DIMACS Technical Report 95-12
May 1995

Equivalence between sorting and priority queues

(extended abstract)

by

Mikkel Thorup[1]
Dept. of Computer Science
University of Copenhagen
Universitetsparken 1
2100 Kbh. Ø
Denmark
mthorup@diku.dk

[1]DIMACS Visitor, Summer 1995

DIMACS is a cooperative project of Rutgers University, Princeton University, AT&T Bell Laboratories and Bellcore.

DIMACS is an NSF Science and Technology Center, funded under contract STC–91–19999; and also receives support from the New Jersey Commission on Science and Technology.

ABSTRACT

For a RAM with arbitrary word size, it is shown that if we can sort n integers, each contained in one word, in time $n \cdot s(n)$, then (and only then) there is a priority queue with capacity for n integers, supporting `find-min` in constant time and `insert` and `delete` in $s(n) + O(1)$ amortized time. Here it is required that when we insert a key, it is not smaller than the current smallest key. The equivalence holds even if n is limited in terms of the word size w.

One application is an $O(n\sqrt{\log n}^{1+\varepsilon} + m)$ algorithm for the single source shortest path problem on a graph with n nodes and m edges.

1 Introduction

By a *priority queue* we mean a representation of a dynamic set, or multiset, X supporting `find-min` returning $\min X$, `insert`(x) setting $X := X \cup \{x\}$, and `delete`(x) setting $X := X \setminus \{x\}$. For `insert`(x) we require $x > \min X$. Thus, the minimum is non-decreasing over time. We shall refer to this as the *monotonicity* requirement, noting that it is not a problem for greedy algorithms like Dijkstra's shortest path algorithm, or Huffman coding. For simplicity, we assume that our queues are always non-empty. Our main result is a general equivalence on RAMs between sorting and plain priority queues.

Theorem 1 *For a RAM with arbitrary word size, if we can sort n integers, each contained in one word, in time $n \cdot s(n)$, where s is non-decreasing, then (and only then) there is a priority queue with capacity for n integers, supporting* `find-min` *in constant time and* `insert` *and* `delete` *in $s(n) + O(1)$ amortized time. The equivalence holds even if n is limited in terms of the word size w.*

To prove Theorem 1 we will show how to maintain a priority queue spending constant time on `find-min` and constant amortized time on `insert` and `delete` plus some sorting time. Each key passing through the priority queue participates in at most one sorting, and when we sort a set of keys, they are all contained in the current queue. Applying Theorem 1 to the recent sorting results from [2], we get

Corollary 2 *There is a RAM priority queue supporting* `find-min` *in constant time and* `insert` *and* `delete` *in*

a) $O(\log\log n)$ *amortized time.*

b) *constant amortized time if $(\log n)^{2+\varepsilon} \leq w$ for some $\varepsilon > 0$.*

Above b) is a new result while a) is already known from [14]. In fact, the result from [14] is stronger for several reasons. For example, the priority queue from [14] works in $O(\log\log n)$ worst case time, and it does not have the restriction on `insert`(x) that $x \geq \min X$. Also, the priority queue from [14] only requires AC^0 operations, and it is simple enough to be of practical relevance. Our reduction to sorting uses the "atomic" priority queue from [10, §2.3] which is complicated and uses operations like multiplication that are not in AC^0 [5].

