Faster Kinetic Heaps and Their Use in Broadcast Scheduling

(Extended Abstract)

Haim Kaplan* Robert E. Tarjan† Kostas Tsioutsiouliklis‡

Abstract

We describe several implementations of the *kinetic heap*, a heap (priority queue) in which the key of each item, instead of being fixed, is a linear function of time. The kinetic heap is a simple example of a kinetic data structure of the kind considered by Basch, Guibas, and Hershberger. Kinetic heaps have many applications in computational geometry, and previous implementations were designed to address these applications. We describe an additional application, to broadcast scheduling. Each of our kinetic heap implementations improves on previous implementations by being simpler or asymptotically faster for some or all applications.

1 Introduction

1.1 Problem Definition A *parametric heap* is a heap (priority queue) in which the key (priority) of an item, instead of being constant, is a linear function of a single parameter t. Specifically, a parametric heap supports the following operations:

make-heap(h): Create a new, empty heap h.

insert(i,a,b,h): Insert item i, with key $at + b$, into heap h, assuming i is not already in h. Here a and b are real numbers.

delete(i,h): Delete item i from heap h, assuming i is in h.

find-min(h, t): Return an item in heap h whose key is minimum for the given value of t.

delete-min(h, t): Find an item in heap h whose key is minimum for the given value of t, delete this item from h, and return it.

The *delete-min* operation is a combination of *find-min* and *delete*; some applications require only *make-heap*, *insert*, and *delete-min*.

If i is an item in a parametric heap, we denote the key of i by $a_i t + b_i$; a_i is the *key coefficient* and b_i is the *key constant*.

A *kinetic heap* is a parametric heap whose operations satisfy the following *monotonicity condition*: successive t-values of *find-min* and *delete-min* operations are nondecreasing. For a kinetic heap, we think of the parameter t as time; we define the *current time* t_c to be the value of t for the most recent *find-min* or *delete-min* operation, or $-\infty$ if no such operation has occured.

We sometimes allow the following additional operation on a kinetic heap:

decrease-key(i,a,b,h): Replace the key of item i in heap h by $at + b$, assuming that i is in h and that, for any time $t \geq t_c$, the new key of i is no greater than the old key.

A decrease-key operation can be performed as a *delete* followed by an *insert*, but, as we shall see, some kinetic heap implementations allow *decrease-key* to be performed faster. If *delete* or *decrease-key* or both are supported, we assume that each item in a heap has stored with it a pointer to its location in the data structure, so that this location can be found in constant time, without a search.

A *semi-dynamic* (*parametric* or *kinetic*) *heap* is one in which the *insert* operation is not supported, but the *make-heap* operation is extended to allow creation of an initial heap containing an arbitrary set of items with specified keys.

Parametric and kinetic heaps as we have defined them are *min-heaps*: a query operation returns a minimum-key item. Equivalently, we can define a *parametric* or *kinetic max-heap* by replacing *find-min*, *delete-min*, and *decrease-key* by the operations *find-max*, *delete-max*, and *increase-key*, respectively, each having the appropriate definition. We can convert a min-heap to a max-heap, or vice versa, merely by negating all keys. The broadcast scheduling application that we shall consider requires a max-heap.

1.2 Previous work Parametric and kinetic heaps have a large assortment of applications in computational geometry [12, 11, 6]. The problem of maintaining a parametric heap is equivalent to that of maintaining the lower envelope of a collection of lines in the Euclidean

*School of computer science, Tel Aviv University, Tel Aviv 69978.

†InterTrust Technologies, 460 Oakmead Parkway, Sunnyvale, CA 94086, and Department of Computer Science, Princeton University, Princeton, NJ 08544. Research at Princeton University partially supported by the NSF, Grant No. CCR-9626862.

‡Department of Computer Science, Princeton University, Princeton, NJ 08544. Research at Princeton University partially supported by the NSF, Grant No. CCR-9626862.

plane, subject to insertions and deletions of lines. The projective dual of this problem is that of maintaining the convex hull of a collection of points in the plane, subject to insertions and deletions of points. For either of these problems, one may allow queries other than the one corresponding to *find-min*. See [12, 11, 6].

