

# Rank-Sensitive Priority Queues

Brian C. Dean and Zachary H. Jones

School of Computing, Clemson University  
Clemson, SC, USA.

{bcdean,zjones}@cs.clemson.edu

**Abstract.** We introduce the rank-sensitive priority queue — a data structure that always knows the minimum element it contains, for which insertion and deletion take  $O(\log(n/r))$  time, with  $n$  being the number of elements in the structure, and  $r$  being the rank of the element being inserted or deleted ( $r = 1$  for the minimum,  $r = n$  for the maximum). We show how several elegant implementations of rank-sensitive priority queues can be obtained by applying novel modifications to treaps and amortized balanced binary search trees, and we show that in the comparison model, the bounds above are essentially the best possible. Finally, we conclude with a case study on the use of rank-sensitive priority queues for shortest path computation.

## 1 Introduction

Let us say that a data structure is *min-aware* if it always knows the minimum element it contains; equivalently, the structure should support an  $O(1)$  time *find-min* operation. Furthermore, we say a data structure is *dynamic* if it supports an *insert* operation for adding new elements, and a *delete* operation for removing elements (as is typical, we assume *delete* takes a pointer directly to the element being deleted, since our data structure may not support an efficient means of finding elements). The functionality of a dynamic min-aware structure captures the essence of the priority queue, the class of data structures to which the results in this paper are primarily applicable. In a priority queue, the *find-min* and *delete* operations are typically combined into an aggregate *delete-min* operation, but we will find it convenient to keep them separate and focus on the generic framework of a dynamic min-aware structure in the ensuing discussion.

In the comparison model of computation, it is obvious that either *insert* or *delete* must run in  $\Omega(\log n)$  worst-case time for any dynamic min-aware structure, since otherwise we could circumvent the well-known  $\Omega(n \log n)$  worst-case lower bound on comparison-based sorting by inserting  $n$  elements and repeatedly deleting the minimum. As a consequence, even the most sophisticated comparison-based priority queues typically advertise a running time bound of  $O(\log n)$  for *insert* or (more commonly) *delete-min*. The sorting reduction above makes it clear that this is the best one can hope to achieve to inserting or deleting the *minimum* element in our structure, but what about other elements? Might it be possible, say, to design a comparison-based dynamic min-aware data structure

with *insert* and *delete* running in only  $O(1)$  time, except for the special case where we insert or delete a new minimum element, which takes  $O(\log n)$  time? At first glance, these requirements no longer seem to run afoul of the sorting lower bound, but they still seem quite ambitious. This motivates the general question: if we take the *rank* of the element being inserted or deleted into consideration, how quickly can a dynamic min-aware structure support *insert* and *delete*?

In this paper, we answer the question above by providing several implementations of what we call *rank-sensitive* priority queues. These are dynamic min-aware data structures capable of performing *insert* and *delete* in  $O(\log(n/r))$  time (possibly amortized or in expectation), where  $n$  denotes the number of elements in the structure and  $r$  denotes the rank of the element being inserted or deleted ( $r = 1$  for the minimum,  $r = n$  for the maximum<sup>1</sup>). Note that the structure is not explicitly told the ranks of the elements being inserted or deleted; rather, its performance simply scales in a graceful manner from  $O(\log n)$  for inserting or deleting near the minimum down to  $O(1)$  for, say, modifying any of the 99% of the largest elements. The resulting structure should therefore be ideally suited for the case where we want to maintain a dynamic collection of elements for which we only occasionally (say, in case of emergency) need priority queue functionality. Our various implementations of rank-sensitive priority queues will appeal to the serious data structure aficionado in that they involve elegant new twists on well-studied data structures, notably treaps, amortized balanced binary search trees, and radix heaps.

After discussing our data structures, we then give a proof that in the comparison model,  $O(\log(n/r))$  is essentially the best one can hope to achieve for a rank-sensitive running time bound. There are two main challenges in doing this, the first being that it is actually not so easy to state such a lower bound theorem the right way; for example, if we are not careful, we can end up with a theorem that is vacuously true since it must hold for the special case where  $r = 1$ . The second main challenge is to perform a reduction that somehow manages to use our data structure to sort by removing mostly non-minimal elements.

Much of the research driving the development of fast priority queues is ultimately focused on speeding up Dijkstra’s shortest path algorithm. Rank-sensitive priority queues are also worth studying in this context. Since the dominant component of the running time for Dijkstra’s algorithm is typically the large number of *decrease-key* operations (in our case, implemented by *delete* followed by *insert* with a new key), we expect a rank-sensitive priority queue to perform well as long as many of our *decrease-key* operations don’t move elements too close to the minimum in rank (a potentially reasonable assumption, for many types of shortest path problems). One might therefore hope that rank-sensitive priority queues might give us performance bounds in practice that match those of more

---

<sup>1</sup> We assume for simplicity that all elements in our structure have distinct values, so ranks are uniquely-defined. In the event of ties, we would need to define  $r$  to be the maximum rank of all tied elements, in order to ensure that our upper bounds still hold in the worst case.

