Heap-on-Top Priority Queues

Boris V. Cherkassky *

Central Economics and Mathematics Institute
Krasikova St. 32
117418, Moscow, Russia
cher@cemi.msk.su

Andrew V. Goldberg

NEC Research Institute
4 Independence Way
Princeton, NJ 08540
avg@research.nj.nec.com

March 1996

Abstract

We introduce the heap-on-top (hot) priority queue data structure that combines the multi-level bucket data structure of Denardo and Fox [9] and a heap. We use the new data structure to obtain an $O(m+n(\log C)^{\frac{1}{3}+\epsilon})$ expected time implementation of Dijkstra's shortest path algorithm [11], improving the previous bounds.

*This work was done while the author was visiting NEC Research Institute.

1 Introduction

A priority queue is a data structure that maintains a set of elements and supports operations `insert`, `decrease-key`, and `extract-min`. Priority queues are fundamental data structures with many applications. Typical applications include graph algorithms (*e.g.* [12]) and event simulation (*e.g.* [5]).

An important subclass of priority queues, used in applications such as event simulation and in Dijkstra's shortest path algorithm [11], are monotone priority queues. A priority queue is *monotone* if keys of elements on the queue are at least as big as the key of the latest element extracted from the queue. In this paper we deal with monotone priority queues.

Unless mentioned otherwise, we refer to priority queues whose operation time bounds depend only on the number of elements on the queue as *heaps*. The fastest implementations of heaps are described in [4, 12, 15]. Alternative implementations of priority queues use buckets (*e.g.* [2, 6, 9, 10]). Operation times for bucket-based implementations depend on the maximum event duration C, defined in Section 2. See [3] for a related data structure.

Heaps are more efficient when the number of elements on the heap is small. Bucket-based priority queues are more efficient when the maximum event duration C is small. Furthermore, certain work of bucket-based implementations can be amortized over elements in the buckets, yielding better bounds if the number of elements is big. In this sense, heaps and buckets complement each other.

We introduce *heap-on-top priority queues (hot queues)*, which combine the multi-level bucket data structure of Denardo and Fox [9] and a heap. The resulting implementation takes advantage of the best performance features of both data structures. We also give an alternative description of the multi-level bucket data structure which is more insightful.

Hot queues are related to, but simpler and more flexible than, the radix heaps[1] of Ahuja et. al. [2]. One advantage of our queues is that they can use any implementation of a heap. In contrast, radix heaps require a heap with operation time that depends on the number of *distinct* keys on the heap. The most complicated part of [2] is a modification of the Fibonacci heap data structure [12] that meets this requirement. Hot queues can use Fibonacci heaps with no modifications while achieving the same bounds as radix heaps.

Using the heap of Thorup [15], we obtain better bounds. As a side-effect, we obtain an $O(m+$

[1]Radix heaps use both buckets and heaps, and their operation time bounds depend on C.

$n(\log C)^{\frac{1}{3}+\epsilon})$ expected time implementation of Dijkstra's shortest path algorithm, improving the previous bounds. Since Thorup's bounds depend on the total number of elements on the heap, radix heaps cannot take immediate advantage of this data structure.

Hot queues appear to be more practical than radix heaps. Our queues are simpler. Furthermore, experience with multi-level bucket implementations [6, 13] suggests that practical implementations of hot queues should have a small number of levels (*e.g.* 3). Such implementations have no similar radix heap variants.

This paper is organized as follows. Section 2 introduces basic definitions. Our description of the multi-level bucket data structure appears in Section 3. Section 4 describes hot queues and their application to Dijkstra's algorithm. Section 5 contains concluding remarks.

2 Preliminaries

A *priority queue* is a data structure that maintains a set of elements and supports operations `insert`, `decrease-key`, and `extract-min`. We assume that elements have *keys* used to compare the elements, and denote the key of an element u by $\rho(u)$. Unless mentioned otherwise, we assume that the keys are integral. By the value of an element we mean the key of the element. The `insert` operation adds a new element to the queue. The `decrease-key` operation assigns a smaller value to the key of an element already on the queue. The `extract-min` operation removes a minimum element from the queue and returns the element. We denote the number of `insert` operations in a sequence of priority queue operations by N.