Most of the previous work on parametric and kinetic heaps was done in the computational geometry setting. We restate the previous results in our data structure setting. To our knowledge, Overmars and van Leeuwen [12] presented the first efficient implementation of parametric heaps. Their structure supports a *find-min* query on an n-item heap in $O(\log n)$ time, a *make-heap* in $O(1)$ time, and each of the other operations in $O(\log^2 n)$ time, worst-case. Hershberger and Suri [11] and independently Chazelle [8] described an implementation of semi-dynamic parametric heaps that is a variant of the Overmars-van Leeuwen structure. Their structure supports *find-min* queries in $O(\log n)$ time, worst-case. The time for an n-item *make-heap* operation is $O(n \log n)$, and the total time for up to n deletions is $O(n \log n)$. Chan [6] applied the dynamization technique of Bentley and Saxe [4] and other ideas to the Hershberger-Suri-Chazelle structure to obtain an implementation of parametric heaps that supports *find-min* in $O(\log n)$ worst-case time and *insert* and *delete* in $O(\log^{1+\epsilon} n)$ amortized time for an arbitrary positive ϵ. Very recently, Brodal and Jacob [5] have described an implementation of parametric heaps that reduces the amortized insertion time to $O(\log n \log\log\log n)$ and the amortized deletion time to $O(\log n \log\log n)$. We describe a similar result, obtained concurrently with and independently of the work of Brodal and Jacob, in Section 3.

Kinetic heaps, being a special case of parametric heaps, can be supported using any implementation of parametric heaps, but they may have simpler or more efficient implementations. Chan [7] described a simplified version of his parametric heap structure that supports kinetic heaps in the same amortized time bounds. Basch, Guibas, and Hershberger [3] treat the kinetic heap as an instance of a *kinetic data structure*, one in which the data, instead of being fixed, change smoothly over time. Their major focus was on the generality of the kinetic paradigm and on data structures more complicated than kinetic heaps. They did, however, give a method for maintaining a static n-item kinetic heap (no insertions or deletions) with a total time bound of $O(n \log^2 n)$ plus $O(1)$ per *find-min*. Basch in his Ph.D. thesis [2] describes two additional kinetic heap data structures that support insertions and deletions. The first structure, called the Kinetic Heater, is based upon treaps. Using the Kinetic Heater one can perform a sequence of n insertions and deletions, interspersed with *find-min* operations in $O(n\alpha(n) \log^2 n)$ expected amortized time. The second structure, called the Kinetic Tournament is deterministic, and supports a sequence of n insertions and deletions, interspersed with *find-min* operations in $O(n\alpha(n) \log^2 n)$ amortized time. All of the data structures so far mentioned, as well as all of our own, require space linear in the number of items.

1.3 Our Results Our interest in kinetic heaps was triggered by the observation that various policies for broadcast scheduling [17, 1] can be implemented efficiently using kinetic max-heaps. In this application, *increase-key* occurs very frequently, so it is important to make it especially efficient. Furthermore the key coefficients are integers whose sizes are related to the number of *increase-key* operations. As we shall see, these facts can be exploited in the design of efficient kinetic heaps for use in broadcast scheduling.

This paper describes the results of our systematic exploration of better ways to implement kinetic heaps. We present four different implementations, each of which has some advantage over previous implementations and none of which is dominated by the others in all situations. We also show how to use kinetic heaps to implement broadcast scheduling policies.

The paper contains six sections in addition to this introduction. In Section 2 we describe the *simple kinetic heap*. This structure is based on the kinetic paradigm of Basch, Guibas, and Hershberger. It consists of a single balanced tree: it supports *make-heap* in $O(1)$ worst-case time, *find-min* in $O(1)$ amortized time, and *insert* and *delete* in $O(\log^2 n)$ amortized time. Although our other structures are more efficient asymptotically, the simplicity of this structure may make it the best choice in an actual application such as broadcast scheduling.

In Section 3 we describe the *balanced kinetic heap*. This structure extends the development thread from Overmars and van-Leeuwen [12], through Hershberger and Suri [11] and Chazelle [8], to Chan [6]. It is simpler than Chan's structure in some ways and more complicated in others. The amortized time per *find-min* or *insert* is $O(\log n)$ and the amortized time per deletion is $O(\log n \log\log n)$. The bound for *find-min* can be reduced to $O(1)$ at the cost of a little additional complication. This is our most efficient structure for the case of keys with arbitrary real-valued coefficients, assuming that *decrease-key* is not explicitly supported. This is the only one of our structures that extends to the parametric heap problem. We discuss this extension, and the relation of our work to that of Brodal and Jacob [5], in Section 3.