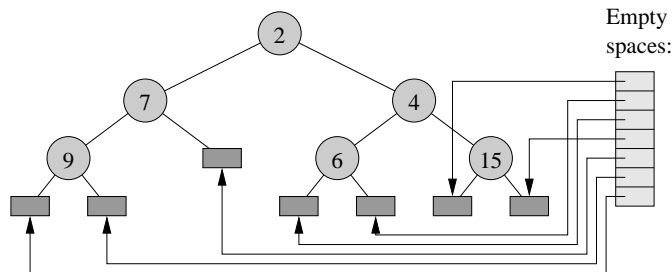
complicated data structures, such as Fibonacci heaps. The last section of our paper investigates this possibility with a discussion of computational results of using rank-sensitive priority queues in shortest path computation.

Several works related to ours appear in the literature. Splay trees [12] and the unified structure [3] satisfy the static and dynamic finger theorems (see [6, 5]), which (if we use the minimum element as a finger) give us amortized bounds of  $O(\log r)$  for insertion or deletion of a rank- $r$  element. This is indeed a form of rank sensitivity, but it is significantly weaker than our  $O(\log(n/r))$  bound above:  $O(\log r)$  evaluates to  $O(\log n)$  for nearly all the elements in a structure, while  $O(\log(n/r))$  evaluates to  $O(1)$  for nearly all the elements. A similar bound is obtained by the *Fishspears* priority queue of Fisher and Patterson [8], where the running time for inserting and deleting element  $x$  is bounded by  $O(\log m(x))$ , with  $m(x)$  giving the maximum rank of  $x$  over its lifetime in the structure. Iacono’s *queap* data structures [11] support *delete-min*( $x$ ) operation in time  $O(\log q(x))$ , where  $q(x)$  gives the number of elements in the structure that are older than  $x$ ; this can also be considered a form of “rank sensitivity”, where “rank” now has the very different meaning of “seniority” in the structure. Note that the working set property of splay trees and the unified structure leads to a symmetric bound:  $O(\log y(x))$ , where  $y(x)$  denotes the number of elements in the structure younger than  $x$ .

## 2 A Randomized Approach

The first idea that comes to mind when trying to build a rank-sensitive priority queue is perhaps whether or not a standard binary heap might be sufficient. Insertion and deletion of an element  $x$  in a binary heap can be easily implemented in time proportional to the height of  $x$ , which is certainly  $O(\log n)$  for the minimum element and  $O(1)$  for most of the high-rank elements in the structure. However, if the maximum element in the left subtree of the root is smaller than the minimum element in the right subtree of the root, it is possible we could end up with, say, the median element being the right child of the root, for which deletion will take  $O(\log n)$  time instead of the  $O(1)$  time required by a rank-sensitive structure.

Randomization gives us a nice way to fix the problem above, giving a simple and elegant implementation of a rank-sensitive priority queue in which *insert* and *delete* run in  $O(\log(n/r))$  expected time. Let us store our elements in a heap-ordered binary tree (not necessarily balanced), where we maintain an unordered array of pointers to the empty “NULL” spaces at the bottom of the tree, as shown in Figure 1. To insert a new element into an  $(n-1)$ -element tree, we place it into one of the  $n$  empty spaces at the bottom of the tree, chosen uniformly at random in  $O(1)$  time, and then we sift it up (by repeatedly rotating with its parent) until the heap property is restored. To delete an element, we set its value to  $+\infty$  and sift it down (by repeatedly rotating with its smallest child) until it becomes a leaf, after which it is removed.



**Fig. 1.** The  $h$ -treap: a heap-ordered  $n$ -element binary tree augmented with an unordered array of pointers to the  $n + 1$  empty “NULL” spaces at the bottom of the tree.