By an *atomic* priority queue we mean a priority queue where all the above operations are done in constant amortized time, as in Corollary 2b). From [10], essentially we have the following result:

Lemma 3 *Provided atomic priority queues for up to $f(n)$ keys, there is a priority queue with capacity n, supporting* `find-min`, `insert`, *and* `decrease-key` *in constant amortized time, and supporting* `delete` *in time $O(\log n/\log f(n))$ time. Here* `decrease-key`(x, d) *sets $x := x - d$.*

Plain priority queue		With constant `decrease-key`	
$O(\log n)$	Williams '64 [15]	$O(\log n)$	Fredman & Tarjan '84 [8]
$O(\sqrt{\log n})$	Fredman & Willard '90 [9]	$O(\log n/\log\log n)$	Fredman & Willard '90 [10]
$O(\log\log n)$	Thorup '95 [14]	$O(\sqrt{\log n}^{1+\varepsilon})$	This paper

Figure 1: Brief history of the developments in priority queues

Proof: A constructive proof is provided in [10, §2.2], setting $B = f(n)/\log n$. Then the construction gives `delete` in time $O(\log n/\log B)$, but from [10, §2.3], we know that $f(n) = \Omega((\log n)^2)$, so $O(\log n/\log B) = O(\log n/\log f(n))$. The insertions from [10, §2.2] do not immediately respect monotonicity, but they all fall in two groups:

- During the "ripple" operation, a minimum x is deleted and a replaced by a larger element y. In order to preserve monotonicity, we just insert y before deleting x.
- During the "consolidate" operation, a set X of keys are turned into a priority queue. Here we initiate the queue with a dummy $0 \le \min X$, which we remove after inserting the elements from X.

With these minor modifications, the construction from [10, §2.2] proves the lemma. ■

In [10, §2.3] they provide an atomic priority queue with capacity $(\log n)^2$, giving `delete` in $O(\log n/\log\log n)$ amortized time. If $w \le \log n$, we can implement all our operations in amortized constant time by simple bucketing, so we may assume $w \ge \log n$. Then Corollary 2b) increases the capacity of atomic priority queues to $2^{\sqrt{\log n}^{1-\varepsilon'}}$ for any fixed $\varepsilon' > 0$. Thus

Corollary 4 *There is a priority queue supporting* `find-min`, `insert`, *and* `decrease-key` *in constant amortized time, and* `delete` *in amortized time $O(\sqrt{\log n}^{1+\varepsilon'})$ for any fixed $\varepsilon' > 0$.*

Figure 1 gives a brief sketch of the developments in priority queues. As an immediate consequence of Corollary 4, we get

Corollary 5 *For any $\varepsilon' > 0$, we can solve the single source shortest path problem on a graph with n nodes and m edges in time $O(n\sqrt{\log n}^{1+\varepsilon'} + m)$.*

The previous best bound linear in m was $O(n \log n/\log\log n + m)$, based on the priority queues from [10]. For small m, we have the complementing $O(n + m\log\log m)$ bound from [14].

It should be mentioned that our bounds generalize to integers occupying any constant number of words (if there is no bound on the number of words per integer, we need to add the time it takes to read the distinguishing prefixes [4]). As discussed in [2], multiple word integers are not an exotic special case. For example, the IEEE 754 floating-point standard is designed so that the ordering of floating point numbers can be deduced by perceiving their representations as multiple word integers [11].

Finally, it should be mentioned that we do not need to know n in advance. In [14, Proof of Theorem 4], based on standard doubling techniques, the details are given on how to dynamically increase/decrease the capacity of priority queues.

2 The general reduction

This section is devoted to the general proof of Theorem 1. Fix $k = \log n \log\log n$. We will assume:

(i) Universal tables, either by hashing [7] in linear space with expected constant look-up time, or for any $\varepsilon > 0$, in space $O(u^{\varepsilon})$ doing look-up by $1/\varepsilon$ comparisons (see discussion in [14, §4]).

(ii) Atomic priority queues for up to k keys. In [10] such queues are provided, which, in fact, take up to $(\log n)^2$ keys. They require $O(n)$ preprocessing time and space.

(iii) Atomic priority queues for (w/k)-bit keys. Such queues will be presented in the next section.

(iv) Your favorite sorting algorithm `sort`.