In Section 4 we describe the *Fibonacci kinetic* *heap* (F-kinetic heap). This structure was designed to support *decrease-key* operations especially efficiently; it is based on the Fibonacci heaps of Fredman and Tarjan [10]. The structure has amortized time bounds of $O(\log n)$ for *find-min*, *insert*, and *decrease-key*, and $O(\log^2 n)$ for *delete*. These bounds are a log factor worse than those of Fibonacci heaps with constant keys. As in the case of balanced kinetic heaps, we can reduce the amortized time per *find-min* to $O(1)$ at the cost of a little additional complication. In Section 5 we describe in detail the broadcast scheduling application. In Section 6 we present the *incremental kinetic heap*, which exploits the constraints that arise in the broadcast scheduling application to reduce the amortized time bound for every operation to $O(\log n)$. We conclude in Section 7 with some remarks and open problems. We present the data structures in Sections 2,3, and 4 as min-heaps; we present the structure in Section 6 as a max-heap, since this is what is required for broadcast scheduling.

2 Simple Kinetic Heaps

Our first data structure, the *simple kinetic heap*, is based on the kinetic paradigm of Basch, Guibas, and Hershberger. The structure is a binary tournament on the items, with winners determined by key (smaller key wins). As time advances, winners become invalid and must be recomputed. Following the kinetic paradigm, we maintain a heap to allow access to the earliest invalid winner. In our structure this heap is a second tournament on the *same* underlying binary tree. This double tournament idea works for any ordering of the items in the external nodes of the tree. We exploit this freedom by ordering the items by key coefficient. This ordering, in combination with using one tree for both tournaments, leads to an especially simple and efficient structure; our bounds for a fully dynamic kinetic heap match those of Basch et al. for a static kinetic heap.

Here is a more detailed description. A *simple kinetic min-heap* is a balanced binary tree (such as a red-black tree [14]) in which the items are stored in the external nodes, ordered left-to-right in decreasing key coefficient order, with a tie broken by key constant. That is, item i is to the left of item j if and only if $a_i > a_j$ or $a_i = a_j$ and $b_i > b_j$. With this ordering, as t increases the position of the item or items of minimum key moves right. We store additional information in the internal nodes of the tree. Each internal node contains a pointer to its predecessor, which is the rightmost external node in its left subtree. These pointers allow insertion of a new item in its correct position, by searching from the root doing appropriate comparisons to decide whether to branch left or right. Each node contains a *winner*, which is the item in its subtree of minimum key at the current time t_c. The winner is defined recursively as follows: if x is an external node, then the winner in x is the item in x. If x is an internal node, with left child y having winner i and right child z having winner j, then the winner in x is i if $a_i t_c + b_i \leq a_j t_c + b_j$, and is j otherwise. Each internal node x has a *swap time* $s_x = (b_j - b_i)/(a_i - a_j)$ if $a_i \neq a_j$, $s_x = -\infty$ otherwise, where i is the winner in the left child of x and j is the winner in the right child of x. This swap time is the time at which the winner in x switches from i to j. Finally, each node x has a *minimum future swap time* f_x, which is the smallest swap time among those of it and its descendants that is greater than t_c, or ∞ if there is no such swap time. The minimum future swap time is defined recursively as follows: if x is an external node, $f_x = \infty$; if x is an internal node, $f_x = \min\{\max\{t_c, t_x\}, \max\{t_c, f_y\}, \max\{t_c, f_z\}\}$ if this value exceeds t_c, $f_x = \infty$ otherwise, where y and z are the children of x.

Having the winner in the root allows a *find-min* query to be answered in $O(1)$ time if the query does not cause the current time to change. An insertion or deletion of an item can be performed as in a standard balanced search tree; the extra information (predecessors, winners, swap times, and minimum future swap times) can be updated bottom-up along the single path in the tree affected by the insertion or deletion (including any needed rotations for rebalancing; see [14]). Thus the time for an insertion or deletion is $O(\log n)$.

The only part of the implementation that is not straightforward is advancing the current time, which must be done if a query occurs whose time is later than that of the previous query. To advance the current time, say from t_c to t'_c, we search down from the root, using minimum future swap times to locate all nodes having a swap time greater than t_c but less than or equal t'_c. The paths from the root to these nodes form a subtree. We recompute winners, swap times, and minimum future swap times for all nodes in this subtree, working bottom-up. The cost of this search and recomputation is proportional to the size of the affected subtree, which is $O(\log n)$ per node whose swap time changes from future to past or present. Although the worst-case cost for advancing the current time is $\Theta(n)$, we can charge this cost to the insertions and deletions by using a potential function that is $O(\log n)$ times the number of nodes whose swap times are in the future. (In this extended abstract we omit the details of the analysis, which uses the standard "potential function" paradigm of amortized analysis [15].) The result is an amortized bound of $O(1)$ per *find-min* and $O(\log^2 n)$ per *insert* and *delete*. (Each update can increase the potential by $O(\log^2 n)$; advancing the current time decreases the potential by an amount that pays for the advance.)