The structure above is closely related to a *treap* [2], a hybrid between a binary search tree (BST) and heap in which every node in a binary tree stores two keys: a “BST” key, and a heap key. The structure satisfies the BST property with respect to the BST keys, and the heap property with respect to the heap keys. The primary application of treaps is to provide a simple balancing mechanism for BSTs — if we store our actual elements in the BST keys, and choose the heap keys randomly, then this forces the shape of the treap to behave probabilistically as if we had built a BST from our elements by inserting them in random order (and it is well known that such randomly-built BSTs are balanced with high probability). In our case, we are using the treap in a new “symmetric” way that does not seem to appear in the literature to date: we are storing our actual elements within the heap keys, and we are effectively assigning the BST keys randomly. However, rather than storing explicit numeric BST keys in the nodes of our tree, these random BST keys are implicit in the sequence encoded by the inorder traversal of our tree. Each time we insert a new element, we are effectively assigning it a random “BST key” since by inserting it into a randomly-chosen empty space at the bottom of the tree, we are inserting it into a randomly-chosen location within the inorder traversal sequence encoded by the tree. For lack of a better name, let us call such a structure an  $h$ -treap (since the actual elements are stored in the heap part of the treap), versus a standard  $b$ -treap in which we store our elements in the BST part. Observe that the  $h$ -treap behaves like a  $b$ -treap in that its shape is probabilistically that of a randomly-built BST, so it is balanced with high probability.

**Theorem 1.** *The insert and delete operations in an  $h$ -treap run in  $O(\log(n/r))$  expected time.*

*Proof.* For any element  $x$  stored in an  $h$ -treap  $T$ , let  $s(x)$  denote the number of elements present in  $x$ ’s subtree (including  $x$ ). Since subtrees in an  $h$ -treap are balanced with high probability, it takes  $O(\log s(x))$  time both in expectation and with high probability to delete  $x$ , since the height of  $x$ ’s subtree is  $O(\log s(x))$  with high probability. Consider now the deletion of element  $x$  having rank  $r$ .

Note that  $s(x)$  is a random variable, owing to the random shape of  $T$ . If we define  $T_{r-1}$  to be the “top” part of  $T$  containing only the elements of ranks  $1 \dots r-1$ , then  $x$  will be located at one of the  $r$  empty spaces at the bottom of  $T_{r-1}$ . Since the remaining  $n-r$  elements of ranks  $r+1 \dots n$  are equally likely to appear in each of these  $r$  spaces, we have  $\mathbf{E}[s(x)] = 1 + (n-r)/r = n/r$ . Due to Jensen’s inequality, we now see that the expected time required to delete  $x$  is  $\mathbf{E}[O(\log s(x))] = O(\log \mathbf{E}[s(x)]) = O(\log(n/r))$ . The expected running time of insertion is the same due to symmetry, since the time required to insert an element of rank  $r$  is exactly the same as the time required to subsequently delete the element (the sequence of rotations performed by the deletion will be the reversal of those performed during insertion).

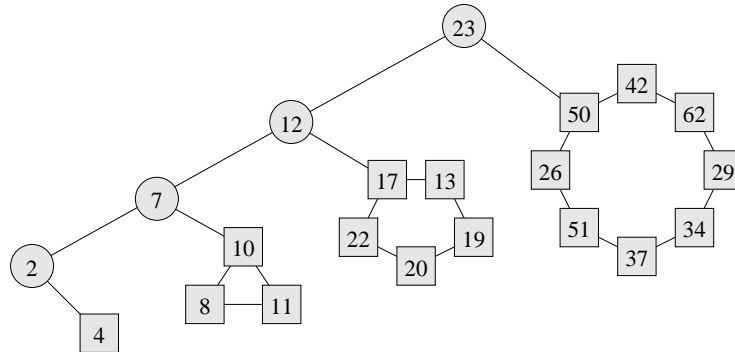
Since  $\mathbf{E}_r[\log(n/r)] = O(1)$ , we note that one can build an  $h$ -treap on  $n$  elements in  $O(n)$  expected time by inserting them sequentially in random order.

### 3 An Amortized Approach

In [7] (problem 18-3), a simple and elegant BST balancing technique of G. Varghese is described that allows for the *insert* and *delete* operations both run in  $O(\log n)$  amortized time. In this section, we build on this approach to obtain a rank-sensitive priority queue with  $O(\log(n/r))$  amortized running times for *insert* and *delete*.