We will perceive the words as divided into k fields of size w/k. For any key x, we let $x[i]$ denote the contents of the ith field of the word containing x. Also, $x[i..j] = x[i] \cdots x[j]$. For any two keys x, y, by $\texttt{split}(x, y)$ we denote the first i such that $x[i] \neq y[i]$. Below, instead of implementing `delete` directly, for simplicity, we will start by just allowing `delete-min`, deleting the smallest element. The general `delete` operation will be included at the end of this section.

Our priority queue consists of the following components:.

- The smallest key μ in the queue. Thus `find-min` is trivially done in constant time. Note for any keys x, y in the queue that $\texttt{split}(\mu, x) > \texttt{split}(\mu, y)$ implies $x < y$.

- Some sorted lists $L_1, \ldots, L_k$ of keys where L_i satisfy $\forall x \in L_i : \texttt{split}(\mu, x) > i$. Note that $i < j$ implies that any key in L_j could also have been in L_i. Also note that if we replace μ by μ' and $\mu \leq \mu' \leq \min L_i$, then this does not violate the condition on L_i.

  Using an atomic priority queue from (ii), in constant amortized time, we can keep track of which of the list $L_1, \ldots, L_k$ that has smallest first key. This key is hence the smallest of all the keys in the lists. Thus $\min \bigcup_i L_i = \min_i\{\min L_i\}$ is computable in constant amortized time.

- From (iii) we get atomic priority queues $A_1, \ldots, A_k$ for w/k-bit keys—corresponding to the fields of our original keys. For $i = 1, \ldots, k$, it is required that $\mu[i] < \min A_i$. With each $a \in A_i$ is associated a set $S_i(a)$ of keys x such that $x[1..i] = \mu[1..i-1]a$. For the membership in A_i and the entries to $S_i(\cdot)$, we use the universal tables from (i).

  Let $\overline{A_i}$ denote $\bigcup_{a \in A_i} S_i(a)$. Note that $i > j$ implies $\max \overline{A_i} < \min \overline{A_j}$.

  We will also have an atomic priority queue keeping track of the maximum index i of a non-empty A_i.

Above $(\{\mu\}, L_1, \ldots, L_k, \overline{A_1}, \ldots, \overline{A_k})$ is understood to be a partitioning of the keys currently in the queue. We are now ready to implement `insert` and `delete-min`.

Algorithm A `insert`(x)

A.1. $i := \texttt{split}(x, \mu)$.

A.2. If $x[i] \notin A_i$ then $\texttt{insert}(x[i], A_i)$, setting $S_i(x[i]) := \emptyset$.

A.3. $S_i(x[i]) := S_i(x[i]) \cup \{x\}$.

Algorithm B `delete-min`

B.1. $\mu_L := \min \bigcup_h L_h$.

B.2. $i := \texttt{split}(\mu_L, \mu)$.

B.3. Let j be maximum such that A_j is non-empty.

B.4. $a := \min A_j$.

B.5. If $i < j$ or $(i = j$ and $\mu_L[i] \geq a)$ then

B.5.1. $L_j := \texttt{sort}(S_j(a))$.

B.5.2. Delete a from A_j.

B.5.3. $\mu_L := \min\{\mu_L, \min L_j\}$.

B.6. $\mu := \mu_L$.

B.7. Delete μ_L from its list.

To see the correctness of Algorithm B, in connection with the condition of step B.5, observe:

- If $i \leq j$, then $L_j = \emptyset$; for assume $L_j \neq \emptyset$. Then $\texttt{split}(\min L_j, \mu) > j$, and hence $\min L_j < \mu_L$ contradicting the minimality of μ_L. Thus we do not loose any elements when overwriting L_j in step B.5.1.
- The condition implies that the coming minimum key in the queue has prefix $\mu[1..j-1]a$. Thus after step B.6, L_j will correctly satisfy that $x[1..j] = \mu[1..j]$ for all $x \in L_j$.