A variant of this data structure has the items stored in the *internal* nodes of a balanced binary tree, in symmetric order by decreasing key coefficient. Then the winners are (in general) determined by three-way comparisons (among the key of the item in a node and the keys of the winners in its two children) instead of two-way comparisons, and swap times must be appropriately redefined.

3 Balanced Kinetic Heaps

To obtain an asymptotically more efficient data structure, we start with the parametric kinetic heap of Overmars and van Leeuwen, which as it happens is related to the structure of Section 2. To obtain the Overmars-van Leeuwen data structure we modify the structure of Section 2 as follows. Instead of computing a single winner for each node x, we compute the sequence $W(x)$ of winners for all possible times. This sequence is ordered not only by time (each item in the sequence is a winner for a single continuous interval of time) but by key coefficient (smallest to largest time corresponds to largest to smallest key coefficient). Furthermore, the sequence of winners of a node x is formed from the sequences of winners of its left child y and right child z in a simple way: $W(x)$ consists of a prefix of $W(y)$ catenated with a suffix of $W(z)$. Furthermore, given $W(y)$ and $W(z)$, we can compute $W(x)$, and the parts of $W(y)$ and $W(z)$ not in $W(x)$ in logarithmic time, if $W(y)$ and $W(z)$ are represented as balanced trees.

The Overmars-van Leeuwen structure consists of a balanced binary tree with the items in the external nodes in decreasing order by key coefficient. The root contains its winner sequence, represented as a balanced tree, along with the split point between the parts coming from each of its children. Each non-root node contains the part of its winner sequence not part of its parent winner sequence, represented as a binary search tree, as well as information about how its own winner sequence is split between its children. A *find-min* query can be performed in $O(\log n)$ time by doing binary search on the winner sequence of the root. An update (either an insertion or a deletion) is done by following the path down through the tree to the insertion/deletion point, concurrently splitting the winner sequences of nodes along the search path and reassembling the winner sequences of their children. Once the insertion/deletion point is found and the desired item is inserted or deleted, a walk back up the search path is done, including reassembly of winner sequences of nodes along the path by splitting from the winner sequences of their children. Any necessary rebalancing is done along the way. With this method, an insertion or deletion takes $O(\log^2 n)$ time; one factor of $\log n$ comes from the length of a search path and the other comes from the time to split and reassemble winner sequences at each node by splitting and catenating the corresponding balanced binary trees.

We use a variant of the Overmars-van Leeuwen structure as a semi-dynamic kinetic (deletions only). Specifically, we represent each winner sequence or subsequence by a finger search tree [16] instead of a search tree. This allows us to perform a split of a winner sequence in time logarithmic in the size of the smaller split part, as compared to logarithmic in the size of the entire sequence if an ordinary search tree is used. We combine this with the observation of Hershberger and Suri [11] and Chazelle [8] that, if there is no rebalancing, deletions only cause winning items to move up the tree, never down. We also slightly modify the Overmars-van Leeuwen deletion algorithm. By combining these ideas, we are able to obtain a semi-dynamic kinetic heap with an amortized $O(n)$ initialization time for an n-item heap (assuming the items are given in key-coefficient order), $O(\log n \log\log n)$ amortized time for *delete*, and $O(1)$ amortized time for *find-min*. To handle *find-min*, we split the winner sequence of the root at the item that is the winner at the current time. (In this extended abstract we omit the amortization argument.)

In this way we improve on the initialization time of the Hershberger-Suri and Chazelle structures, reducing their $O(n \log n)$ bound to $O(n)$, while increasing the amortized time of *delete* to $O(\log n \log\log n)$. This trade-off is advantagous for the next step, in which we add the ability to handle insertions by applying the dynamization technique of Bently and Saxe [4]. We use a simple binary counting scheme in which there are up to $\log n$ separate semi-dynamic structures, each corresponding to a different power of two. Insertion is done by creating a new single-node structure and then if necessary repeatedly combining structures (corresponding to a binary addition of one), until there is again at most one structure for each power of two. Combining two structures requires merging their item lists in key-coefficient order, forming a new binary tree, and assembling and splitting winner sequences, working bottom-up through the tree. This all takes linear time, including amortized time charged against the initialization for later deletions. Overall, a given item participates in $O(\log n)$ initializations, which results in an $O(\log n)$ amortized time for insertion. The amortized deletion time remains $O(\log n \log\log n)$. The amortized *find-min* time is $O(\log n)$, because $O(\log n)$ structures must be queried, but this can be reduced to $O(1)$ by adding an additional top-level structure. (We omit the details in this extended abstract.)