Let  $s(x)$  denote the size (number of elements) of  $x$ ’s subtree in a BST. For any  $\alpha \in [1/2, 1)$ , we say  $x$  is  $\alpha$ -weight-balanced if  $s(\text{left}(x)) \leq \alpha s(x)$  and  $s(\text{right}(x)) \leq \alpha s(x)$ . A tree  $T$  is  $\alpha$ -weight-balanced if all its elements are  $\alpha$ -weight-balanced, and it is easy to show that an  $n$ -element  $\alpha$ -weight-balanced tree has maximum height  $\log_{1/\alpha} n$ . The amortized rebalancing method of Varghese selects  $\alpha \in (1/2, 1)$  and augments each element  $x$  in an  $\alpha$ -weight-balanced BST with its subtree size  $s(x)$ . Whenever an element is inserted or deleted, we examine the elements along the path from the root down to the inserted or deleted element, and if any of these are no longer  $\alpha$ -weight-balanced, we select the highest such node  $x$  in the tree and rebalance  $x$ ’s subtree in  $O(s(x))$  time so it becomes  $1/2$ -balanced. It is easy to show that *insert* and *delete* run in only  $O(\log n)$  time, because we can pay for the expensive operation of rebalancing the subtree of a non- $\alpha$ -weight-balanced element  $x$  by amortizing this across the  $\Omega(s(x))$  intervening inserts and deletes that must have occurred within  $x$ ’s subtree since the last time  $x$  was  $1/2$ -weight-balanced.

We can build an effective rank-sensitive priority by relaxing the amortized balanced BST above so that the right subtree of every element is left “unbuilt”, storing the elements of each unbuilt subtree in a circular doubly-linked list. As shown in Figure 2, only the left spine of the tree remains fully built. The only elements that effectively maintain their subtree sizes are now those on the left spine, and each one of these elements also maintains the size of its right subtree. To allow us to walk up the tree from any element, we augment every element in the circular linked list of the right subtree of element  $x$  with a pointer directly



**Fig. 2.** An  $\alpha$ -weight-balanced tree with all its right subtrees left unbuilt (whose elements are stored in circular doubly-linked lists).

to  $x$ . In order to support *find-min* in  $O(1)$  time, we maintain a pointer to the lowest element on the left spine.

Note that the mechanics of the amortized BST rebalancing mechanism above continue to function perfectly well in this relaxed setting. To insert a new element, we walk down the left spine until the BST property dictates into which right subtree it should be placed, unless the element is the new minimum, in which case it is placed at the bottom of the left spine. We then walk back up the tree, updating the subtree sizes and checking if rebalancing should occur (we discuss how to rebalance in a moment). Deletion is accomplished lazily, by marking elements as inactive, maintaining a count of the number of inactive elements in each subtree, and rebuilding any subtree  $x$  for which  $(2\alpha - 1)s(x)$  of its elements have become inactive (if multiple subtrees satisfy this condition, we rebuild the highest one); this can be done using essentially the same mechanism we use for rebuilding in the event that elements become non- $\alpha$ -weight balanced.

For rebalancing, we must take some care because the standard method of rebalancing a BST in linear time exploits the fact that we can obtain a sorted ordering of its elements via an inorder traversal. Fortunately, since our structure is relaxed, we do not need to know the full sorted ordering of our elements in order to rebalance. The following procedure will make any  $k$ -element subtree  $1/2$ -weight-balanced in  $O(k)$  time:

1. Select the median element  $m$  in the subtree in  $O(k)$  time.
2. Partition the  $k - 1$  remaining elements about  $m$  into two sets  $S_<$  and  $S_>$  such that  $|S_<|$  and  $|S_>|$  differ by at most one.
3. Make  $m$  the root of the rebalanced subtree, setting its right subtree to be the doubly-linked list containing elements in  $S_>$ .
4. Recursively build a  $1/2$ -weight-balanced subtree from  $S_<$  and set this to be  $m$ 's left subtree.

For an even simpler algorithm, we could choose  $m$  randomly instead of using a deterministic median-finding algorithm. If we do this, our running time bounds will all still hold in expectation.

**Lemma 1.** *In an  $\alpha$ -weight-balanced tree with  $n$  elements, the depth of a rank- $r$  element is at most  $\log_{1/\alpha}(n/r)$ .*

*Proof.* Let  $k = \lceil \log_{1/\alpha}(n/r) \rceil$ , and let  $x_1, x_2, \dots$  denote the elements down the left spine of an  $n$ -element  $\alpha$ -weight-balanced tree, with  $x_1$  being the root. Since  $s(x_{i+1}) \leq \alpha s(x_i)$ , we have  $s(x_{k+1}) \leq \alpha^k s(x_1) = \alpha^k n$ , so at least the  $1 + (1 - \alpha^k)n \geq 1 + (1 - (r/n))n = 1 + n - r$  largest elements live at depth at most  $k$  in our tree, and this set includes the rank- $r$  element.