Let u be the latest element extracted from the queue. An *event* is an `insert` or a `decrease-key` operation on the queue. Given an event, let v be the element inserted into the queue or the element whose key was decreased. The *event duration* is $\rho(v) - \rho(u)$. We denote the maximum event duration by C. An application is *monotone* if all event durations are nonnegative. A *monotone* priority queue is a priority queue for monotone applications. To make these definitions valid for the first insertion, we assume that during initialization, a special element is inserted into the queue and deleted immediately afterwards. Without loss of generality, we assume that the value of this element is zero. (If it is not, we can subtract this value from all element values.)

In this paper, by *heap* we mean a priority queue whose operation time bounds are functions of the number of elements on the queue. We assume that heaps also support the `find-min` operation that returns the minimum element on the heap.

We call a sequence of operations on a priority queue *preemptive* if the sequence starts and ends with an empty queue. In particular, implementations of Dijkstra's shortest path algorithm produce preemptive operation sequences.

In this paper we use the RAM model of computation [1]. The only non-obvious result about the model we use appears in [7], where it is attributed to B. Schieber. The result is that given two machine words, we can find, in constant time, the index of the most significant bit in which the two words differ.

3 Multi-Level Buckets

In this section we describe the k-level bucket data structure of Denardo and Fox [9]. We give a simpler description of this data structure by treating the element keys as base-Δ numbers for a certain parameter Δ. A k-level bucket structure B contains k levels of buckets. Except for the top level, a level contains an array of Δ buckets. The top level contains infinitely many buckets. Each top level bucket corresponds to an interval $[i\Delta^k, (i+1)\Delta^k - 1]$. We chose Δ so that at most Δ buckets at the top level can be nonempty; we need to maintain only these buckets.[2]

We denote bucket j at level i by B^i_j. A bucket contains a set of elements in a way that allows constant-time additions and deletions, *e.g.* in a doubly linked list.

Given k, we chose Δ as small as possible subject to two constraints. First, each top level bucket must contain at least $(C+1)/\Delta$ integers. Then by definition of C, keys of elements in B belong to at most Δ top level buckets. Second, Δ must be a power of two so that we can manipulate base-Δ numbers efficiently using RAM operations on words of bits. We set Δ to the smallest power of two greater or equal to $(C+1)^{1/k}$.

We maintain μ, the key of the latest element extracted from the queue. Consider the base-Δ representation of the keys and an element u in B. By definitions of C and Δ, μ and the k least significant digits of the base-Δ representation of $\rho(u)$ uniquely determine $\rho(u)$. If $\overline{\mu}$ and $\overline{\rho}$ are the numbers represented by the k least significant digits of μ and $\rho(u)$, respectively, then $\rho(u) = \mu + (\overline{\rho} - (\overline{\mu}))$ if $\overline{\rho} \geq \overline{\mu}$ and $\rho(u) = \mu + \Delta^k + (\overline{\rho} - \overline{\mu})$ otherwise. For $1 \leq i < k$, we denote by μ_i the i-th least significant digit of the base-Δ representation of μ. We denote the number obtained by deleting the $k-1$ least significant digits of μ by μ_k. Similarly, for $1 \leq i < k$, we denote the i-th least significant digits of $\rho(u)$ by u_i and we denote the number obtained by deleting $k-1$ least significant digits of $\rho(u)$ by u_k.

[2]The simplest way to implement the top level is to "wrap around" modulo Δ.

The levels of B are numbered from k (top) to 1 (bottom) and the buckets at each level are numbered from 0 to $\Delta - 1$. Let i be the index of the most significant digit in which $\rho(u)$ and μ differ or 1 if $\rho(u) = \mu$. Given μ and u with $\rho(u) \geq \mu$, we say that the *position of u with respect to μ* is (i, u_i). If u is inserted into B, it is inserted into $B^i_{u_i}$. For each element in B, we store its position. If an element u is in B^i_j, then all except for $i+1$ most significant digits of $\rho(u)$ are equal to the corresponding digits of μ and $u_i = j$.

The following lemma follows from the fact that keys of all elements on the queue are at least μ.

Lemma 3.1 *For every level i, buckets B^i_j for $0 \leq j < \mu_i$ are empty.*

At each level i, we maintain the number d_i of elements at this level. We also maintain the total number of elements in B.