Chan's kinetic heap structure [7] is similar to ours except that he uses doubly-linked lists instead of finger search trees to obtain a semi-dynamic structure, and his bound for updating this structure is the same as that of Hershberger and Suri [11] and Chazelle [8], namely $O(n \log n)$ for initialization and any number of deletions. To obtain his $O(\log^{1+\epsilon} n)$ update bound for a fully dynamic structure, he uses non-binary dynamization and a constant number (related to ϵ) of bootstrapping recursions of the entire structure. Thus, though his semi-dynamic structure is simpler than ours, his fully dynamic structure is more complicated and asymptotically slower.

Our semi-dynamic kinetic heap becomes a semi-dynamic parametric heap by omitting the split at the current time of the winner sequence at the root. The worst-case query time is $O(\log n)$. Brodal and Jacob independently obtained a semi-dynamic parametric heap with the same time bounds. Instead of using finger search trees to represent winner sequences, they split such a sequence into blocks of size $\log n$, each represented by a balanced tree, with the overall sequence represented as a doubly-linked list of blocks. Applying dynamization as decribed above to either our structure or their structure results in a fully dynamic parametric heap with $O(\log^2 n)$ worst-case *find-min* time, $O(\log n)$ amortized *insert* time, and $O(\log n \log\log n)$ *delete* time. By adding an additional top level structure as well as some bootstrapping, one can reduce the *find-min* time to $O(\log n)$, but apparently at the cost of increasing the amortized *insert* time to $O(\log n \log\log\log n)$. Details can be found in [5]; see also [6].

4 Fibonacci Kinetic Heaps

In this section we develop the Fibonacci kinetic heap (F-kinetic heap), which supports *find-min*, *insert* and *decrease-key* in $O(\log n)$ amortized time and *delete* in $O(\log^2 n)$ amortized time. The Fibonacci kinetic heap is based on the Fibonacci heap (F-heap), which is a heap structure for items of fixed key [10]. In this section we shall assume familiarity with F-heaps, but we review the key properties that underly our F-kinetic heap structure.

A Fibonacci heap consists of a collection of heap-ordered *Fibonacci trees* (F-trees) whose nodes store the heap items. Each node has a non-negative integer *rank* that is $O(\log n)$. An atomic operation on such trees is *linking*, which combines two trees by making the root of larger key a child of the root of the smaller key, breaking a tie arbitrarily. We only link trees whose roots have equal rank.

Our implementation of the F-kinetic heap has an F-heap as the underlying structure, but we delay doing links until we can be sure that the outcomes of the corresponding comparisons are guaranteed not to change in the future. Delaying the links in this way requires two auxiliary structures: one keeps track of trees with roots of the same rank that cannot yet be linked; the other keeps track of times at which links can be done.

Our data structure for an F-kinetic heap consists of a collection of whose nodes are the items. These F-trees are grouped into sets S_r, $0 \leq r \leq c \log n$, for an appropriate positive constant c. Set S_r contains every F-tree of root rank r. Within each set S_r, the trees are totally ordered by the key of the root and stored in a sorted list in increasing root-key order at the current time t_c. We implement each of these sorted lists as a balanced tree such as a red-black tree [14]. We shall denote an F-tree whose root is item i by T_i. Within each sorted list S_r, we maintain the property that each pair of consecutive trees T_i and T_j satisfy $a_i \geq a_j$ and $b_i \leq b_j$. (Recall that the keys of i and j are $a_i\, t + b_i$ and $a_j\, t + b_j$, respectively.) This property implies that we can use the key coefficients of the root items rather than their keys to search the lists S_r. For each such pair T_i, T_j, we compute the time t_{ij} at which their keys switch order by the formula $t_{ij} = (b_j - b_i)/(a_i - a_j)$. The ordering relationships on S_r imply that $t_{ij} \geq t_c$, where t_c is the current data-structure time. We maintain a separate heap H containing each pair T_i, T_j occurring consecutively in some S_r, with t_{ij} as the key of the pair in H. Since time only increases, the heap H is actually a monotone heap [9]. (We do not use this property in our results.)