**Theorem 2.** *The insert and delete operations in the structure above run in  $O(\log(n/r))$  amortized time.*

*Proof.* To amortize the cost of rebalancing properly, let  $c(x)$  denote the number of insertions and deletions in  $x$ 's subtree after the last time  $x$  was made to be  $1/2$ -weight-balanced, and let us define a potential function  $\Phi = \frac{1}{2\alpha-1} \sum_x c(x)$ . The amortized cost of an operation is given by its actual (“immediate”) cost plus any resulting change in potential. Assume for a moment that no rebalancing takes place. For *insert*, the preceding lemma tells us that the actual cost is  $O(\log(n/r))$ , and we also add up to  $\frac{1}{2\alpha-1} \log_{1/\alpha}(n/r)$  units of potential. For *delete*, the actual cost is  $O(1)$ , and again we add up to  $\frac{1}{2\alpha-1} \log_{1/\alpha}(n/r)$  new units of potential. Now consider the case where we rebalance; this can occur either if (i) some element  $x$  becomes non- $\alpha$ -weight-balanced, or (ii) if some element  $x$  is found to contain at least  $(2\alpha - 1)s(x)$  inactive elements in its subtree. For case (i) to occur, we must have  $|s(\text{left}(x)) - s(\text{right}(x))| \geq (2\alpha - 1)s(x)$ , and since  $c(x) \geq |s(\text{left}(x)) - s(\text{right}(x))|$ , we find that in both (i) and (ii), we always have  $c(x) \geq (2\alpha - 1)s(x)$  when rebalancing occurs at  $x$ , so the decrease in potential caused by setting  $c(x) = 0$  is at least  $s(x)$ , the amount of work required to rebalance. Rebalancing is therefore essentially “free” (in the amortized sense), since we can pay for it using previously-generated credit invested in our potential function.

It is worth noting the similarity between the amortized rank-sensitive priority queue above and the well-studied *radix heap* [1]. Radix heaps are RAM data structures that store integer-valued keys in a fixed known range, but their operation is quite similar to our amortized structure above — they also leave right subtrees unbuilt, and perform periodic rebalancing when the minimum value in the heap reaches a specific threshold. In fact, one might wish to think of our structure as a natural comparison-based analog of the radix heap. Another related structure worth considering is the *scapegoat tree* [9], a more sophisticated variant of the amortized balanced BST above that manages to avoid storing any augmented information while still achieving  $O(\log n)$  height at all times. Since scapegoat trees are not always  $\alpha$ -weight-balanced (they adhere to a slightly more relaxed notion of this property), it does not appear that one can easily modify them in the manner above to obtain an analogous rank-sensitive priority queue.

If we want to build a rank-sensitive priority queue with  $O(\log(n/r))$  *worst-case* performance for *insert* and *delete*, then we can do so by “de-amortizing” the structure above in a standard mechanical fashion — rebuilds are performed at an accelerated rate, several steps at a time (but still only  $O(1)$  at once), in parallel with the subsequent insertions and deletions occurring after the rebuild. We omit further details until the full version of this paper. Since the resulting data structure is rather clumsy, it remains an interesting open question whether or not there is a simple and more elegant method to obtain  $O(\log(n/r))$  worst-case bounds.

## 4 Lower Bounds

We now argue that  $O(\log(n/r))$  is essentially the best bound one can hope to achieve, in the comparison model, for deletion in a rank-sensitive priority queue. It is slightly challenging to find the “right” statement of this lower bound, however. Suppose we fix a value of  $\rho \in (0, 1]$  and consider an access sequence  $S$  of *insert* and *delete* operations in a dynamic min-aware data structure, all involving elements for which  $r/n \leq \rho$ . We would like to claim that the average cost of a *delete* operation in our access sequence must be  $\Omega(\log(1/\rho))$  in the worst case, in the comparison model (henceforth, we assume we are in the comparison model). Unfortunately, this claim is trivially true since it holds for the special case where  $S$  is the access sequence arising when we use our structure to sort  $n$  elements —  $n$  *deletions* of the rank  $r = 1$  element. Moreover, if we try to remedy this problem by considering only access sequences without any deletions at rank  $r = 1$ , then the claim above becomes false because now our data structure can now be assured that it will never need to remove the minimum element, so it can “cheat” and use less work maintaining the current minimum than it would normally need to do (e.g., it could simply maintain a pointer to the current minimum that is reset whenever a new minimum element is inserted, thereby supporting both *insert* and *delete* in  $O(1)$  time).