The `extract-min` operation can change the value of μ. As a side-effect, positions of some elements in B may change. Suppose that a minimum element is deleted and the value of μ changes. Let μ' be the value of μ before the deletion and let μ'' be the value of μ after the deletion. By definition, keys of the elements on the queue after the deletion are at least μ''. Let i be the position of the least significant digit in which μ' and μ'' differ. If $i = 1$ (μ' and μ'' differ only in the last digit), then for any element in B after the deletion its position is the same as before the deletion. If $i > 1$, than the elements in bucket $B^i_{\mu''_i}$ with respect to μ' are exactly those whose position is different with respect to μ''. These elements have a longer prefix in common with μ'' than with μ' and therefore they belong to a lower level with respect to μ''.

The *bucket expansion* procedure moves these elements to their new positions. The procedure removes the elements from $B^i_{\mu''_i}$ and puts them into their positions with respect to μ''. The two key properties of bucket expansions are as follows:

- After the expansion of $B_{\mu''_i}$, all elements of B are in correct positions with respect to μ''.
- Every element of B moved by the expansion is moved to a lower level.

Now we are ready to describe the multi-level bucket implementation of the priority queue operations.

- `insert`

  To insert an element u, compute its position (i, j), insert u into B^i_j, and increment d_i.

- `decrease-key`

  Decrease the key of an element u in position (i, j) as follows. Remove u from B^i_j and decrement d_i. Set $\rho(u)$ to the new value and insert u as described above.

- `extract-min`

  (We need to find and delete the minimum element, update μ, and move elements affected by the change of μ.)

  Find the lowest nonempty level i.

  Find the first nonempty bucket at level i.

  If $i = 1$, delete an element from B^i_j, set $\mu = \rho(u)$, and return u. (In this case old and new values of μ differ in at most the last digit and all element positions remain the same.)

  If $i > 1$, examine all elements of B^i_j and delete a minimum element u from B^i_j. Set $\mu = \rho(u)$ and expand B^i_j. Return u.

Next we deal with efficiency issues.

Lemma 3.2 *Given μ and u, we can compute the position of u with respect to μ in constant time.*

Proof. By definition, $\Delta = 2^\delta$ for some integer δ, so each digit in the base-Δ representation of $\rho(u)$ corresponds to δ bits in the binary representation. It is straightforward to see that if we use appropriate masks and the fact that the index of the first bit in which two words differ can be computed in constant time [8], we can compute the position in constant time. ■

Iterating through the levels, we can find the lowest nonempty level in $O(k)$ time. Using binary search, we can find the level in $O(\log k)$ time. We can do even better using the power of the RAM model:

Lemma 3.3 *If $k \le \log C$, then the lowest nonempty level of B can be found in $O(1)$ time.*

Proof. Define D to be a k-bit number with $D_i = 1$ if and only if level i is nonempty. If $k \le \log C$, D fits into one RAM word, and we can set a bit of D and find the index of the first nonzero bit of D in constant time. ■

As we will see, the best bounds are achieved for $k \le \log C$.

A simple way of finding the first nonempty bucket at level i is to go through the buckets. This takes $O(\Delta)$ time.

Lemma 3.4 *We can find the first nonempty bucket at a level in $O(\Delta)$ time.*

Remark. One can do better [9]. Divide buckets at every level into groups of size $\lceil \log C \rceil$, each group containing consecutive buckets. For each group, maintain a $\lceil \log C \rceil$-bit number with bit j equal to 1 if and only if the j-th bucket in the group is not empty. We can find the first nonempty group in $O\left(\frac{\Delta}{\log C}\right)$ time and the first nonempty bucket in the group in $O(1)$ time. This construction gives a $\log C$ factor improvement for the bound of Lemma 3.4. By iterating this construction p times, we get an $O\left(p + \frac{\Delta}{\log^p C}\right)$ bound.

Although the above observation improves the multi-level bucket operation time bounds for small values of k, the bounds for the optimal value of k do not improve. To simplify the presentation, we use Lemma 3.4, rather than its improved version, in the rest of the paper.