Two atomic operations on this data structure are to insert a new F-tree into the appropriate set S_r, and to delete two consecutive F-trees from some set S_r and link them to form a new F-tree of rank $r + 1$. One of these operations can trigger the other, and this process can cascade. In particular, suppose we are given a set U of F-trees that we wish to insert into the structure, while preserving the ordering properties. We can do this by repeating the following steps until U is empty: Delete from U some tree, say T_i of rank r. Insert T_i into S_r by searching in S_r using the key coefficients of the root items. If tree T_j follows T_i in S_r and $t_{ij} < t_c$, delete T_i and T_j from S_r, link them, and insert the new tree into U. Otherwise, if tree T_k precedes T_i in S_r and $t_{ki} < t_c$, delete T_k and T_i from S_r, link them, and insert the new tree into U. In addition, as trees are inserted into and deleted from the sets S_r, update H correspondingly: each insertion of a tree into a set S_r requires up to one deletion from and two insertions into H; each deletion of a tree from a set S_r requires up to two deletions from and one insertion into H.

The time required for this process is $O((u+v)\log n)$, where u is the initial size of U and v is the number of links done: each insertion of an item into or deletion of an item from a set S_r requires $O(\log n)$ time, including the needed updates to H; each link reduces the total number of F-trees in the structure by one. This allows us to amortize away the cost of the links, by assigning a potential of $c \log n$ for an appropriate constant c to each F-tree. Then the amortized cost of the cascading tree insertion process is $O((1+u)\log n)$.

We can implement the various kinetic heap operations using cascading insertion. The running time analysis is essentially the same as for F-heaps, except that we multiply the potential by $\log n$. (We omit the amortized analysis in this extended abstract.)

To perform *find-min(h,t)*, we first advance the current time to t and then return the smallest item among the roots of the first trees in the respective sorted sets S_r. We advance the current time to t as follows. We initialize U to be empty and repeat the following steps until every pair T_i, T_j in H has $t_{ij} \geq t$: delete from H a pair T_i, T_j with $t_{ij} < t$, delete T_i and T_j from the set S_r containing them (updating H appropriately), link T_i and T_j, and add the new tree to U. Once every pair T_i, T_j in H has $t_{ij} \geq t$, do cascading insertion on U. The *find-min* operation takes $O(\log n)$ amortized time.

To perform *insert*(i,a,b,h), we create a new, one-node F-tree T_i, give item i key $at+b$, initialize $U = \{T_i\}$, and do cascading insertion on U. This takes $O(\log n)$ amortized time.

To perform *decrease-key*(i,a,b,h), we determine whether i is the root of some F-tree or not. If it is, say of T_i, we delete T_i from the set S_r containing it (updating H appropriately), assign i its new key, set $U = \{T_i\}$, and do cascading insertion on U. This takes $O(\log n)$ amortized time. If, on the other hand, i is not the root of some thin tree, we do a *decrease-key* on the F-tree containing i as described in [10]. This results in a set of new F-trees, one of which has root i. We initialize U to contain these trees. In addition, the rank of the root of the tree previously containing i may drop by one. If this happens, we delete this tree from the set S_r containing it (updating H appropriately) and add this tree to U. Then we do cascading insertion on U. This case of *decrease-key* also takes $O(\log n)$ amortized time.

The last operation on heaps with time is *delete*. To perform *delete*(i,h), we do an *decrease-key* on i in the F-tree containing it, collecting all the newly formed F-trees into a set U. If the *decrease-key* decreases the rank of the tree previously containing i, we delete this tree from the set S_r containing it (updating H appropriately) and add this tree to U. We delete T_i from U, delete the root of T_i (which is i), and add to U each new tree rooted at an old child of i. Finally, we do cascading insertion on U. Deletion takes $O(\log^2 n)$ amortized time.

We mention one final implementation detail. Each node must contain not only its rank r but also a pointer to the corresponding set S_r, to allow insertion of a F-tree into the correct set S_r. Since the rank of a node only changes by one (up or down), maintaining this information takes only $O(1)$ time per rank change.

By adding an additional top-level structure (details omitted in this extended abstract), we can reduce the amortized time of *find-min* to $O(1)$.

5 Broadcast Scheduling via a Kinetic Heap

Broadcast scheduling is the following communication problem. We are given a collection of items (such as web pages) and a single channel, over which we can send one item at a time, simultaneously to all interested parties. Requests for various items arrive from time-to-time. When the channel is free, we select one among the requested items to broadcast over the channel; this transmission satisfies all existing unsatisfied requests for the item. Such a system requires a policy for deciding, on-line (that is, without knowledge of future requests) which item to send next. One promising scheduling policy is *Longest Wait First* (LWF) [17] in which an item receives a score that is the sum of the waiting times of all its unsatisfied requests, and an item of maximum score is sent. A related scheduling policy is RxW [1], in which the score of an item is the number of unsatisfied requests times the waiting time of the earliest unsatisfied request; again, an item of maximum score is sent.