In order to obtain a meaningful lower bound, we therefore need to consider access sequences in which deletion of a rank-1 element is possible (just to “keep the data structure on its toes” and make it do an honest amount of work in maintaining the current minimum), but where we cannot allow so many rank-1 deletions that we aggravate the comparison-based sorting lower bound and obtain a trivial  $\Omega(\log n)$  worst-case lower bound per deletion that does not incorporate rank. Our solution is to consider access sequences of the following form:

**Definition 1.** *An access sequence  $S$  of insertions and deletions containing  $k$  deletions is  $\rho$ -graded if all  $k$  deletions  $S$  satisfy  $r/n \leq \rho$ , and if for every  $a \geq 1$ , at most  $k/a$  deletions satisfy  $r/n \leq \rho/a$ .*

For example, in a  $\frac{1}{8}$ -graded sequence, all deletions operate on elements having rank  $r \leq n/8$  (i.e., elements in the smallest  $1/8$  portion of the structure), at most half the deletions can remove elements of rank  $r \leq n/16$ , at most a quarter can

remove elements of rank  $r \leq n/32$ , and at most an  $8/n$  fraction of these deletions can involve a rank-1 element.

A key ingredient we will need soon is a *d-limit heap* — a priority queue from which we promise to call *delete-min* at most  $d$  times. It is reasonably straightforward to implement a *d-limit heap* that can process  $x$  calls to *insert* and  $y$  calls to *delete-min* in  $O(x) + y \log_2 d + o(y \log d)$  time; details will appear in the full version of this paper. Another ingredient we need is that for any dynamic min-aware structure  $M$ , we can instrument  $M$  so as to maintain the directed acyclic graph  $G(M)$  of all its comparisons to date. The DAG  $G(M)$  contains a directed edge  $(x, y)$  after  $M$  directly compares  $x$  and  $y$  and finds  $x < y$ . Elements inserted and removed from  $M$  are correspondingly inserted and removed from  $G(M)$ . If  $M$  makes  $c$  comparisons during its lifetime, then maintenance of  $G(M)$  can be achieved in only  $O(c)$  time, thereby leaving the asymptotic running time of  $M$  unchanged.

**Lemma 2.** *In any dynamic min-aware structure  $M$ , if all elements previously removed from  $M$  are smaller than those presently in  $M$ , then every non-minimal element of  $M$  must have positive in-degree in  $G(M)$ .*

*Proof.* If this were not the case, then a non-minimal element with zero in-degree would have no way of certifying its non-minimality.

The main result of this section is now the following.

**Theorem 3.** *For  $\rho \in (0, 1/2)$ , in the comparison model, any dynamic min-aware data structure (starting out empty) must spend  $\Omega(|S| + n \log(1/\rho))$  worst-case time processing a  $\rho$ -graded sequence  $S$  containing  $n$  deletions.*

*Proof.* We show how to sort a set  $E$  containing  $n$  elements by using a small number of comparisons plus the execution of a carefully-crafted  $\rho$ -graded sequence  $S$  containing  $O(n)$  deletions on a dynamic min-aware data structure  $M$ . Since the sorting problem requires at least  $n \log_2 n - o(n \log n)$  comparisons in the worst case, we will be able to show that at least  $\Omega(|S| + n \log(1/\rho))$  comparisons must come from  $M$ . The main challenge in our reduction is to sort with only a limited number of rank-1 deletions. To do this, we initially insert into  $M$  the contents of  $E$  followed by a set  $D$  of  $d = \frac{\rho n - 1}{1 - \rho} \leq \frac{\rho}{1 - \rho} n \leq n$  dummy elements that are all taken to be smaller than the elements in  $E$ . Note that the element of relative rank  $\rho$  in our structure has rank  $\rho(n + d) = d + 1$ , so it is the smallest of the elements in  $E$ . We maintain pointers to the  $d$  dummy elements, since we will occasionally delete and re-insert them. We now sort the contents of  $E$  by enacting  $n/d$  rounds, each of which involves these steps:

1. Initialize a new *d-limit heap*  $H$ .
2. For each dummy element  $e \in D$ , largest to smallest, delete and then re-insert  $e$ , taking the new values of the dummy elements to be larger than all the old values (but still smaller than the elements of  $E$ ), so we satisfy the conditions of Lemma 2. In this step and for the rest of the phase, any element  $e \in M \setminus D$  which acquires a new incoming edge in  $G(M)$  from some element of  $D$  is inserted in  $H$ .

3. Repeat the following  $d$  times: call *delete-min* in  $H$  to obtain element  $e$  (the smallest remaining element in  $E$ , thanks to Lemma 2). Then delete  $e$  from  $M$  and add all  $e$ 's immediate successors in  $G(M)$  to  $H$  (elements  $e'$  for which  $(e, e')$  is an edge). Finally, insert a large dummy element at the end of  $M$  to keep the total element count equal to  $n + d$ .
4. Destroy  $H$ .