Above we maintain the priority queue spending constant amortized time per `insert` or `delete-min` operation plus sorting. Each key participate in exactly one sorting; namely when it is transferred from $\bigcup_i \overline{A_i}$ to $\bigcup_i L_i$. Thus `sort` is only called with *disjoint* subsets of the keys, and whenever $\texttt{sort}(X)$ is called, X is a subset of the keys currently in the queue.

We will now augment our priority queue with the general `delete` operation. We will use a universal table $D \subseteq \bigcup_i L_i$ denoting keys that are to be deleted. Then `delete` may be implemented as follows.

Algorithm C `delete`(x)

C.1. If $x = \mu$ then

C.1.1. $\mu_L := \min \bigcup_h L_h$.

C.1.2. While $\mu_L \in D$,

C.1.2.1. Delete μ_L from D and from its list L_h.

C.1.2.2. $\mu_L := \min \bigcup_h L_h$.
C.1.3. `delete-min`
C.2. else
C.2.1. $i :=$ `split`(x, μ)
C.2.2. If $x[i] \in A_i$ then
C.2.2.1. $S_i(x[i]) := S_i(x[i]) \setminus \{x\}$.
C.2.2.2. If $S_i(x[i]) = \emptyset$, delete $x[i]$ from A_i.
C.2.3. else
C.2.3.1. $D := D \cup \{x\}$.

For space reasons, if ever $|D| \geq |\bigcup_i L_i|/2$, we scan through all the L_i deleting the keys from D, and setting $D := \emptyset$ at the end. The cost of the scan is attributed to the `delete` operations that filled D. Thus still we maintain the priority queue spending constant amortized time per operation plus sorting. Sorting is only called on disjoint subsets of keys, and whenever `sort`(X) is called, X is a subset of the keys currently in the queue. This completes the proof of Theorem 1, assuming the atomic priority queue from (iii), to be described in the next section.

3 An atomic priority queue for small integers

From [3], we have the following result:

Proposition 6 *We can merge two sorted lists, each of at most k keys stored in a single word, into a single sorted list stored in two words in time $O(\log k)$.*

Corollary 7 *For $n \geq k$, given two lists of n (w/k)-bit keys, spread over $\lceil n/k \rceil$ words, we can merge them into $\lceil 2n/k \rceil$ words in time $O(n/k \cdot \log k)$.*

Corollary 7 is implicit in [3] in a parallel setting (see [14] for a simple sequential derivation). As in the previous section, we fix $k = \log n \log\log n$. As promised in (iii) in the last section, we will prove:

Theorem 8 *There is an atomic priority queue with up to n $(w/(\log n \log\log n)$-bit keys, supporting* `find-min` *in constant time, and supporting* `insert`, *and* `delete` *in constant amortized time.*

Our construction has some similarities to the one used in [14, §2]. The keys will be stored in sorted lists $L_0, \ldots, L_{\log n}$ where either $L_i = \emptyset$, or $2^{i-1} < |L_i| \leq 2^i$. Moreover, if L_i is non-empty, it is distributed over at most $\lceil |L_i|/k \rceil$ words.

Algorithm D `insert`(x)
D.1. $L' := (x)$.

D.2. $i := 0$.
D.3. While $L_i \neq \emptyset$ do
D.3.1. Merge L' and L_i into L'.
D.3.2. $L_i := \emptyset$.
D.3.3. $i := i + 1$.
D.4. $L_i := L'$.

In order to support `find-min` and `delete-min`, we use the atomic priority queue from [10] to keep track of which of the at most $\log n$ lists that has the smallest first element. Let L_i be the list with the smallest first element x. Then `find-min` just returns x. To implement `delete-min`, we first delete x from L_i. If thereby $|L_i| \leq 2^{i-1}$, we empty L_i, reinserting the elements one by one using `insert`.