It has been argued [1, 17] that LWF is impractical because of high overhead to determine the next item to send. Indeed, RxW was proposed as a lower-overhead alternative to LWF. Nonetheless, even RxW can be expensive to implement [1], a problem addressed in [1] by using a parameterized family of easier-to-compute approximations to RxW. Both LWF and RxW have the property that the relative order of items by score can change with time, even in the absence of new requests, which means that standard heap (priority queue) data structures [14] cannot be used in any obvious way to keep track of maximum-score items. The scheduling algorithms described in the literature scan all or many of the requested items each time a maximum-score item is to be found, resulting in linear or almost-linear scheduling time per broadcast. In contrast, our results allow both LWF and RxW to be implemented in logarithmic time per item request.

We can use a kinetic max-heap h to implement a scheduling policy such as LWF or RxW. We call an item *active* if it has at least one unsatisfied request. The items in the heap h are exactly the active items, with keys equal to their scores. We explicitly make the following assumptions, which are implicit in the previous papers on the problem. First, we assume that the time to perform a heap operation is small compared to the time to broadcast an item, and therefore one can ignore the fact that the item scores change during the heap operation. Second, we assume that requests arrive in an input queue and do not contribute to the scores until removed from the queue and processed.

We implement LWF as follows. Initially there are no active items, and h is empty. Suppose a request for an item i occurs at time t_s. If i is currently inactive, we perform *insert*$(i, 1, -t_s, h)$, making i active; otherwise, we perform *increase-key*$(i, a+1, b-t_s, h)$, where $at+b$ is the key of item i before the update. Suppose a broadcast slot becomes available at time t_s. We perform *delete-max*(h, t_s) and broadcast the item returned.

We implement RxW in the same way, the only difference being in the third parameter of *increase-key*: to handle an additional request for an active item i, we perform *increase-key*$(i, a+1, (a+1)b/a, h)$, where $at+b$ is the key of item i before the update.

In this implementation, the only deadline-driven operation is *delete-max*: when a broadcast slot becomes available, we want to choose as rapidly as possible an item to broadcast. The other operations can all proceed concurrently with the current broadcast. In our implementations of kinetic heaps in Sections 2, 3, 4, and 6, the actual time for *delete-max* may be long when the time difference from a previous *delete-max* or *find-max* is long. In these situations the current time of the kinetic heap is far behind the time of the operation, and we may need to invest a considerable amount of work to advance it. We can avoid such situations in an actual implementation by periodically performing *find-max* operations to advance the current time of the kinetic heap. This way when a broadcast is due the actual time of the *delete-max* is not likely to be long. We can also break the *delete-max* operation into a *find-max* followed by a *delete*; The broadcast can begin once the *find-max* is complete.

6 Incremental Kinetic Heaps

In this section we describe the incremental kinetic heap, which exploits extra constraints in the broadcast scheduling application. Specifically, we make the following assumption, which holds for that scheduling application.

Assumption 6.1. *When an item i is inserted into the heap, $a_i = 1$, and each* increase-key *on i increases a_i by exactly one.*

We develop a data structure for a kinetic max-heap obeying Assumption 6.1 that has the following amortized time bounds: $O(1)$ (worst-case) for *find-min*, $O(1)$ for *insert*, $O(\log \min\{k, n\})$ for the k^{th} *increase-key* on a given item, and $O(\log n)$ for *delete* or *delete-min*. Note that we describe here a max-heap rather than a min-heap as in previous sections since that is what needed for broadcast scheduling. The main idea we use is to group items by the value of their key coefficient, using a separate heap B_a for each value a of a key coefficient. The heaps B_a are F-heaps for constant keys. To reduce the time for *increase-key* operations, we use the fact that F-heaps support the operation of moving an F-tree from one heap to another in $O(1)$ time. This movement of trees among the heaps B_a means that the grouping of items by the value of their key coefficient is not perfect: items can become displaced from their home heaps, to be moved home later.

The data structure consists of two parts. One part is a collection of F-heaps B_a for various integer values a. Each item is in exactly one of the heaps B_a. The *home heap* of an item i with $a_i = a$ is B_a. Item i is *displaced* if it is not in its home heap. A heap B_a exists in the structure if B_a is non-empty or if some displaced item has B_a as its home heap. For each item, we maintain a pointer to its home heap. We also maintain with each heap a count of the number of items for which it is home. We maintain the existing heaps B_a in a doubly-linked list L in increasing order on a. The size of L is at most $2n$, since each item can account for at most two existing heaps B_a (one its home and one containing it). Because of Assumption 6.1, we can use the list of heaps to find in constant time the heaps needing updating in *insert* and *increase-key* operations. For *insert*, we need to find B_1, or create and insert it if it does not exist. For *increase-key*, given an item with a home heap B_a, we need to find B_{a+1}, or create and insert it if it does not exist.