We claim that the sequence of operations we perform in  $M$  above is  $\rho$ -graded — half the deletions occur at relative rank  $\rho$ , and the other half are spread over the range  $[0, \rho]$ . The elements we remove from  $H$  over all  $n/d$  phases give us the contents of  $E$  in sorted order, so the procedure above must indeed require  $n \log_2 n - o(n \log n)$  comparisons in the worst case. Letting  $c$  denote the total number of comparisons made by  $M$  during our procedure, we note that the total number of comparisons we make outside the operation of  $M$  is bounded by  $O(c)$  except for those made by  $H$ . If we include the comparisons made by  $H$ , we find that the total number of comparisons is  $O(c) + n \log_2 d + o(n \log d)$ . Since this must be at least  $n \log_2 n - o(n \log n)$ , we have  $c = \Omega(n \log n/d) = \Omega(n \log \frac{1-\rho}{\rho}) = \Omega(n \log(1/\rho))$ .

## 5 Case Study: Shortest Path Computation

Rank-sensitive priority queues are worth considering in conjunction with Dijkstra's shortest path algorithm since they may support a fast *decrease-key* operation in practice. To decrease the key of an element, we delete it and re-insert it with a new value, and as long as this new value gives the element a new rank sufficiently far from the minimum, we expect the entire operation to run quite fast; for many shortest path instances, we expect most of the *decrease-key* invocations to run essentially in constant time, which gives the rank-sensitive priority queue the potential for matching the performance in practice of more sophisticated priority queues, such as Fibonacci heaps.

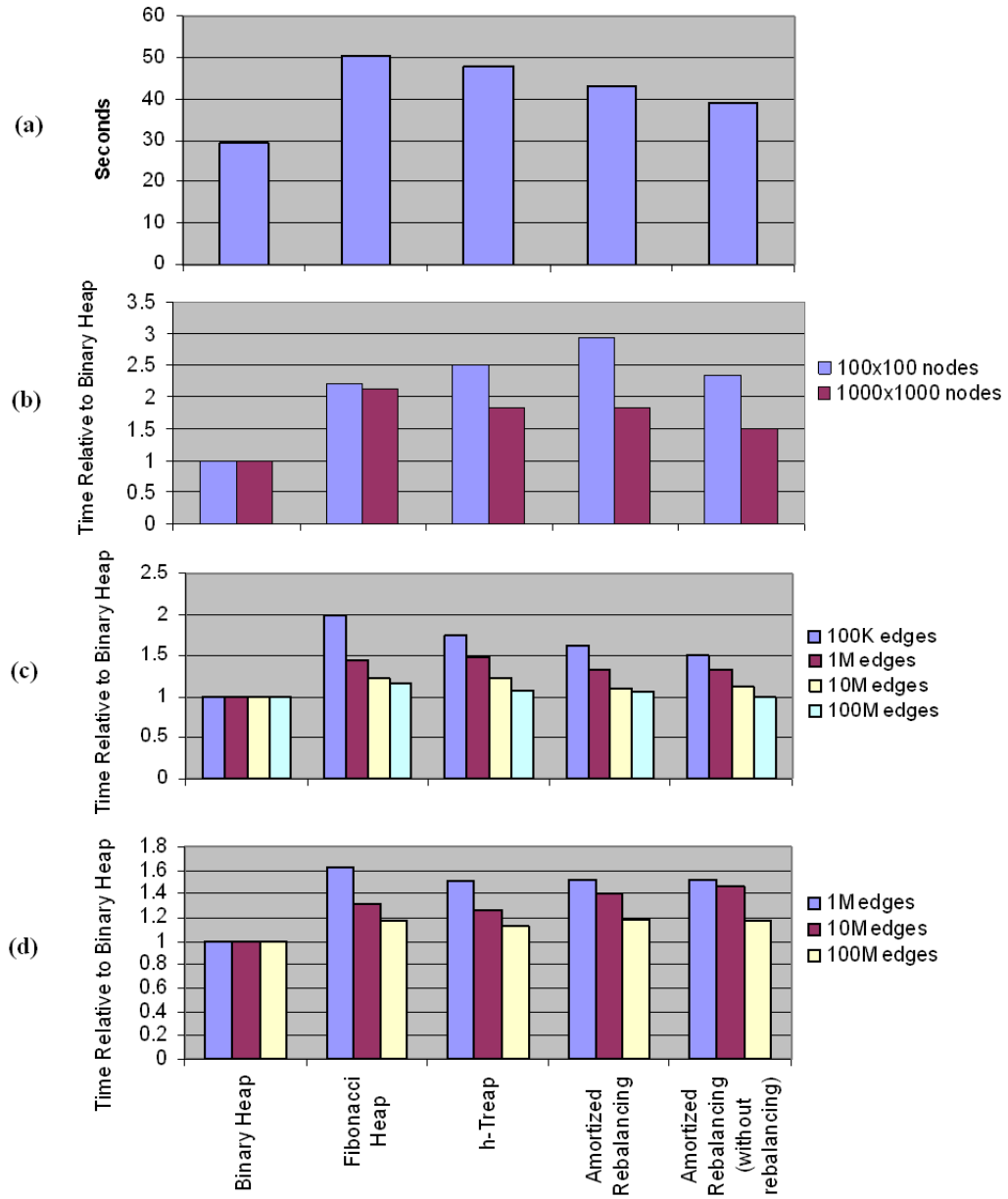
To evaluate the utility of rank-sensitive priority queues for shortest path computation, we implemented the structures outlined in Sections 2 and 3 and compared them with binary heaps and Fibonacci heaps on a variety of shortest path instances. We also tested a variant of the amortized balanced structure from Section 3 in which rebalancing was turned off, in which we only rebalanced at the lower end of the left spine in response to removal of the minimum; owing to the “well-behaved” structure of most of our inputs, this structure actually tended to perform better in practice than its counterpart with rebalancing enabled. Our implementations of the amortized balanced structures perform rebalancing of a subtree by partitioning on a randomly-chosen element, rather than by finding the median deterministically. Implementations of Dijkstra's algorithm using a binary heap and Fibonacci heap were obtained from the widely-used “splib” library [4], and inputs and random network generators were obtained from the DIMACS shortest path implementation challenge. All computational experiments were run on a 1GHz Opteron processor with 4GB memory.