We will now argue that we spend only constant amortized time for each key inserted. If a key is reinserted by `delete-min`, we restart the accounting, attributing the cost to `delete-min`. Consider a particular key x. The merges (step D.3.1) that x participate in must have strictly increasing values of i. In a given merging, the lists contain between $2^i + 1$ and 2^i keys. Loosing at most a factor 2, we may assume that both lists are full, containing 2^i elements. Then, if $2^i \leq k$, the per key cost of the merging is $O(i/2^i)$. If $2^i > k$, the per key cost of the merging is $O(\log k/k)$. Thus, the maximum total cost per key is

$$(\sum_{i=1}^{\log k-1} i/2^i + \sum_{\log k}^{\log n} 1) = O(1).$$

Recall that `delete-min` has constant cost if we ignore the reinsertions. To account for the reinsertions, we use a potential function $\Pi = \sum_{L_i \neq \emptyset}(2^i - |L_i|)$ measuring the number of "missing" elements in the lists. Thus Π starts at 0, and generally $0 \leq \Pi \leq 2n$. Note that `insert` does not affect Π. The immediate consequence of `delete-min` is that we delete an element from a list, hence that we increase Π by one. Now, if L_i gets emptied, $|L_i| \leq 2^{i-1}$, so this decreases Π by $\geq 2^{i-1}$. Thus the total number of reinsertions is bounded from above by the total number of `delete-min` operations, implying that the amortized cost of `delete-min` is $O(1)$. To get general `delete`, as in the last section, we just introduce a table D over keys to be deleted, checking new minimums against D. This completes the proof of Theorem 8.

References

[1] M. Ajtai, M. Fredman, and J. Komlós, Hash functions for priority queues, *Inf. Contr.* **63** (1984), 217–225.

[2] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman, Sorting in linear time? To appear at STOC'95.

[3] S. Albers and T. Hagerup, Improved parallel integer sorting without concurrent writing, *in* "Proc. 3rd SODA," pp. 463–472, 1992.

[4] A. ANDERSSON AND S. NILSSON, A new efficient radix sort , *in* "Proc. 35th FOCS," pp. 714–731, 1994.

[5] P. BEAME AND J. HÅSTAD, Optimal bounds on the decision problems on the CRCW PRAM, *J. ACM* **36** (1989), 643–670.

[6] A.M. BEN-AMRAM AND Z. GALIL, When can we sort in $o(n \log n)$ time?, *in* "Proc. 34th FOCS," pp. 538–546, 1993.

[7] J.L. CARTER AND M.N. WEGMAN, Universal classes of hash functions, *J. Comp. Syst. Sci.* **18** (1979), 143–154.

[8] M.L. FREDMAN AND R.E. TARJAN, Fibonacci heaps and their uses in improved network optimization algorithms, *J. ACM* **34** (1987) 596–615. See also FOCS'84.

[9] M.L. FREDMAN AND D.E. WILLARD, Surpassing the information theoretic bound with fusion trees, *J. Comp. Syst. Sc.* **47** (1993) 424–436. See also STOC'90.

[10] M.L. FREDMAN AND D.E. WILLARD, Trans-dichotomous algorithms for minimum spanning trees and shortest paths, *J. Comp. Syst. Sc.* **48** (1994) 533–551. See also FOCS'90.

[11] J.L. HENNESSY AND D.A. PATTERSON, *Computer Organization and Design: the Hardware/Software Interface,* Morgan Kaufmann, San Mateo, CA, 1994.

[12] P.B. MILTERSEN, Lower bounds for union-split-find related problems on random access machines, *in* "Proc. 26th STOC," pp. 625–634, 1994.

[13] P.B. MILTERSEN, N. NISAN, S. SAFRA, AND A. WIGDERSON, On data structures and asymmetric communication complexity. To appear at STOC'95.

[14] M. THORUP, An $O(\log \log n)$ priority queue, Technical Report DIKU-TR-95-5, Department of Computer Science, University of Copehagen, 1995.

[15] J.W.J. WILLIAMS, Heapsort, *Commun. ACM* **7**, 5 (1964), 347–348.