Theorem 3.5 *Amortized bounds for the multi-level bucket implementation of priority queue operations are as follows: $O(k)$ for* `insert`*, $O(1)$ for* `decrease-key`*, $O(C^{1/k})$ for* `extract-min`*.*

Proof. The `insert` operation takes $O(1)$ worst-case time. We assign it an amortized cost of k because we charge moves of elements to a lower level to the insertions of the elements.

The `decrease-key` operation takes $O(1)$ worst case time and we assign it an amortized cost of $O(1)$.

For the `extract-min` operation, we show that its worst-case cost is $O(k + C^{1/k})$ plus the cost of bucket expansions. The cost of a bucket expansion is proportional to the number of elements in the bucket. This cost can be amortized over the `insert` operations, because, except for the minimum element, each element examined during a bucket expansion is moved to a lower level.

Excluding bucket expansions, the time of the operation is $O(1)$ plus the $O(\Delta)$ for finding the first nonempty bucket. This completes the proof since $\Delta = O(C^{1/k})$. ■

Note that in any sequence of operations the number of `insert` operations is at least the number of `extract-min` operations. In a preemptive sequence, the two numbers are equal, and we can modify the above proof to obtain the following result.

Theorem 3.6 *For a preemptive sequence, amortized bounds for the multi-level bucket implementation of priority queue operations are as follows: $O(1)$ for* `insert`*, $O(1)$ for* `decrease-key`*, $O(k + C^{1/k})$ for* `extract-min`*.*

Remark. We can easily obtain an $O(1)$ implementation of the `delete` operation for multi-level buckets. Given a pointer to an element, we delete it by removing the element from the list of elements in its bucket.

For $k = 1$, the `extract-min` bound is $O(C)$. For $k = 2$, the bound is $O(\sqrt{C})$. The best bound of $O\left(\frac{\log C}{\log\log C}\right)$ is obtained for $k = \lceil\frac{\log C}{\log\log C}\rceil$.

Remark. The k-level bucket data structure uses $\Theta(kC^{1/k})$ space.

4 Hot Queues

A *k-level hot queue* uses a heap H and the k-level bucket structure B. Each element of the queue is either in H or in B. Once an element is inserted into H, the element stays in H until removed by the `extract-min` operation. An element in B can move to H during a `decrease-key` or an `extract-min` operation. We maintain the invariant that each level of B accounts for at most t elements currently on H. Therefore H always contains at most kt elements.

To enforce this invariant, at each level i we maintain a counter c_i. Before inserting an element u into its new position (i, j) in B, we examine c_i. If $c_i < t$, we insert the element into H and say that *u is accounted for by level i*. We also define $a(u) = i$ and store $a(u)$ with u. If $c_i \geq t$, we insert u into B^i_j.

Counters c_i are maintained as follows. Initially the counters are zero. When u is inserted into H, we increment $c_{a(u)}$. When u is extracted from H, we decrement $c_{a(u)}$. Obviously, this implementation maintains the invariant that H contains at most kt elements. The counters c_i are always maintained in this way, and we omit an explicit description of updating the counters in our description of the hot queue operations.

The `extract-min` operation works similarly to the multi-level bucket case, except the minimum can be in B or in H. One way deal with this is to insert into B a dummy element with the key equal to the key of the minimum element of H and then apply `extract-min` to B, obtaining v. We proceed depending on whether v is the dummy element or not.

The detailed description of the queue operations is as follows.

- `insert`

  To insert an element u, compute its position (i, j). If $c_i < t$, insert u into H. Otherwise insert u into B^i_j and increment d_i.

- `decrease-key`

Decrease the key of an element u as follows. If u is in H, decrease the key of u in H. Otherwise, let (i, j) be the position of u in B. Remove v from B_j^i and decrement d_i. Then insert u as described above.

- `extract-min`

  If one of H or B is empty, extract a minimum element from the other one and return it.

  Otherwise, find a minimum element u of H. For the dummy element z, set $\rho(z) = \rho(u)$ and insert z into B. Then extract the minimum element v from B.

  If $v = z$, extract a minimum element from H and return it. Otherwise remove z from B. Return v.

Note that the above implementation of `extract-min` uses Lemma 3.1 to avoid examining buckets B_j^i for $0 \leq i < \mu_i$. This result can also be used in the multi-level bucket implementation, but there it is not required to get the desired bounds.

