 28 (4) (1999) 1326–1346 (announced at SODA'97).
[8] L.J. Comrie, The hollerith and powers tabulating machines, Transactions of the Office Machinery Users Association Limited, 1929–30, pp. 25–37.
[9] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 2nd Edition, MIT Press, McGraw-Hill, 2001. ISBN 0-262-03293-7, 0-07-013151-1.
[10] D.V. Denardo, B.L. Fox, Shortest-route methods 1: reaching, pruning and buckets, Oper. Res. 27 (1979) 161–186.
[11] R.B. Dial, Algorithm 360: shortest path forest with topological ordering, Comm. ACM 12 (11) (1969) 632–633.
[12] E.W. Dijkstra, A note on two problems in connection with graphs, Numer. Math. 1 (1959) 269–271.
[13] J.R. Driscoll, H.N. Gabow, R. Shrairman, R.E. Tarjan, Relaxed heaps: an alternative to Fibonacci heaps with applications to parallel computation, Comm. ACM 31 (11) (1988) 1343–1354.
[14] A.I. Dumey, Indexing for rapid random access memory systems, Comput. Automat. 5 (12) (1956) 6–9.
[15] M.L. Fredman, R.E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, J. ACM 34 (3) (1987) 596–615 (announced at FOCS'84).
[16] M.L. Fredman, D.E. Willard, Surpassing the information theoretic bound with fusion trees, J. Comput. System Sci. 47 (1993) 424–436 (announced at STOC'90).
[17] M.L. Fredman, D.E. Willard, Trans-dichotomous algorithms for minimum spanning trees and shortest paths, J. Comput. System Sci. 48 (1994) 533–551 (announced at FOCS'90).
[18] T. Hagerup, Improved shortest paths on the word RAM, in: Proceedings of the 27th ICALP, Lecture Notes in Computer Science, Vol. 1853, 2000, pp. 61–72.
[19] Y. Han, Deterministic sorting in $O(n \log\log n)$ time and linear space, J. Algorithms 50 (1) (2004) 95–105 (announced at STOC'02).
[20] Y. Han, M. Thorup, Integer sorting in $O(n\sqrt{\log\log n})$ expected time and linear space, in: Proceedings of the 43rd FOCS, 2002, pp. 135–144.
[21] IEEE, Standard for binary floating-point arithmetic, ACM Sigplan Notices 22 (1985) 9–25.

[22] B.W. Kernighan, D.M. Ritchie, The C Programming Language, 2nd Edition, Prentice-Hall, Englewood Cliffs, NJ, 1988.
[23] D.E. Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching, 2nd Edition, Addison-Wesley, Reading, MA, 1998.
[24] R. Raman, Priority queues: small, monotone and trans-dichotomous, in: Proceedings of the Fourth ESA, Lecture Notes in Computer Science, Vol. 1136, 1996, pp. 121–137.
[25] R. Raman, Recent results on the single-source shortest paths problem, SIGACT News 28 (2) (1997) 81–87.
[26] M. Thorup, Undirected single source shortest paths with positive integer weights in linear time, J. ACM 46 (3) (1999) 362–394 (announced at FOCS'97).
[27] M. Thorup, On RAM priority queues, SIAM J. Comput. 30 (1) (2000) 86–109 (announced at SODA'96).
[28] M. Thorup, Equivalence between priority queues and sorting, in: Proceedings of the 43rd FOCS, 2002, pp. 125–134.
[29] M. Thorup, On AC^0 implementations of fusion trees and atomic heaps, in: Proceedings of the 14th SODA, 2003, pp. 699–707.
[30] P. van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space, Inform. Process. Lett. 6 (3) (1977) 80–82.
[31] P. van Emde Boas, R. Kaas, E. Zijlstra, Design and implementation of an efficient priority queue, Math. System Theory 10 (1977) 99–127.