Figure 3 illustrates our computational results. The graphs we tested were (a) the USA road network, with roughly 23 million nodes and 58 million edges, (b) 2d grids, (c) random graphs  $G_{n,m}$  with  $n = 10,000$  and  $m$  ranging from 100,000 up to 100 million, and (d) random Euclidean graphs (defined by  $n$  nodes embedded in the 2d Euclidean plane, with edge length equal to squared Euclidean distance) with 10,000 nodes and 1 to 100 million edges. On all of our tests, the rank-sensitive structures demonstrated performance slightly worse than a binary heap and slightly better than a Fibonacci heap, with the performance gap versus the binary heap narrowing as our graphs become more dense. The random edge lengths in our grids (b) and random graphs (c) are chosen independently from a common uniform distribution, so we should not be surprised to see the standard binary heap perform so well even on dense graphs, since Goldberg and Tarjan have shown that Dijkstra’s algorithm only performs  $O(n \log n)$  *decrease-key* operations with high probability if edge lengths are independently generated from the same probability distribution [10].

We conclude from our studies that a rank-sensitive priority queue is a respectable data structure to use for shortest path computation, but most likely not the fastest choice available in practice. A possibly interesting question for future research might be determining which types of random graphs (if any) allow us to obtain provable expected performance bounds for Dijkstra’s algorithm with a rank-sensitive priority queue that are close to those of a Fibonacci heap.

## References

1. R.K. Ahuja, K. Melhorn, J.B. Orlin, and R.E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 37:213–22, 1990.
2. C.R. Aragon and R. Seidel. Randomized search trees. *Algorithmica*, 16:464–497, 1996.
3. M. Bădoiu, R. Cole, E.D. Demaine, and J. Iacono. A unified access bound on comparison-based dictionaries. *Theoretical Computer Science*, 382(2):86–96, 2007.
4. B.V. Cherkassky, A.V. Goldberg, and T. Radzik. Shortest path algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73:129–174, 1996.
5. R. Cole. On the dynamic finger conjecture for splay trees. part II: The proof. *SIAM Journal on Computing*, 30(1):44–85, 2000.
6. R. Cole, B. Mishra, J. Schmidt, and A. Siegel. On the dynamic finger conjecture for splay trees. part I: Splay sorting log n-block sequences. *SIAM Journal on Computing*, 30(1):1–43, 2000.
7. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.
8. M.J. Fischer and M.S. Paterson. Fishspears: A priority queue algorithm. *Journal of the ACM*, 41(1):3–30, 1994.
9. I. Galperin and R.L. Rivest. Scapegoat trees. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 165–174, 1993.
10. A.V. Goldberg and R.E. Tarjan. Expected performance of dijkstra’s shortest path algorithm. Technical Report TR-96-063, NEC Research Institute, 1996.
11. J. Iacono and S. Langerman. Queaps. *Algorithmica*, 42(1):49–56, 2005.
12. D.D. Sleator and R.E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.



**Fig. 3.** Performance of rank-sensitive priority queues versus binary heaps and Fibonacci heaps on (a) the USA road network, (b) random grid graphs, (c) random  $G_{n,m}$  graphs, and (d) random Euclidean graphs.