Lemma 4.1 *The amortized cost of checking if H contains the minimum element is $O(\Delta/t)$.*

Proof. We scan at most one nonempty bucket during a check. We scan an empty bucket at level i at most once during the period of time while the prefix of μ including all except the last $i-1$ digits remains the same. Furthermore, we scan the buckets only when level i is nonempty. This can happen only if the level accounted for t insertions of elements into H during the period the prefix of μ does not change. We charge bucket scans to insertions of these elements into H and get the $O(\Delta/t)$ bound. ■

Theorem 4.2 *Let $I(N)$, $D(N)$, $F(N)$, and $X(N)$ be the time bounds for heap* `insert`*,* `decrease-key`*,* `find-min`*, and* `extract-min` *operations. Then amortized times for the hot queue operations are as follows: $O(k + I(kt))$ for* `insert`*, $O(D(kt) + I(kt))$ for* `decrease-key`*, and $O(F(kt) + X(kt) + \frac{C^{1/k}}{t})$ for* `extract-min`*.*

Proof. Two key facts are crucial for the analysis. The first fact is that the number of elements on H never exceeds kt since each level accounts for at most t elements. The second fact is Lemma 4.1. Given the first fact and Theorem 3.5, the bounds are straightforward. ■

For Fibonacci heaps [12], the amortized time bounds are $I(N) = D(N) = F(N) = O(1)$, and $X(N) = O(\log N)$. This gives $O(k)$, $O(1)$, and $O(\log(kt) + \frac{C^{1/k}}{t})$ amortized bounds for the

queue operations `insert`,
`decrease-key`, and `extract-min`, respectively. Setting $k = \sqrt{\log C}$ and $t = 2^k$, we get $O(\sqrt{\log C})$, $O(1)$, and $O(\sqrt{\log C})$ amortized bounds. Radix heaps achieve the same bounds but are more complicated.

For Thorup's heaps [15], the expected amortized time bounds are $I(N) = D(N) = F(N) = O(1)$, and $X(N) = O(\log^{\frac{1}{2}+\epsilon} N)$. This gives $O(k)$, $O(1)$, and $O(\log^{\frac{1}{2}+\epsilon}(kt) + \frac{C^{1/k}}{t})$ expected amortized time bounds for the queue operations `insert`, `decrease-key`, and `extract-min`, respectively. Here ϵ is any positive constant. Setting $k = \log^{\frac{1}{3}} C$ and $t = 2^k$, we get $O(\log^{\frac{1}{3}} C)$, $O(1)$, and $O(\log^{\frac{1}{3}+\epsilon} C)$ expected amortized time.

Similarly to Theorem 3.6, we can get bounds for a preemptive sequence of operations.

Theorem 4.3 *Let $I(N)$, $D(N)$, $F(N)$, and $X(N)$ be the time bounds for heap* `insert`, `decrease-key`, `find-min`, *and* `extract-min` *operations and a consider a preemptive sequence of the hot queue operations. The amortized bounds for the operations are as follows: $O(I(kt))$ for* `insert`, *$O(D(kt) + I(kt))$ for* `decrease-key`, *and $O(k + F(kt) + X(kt) + \frac{C^{1/k}}{t})$ for* `extract-min`.

Using Fibonacci heaps, we get $O(1)$, $O(1)$, and $O(k + \log(kt) + \frac{C^{1/k}}{t})$ amortized bounds for the queue operations. Consider `extract-min`, the only operation with non-constant bound. Setting $k = 1$ and $t = \frac{C}{\log C}$, we get an $O(\log C)$ bound. Setting $k = 2$ and $t = \frac{\sqrt{C}}{\log C}$, we get an $O(\log C)$ bound. Setting $k = \sqrt{\log C}$ and $t = 2^k$, we get an $O(\sqrt{\log C})$ bound.

Remark. All bounds are valid only when $kt \leq n$. For $kt > n$, one should use a heap.

Remark. Consider the 1- and 2-level implementations. Although the time bounds are the same, the two-level implementation has two advantages: It uses less space and its time bounds remain valid for a wider range of values of C.

Using Thorup's heaps and setting $k = \log^{\frac{1}{3}} C$ and $t = 2^k$, we get $O(1)$, $O(1)$, and $O(\log^{\frac{1}{3}+\epsilon} C)$ expected amortized time bounds.

The above time bounds allow us to get an improved bound on Dijkstra's shortest path algorithm. Suppose we are given a graph with n vertices, m arcs, and integral arc lengths in the range $[0, C]$. The running time of Dijkstra's algorithm is dominated by a preemptive sequence of priority queue operations that includes $O(n)$ `insert` and `extract-min` operations and $O(m)$ `decrease-key` operations (see *e.g.* [14]). The maximum event duration for this sequence of operations is C. The bounds for the queue operations immediately imply the following result.

Theorem 4.4 *On a network with N vertices, m arcs, and integral lengths in the range* $[0, C]$, *the shortest path problem can be solved in* $O(m + n(\log C)^{\frac{1}{3}+\epsilon})$ *expected time.*

This improves the deterministic bound of $O(m + n\sqrt{\log C})$ that uses radix heaps [2]. (The hot queue implementation based on Fibonacci heaps matches this deterministic bound.)

5 Concluding Remarks

The hot queue data structure combines the best features of heaps and multi-level buckets in a natural way. If C is very small compared to n, the data structure performs essentially as the multi-level bucket structure. If C is very large, the data structure performs as the heap used in it. For intermediate values of C, the data structure performs better than either the heap or the multi-level buckets structure.

Hot queues can be adapted to nonmonotone applications in several ways. The simplest way is as follows. We define *normal* and *exceptional* elements as follows. The element inserted and extracted during the initialization is normal; μ is the key value of the last normal element extracted from the queue. An element on the queue is normal while its key value is greater or equal to μ and exceptional otherwise. Normal elements are handled by the standard hot queue operations. Exceptional elements are always added to the heap and processed using the heap operations. In a near-monotone application with very few exceptional elements, this version of the queue may perform better than a heap.

Acknowledgments

We would like to thank Bob Tarjan for stimulating discussions and insightful comments.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] R. K. Ahuja, K. Mehlhorn, J. B. Orlin, and R. E. Tarjan. Faster Algorithms for the Shortest Path Problem. *J. Assoc. Comput. Mach.*, 37(2):213–223, April 1990.

[3] P. Van Emde Boas, R. Kaas, and Zijlstra. Design and Implementation of Efficient Priority Queue. *Math. Systems Theory*, 10:99–127, 1977.

[4] G. S. Brodal. Worst-Case Efficient Priority Queues. In *Proc. 7th ACM-SIAM Symposium on Discrete Algorithms*, pages 52–58, 1996.

[5] R. Brown. Calandar Queues: A Fast $O(1)$ Priority Queue Implementation for the Simulation Event Set Problem. *Comm. ACM*, 31:1220–1227, 1988.

[6] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest Paths Algorithms: Theory and Experimental Evaluation. In *Proc. 5th ACM-SIAM Symposium on Discrete Algorithms*, pages 516–525, 1994. To appear in Math. Prog.

[7] R. Cole and U. Vishkin. Deterministic Coin Tossing and Accelerating Cascades: Micro and Macro Techniques for Designing Parallel Algorithms. In *Proc. 18th Annual ACM Symposium on Theory of Computing*, pages 206–219, 1986.

[8] R. Cole and U. Vishkin. Deterministic Coin Tossing with Applications to Optimal Parallel List Ranking. *Information and Control*, 70:32–53, 1986.

[9] E. V. Denardo and B. L. Fox. Shortest–Route Methods: 1. Reaching, Pruning, and Buckets. *Oper. Res.*, 27:161–186, 1979.

[10] R. B. Dial. Algorithm 360: Shortest Path Forest with Topological Ordering. *Comm. ACM*, 12:632–633, 1969.

[11] E. W. Dijkstra. A Note on Two Problems in Connection with Graphs. *Numer. Math.*, 1:269–271, 1959.

[12] M. L. Fredman and R. E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *J. Assoc. Comput. Mach.*, 34:596–615, 1987.

[13] A. V. Goldberg and C. Silverstein. Implementations of Dijkstra's Algorithm Based on Multi-Level Buckets. Technical Report 95–187, NEC Research Institute, Princeton, NJ, 1995.

[14] R. E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[15] M. Thorup. On RAM Priority Queues. In *Proc. 7th ACM-SIAM Symposium on Discrete Algorithms*, pages 59–67, 1996.