We maintain certain invariants on the heaps B_a. At most one F-tree root in B_a is displaced. Such a displaced root is a maximum-key item and is indicated by the maximum pointer of B_a. If an item j is a child of item i in heap B_a, then $a_j \leq a_i$ and the key of i is no smaller than the key of j (at the current time and hence at all future times). Finally, if i is a displaced item in heap B_a, then $a_i > a$ if i is a root and $a_i < a$ if i is a non-root. A displaced root has a pointer to the heap B_a containing it.

The second part of the structure is a doubly-linked list M of some (or all) of the maximum-key items of the heaps B_a. An item i is on M if and only if the key of i at the current time is greater than the key of every other item j such that $a_i \leq a_j$. (We assume that ties are broken consistently, say by a total ordering of the items.) Items on M are ordered in decreasing order by key and hence in increasing order on a. The first item on M is thus the item of globally maximum key. For each item on M other than the first, we maintain the time t_i at which the key of i and that of its predecessor on M switch order. Each such t_i is greater than the current time t_c.

To represent M, we use a heterogeneous red-black finger search tree with a finger at the front [16]; the leaves are the elements of M. We store cumulative information about the t_i-values in this tree, as follows. Each tree node except for those on the left spine (leftmost path) contains the minimum t_i-value among its leaves. Each tree node on the left spine contains the minimum t_i value among the leaves that follow it in symmetric order. With this structure, we can insert or delete an item i in $O(\log \min\{a_i, n\})$ amortized time. We can also delete k predecessors of an item i in M in $O(k + \log \min\{a_i, n\})$ amortized time. (Item i is at most a_i places from the front, and there are at most n items.)

Having described the data structure, we proceed to explain how to perform the kinetic heap operations. To perform *find-min*(h, t), we repeat the following step until every item i in M other than the first has $t_i > t$: Find an item i in M with t_i minimum. Delete from M the item preceding i, and any additional predecessors with keys no greater than that of i. Then we return the first item on M.

To perform *insert*(i,1,b,h), we create a new, one-node F-tree with root i having key $t + b$, and insert it into B_1, first creating B_1 and adding it to L if necessary. If i becomes the maximum-key item in B_1, we update M by inserting i if necessary. In addition, if the previous maximum-key item in B_1 is displaced, we move it, along with its entire F-tree, to its home heap B_a. This may require updating the maximum pointer of B_a; and, if this happens, moving home the F-tree rooted at the previous maximum-key item of B_a. Thus the movement of displaced roots home can cascade: each movement shifts an F-tree from one heap to another of smaller index and may trigger another movement home.

To perform *increase-key*(i,a,b,h), we create a new empty heap B_a if it does not exist and insert it into L. If i is the maximum-key item of some heap $B_{a'}$ (for $a' \leq a - 1$), we merely insert i into M if necessary and delete predecessors of i in M if necessary to restore the invariant on M. In this case i becomes displaced, if it was not displaced already. If, on the other hand, i is not the maximum-key item in the heap containing it, then we do an ordinary F-heap *increase-key* on i as described in [10]. This makes i a root if it was not one already, and it may create some other new F-tree roots via cascading cuts (see [10]). We move i and each other new root, along with its entire F-tree, to its home heap, updating the maximum pointer of that home heap. If i is now in heap B_a and is now the maximum-key item in B_a, we insert i into M if necessary and delete predecessors of i in M until the invariants on M are restored. Moving the other roots home does not require updating M, because such a root was not maximum-key in its previous heap, and it moves to a heap of smaller index, which means that it cannot force a change to M. On the other hand, a root moved home may become the new maximum-key item in its home heap. If the previous maximum-key item is displaced, it must be moved home along with its entire tree. A movement home of this kind can trigger a cascade of movements home, just as in the case of an insertion. We perform all such cascading movements home. We complete the *increase-key* by deleting B_{a-1} from L if it is now empty and home to no items. Such a deletion can occur whether or not i was initially a maximum-key item.

The operation *delete*(i, h) has two cases, depending on whether i is the maximum-key item in the heap B_a containing it. If so, we begin by deleting i from M if i is in M. Item i is the root of an F-tree in B_a. We delete item i from this tree, resulting in a new tree for each previous child of i. If any of these new roots is displaced, we move it and its tree to its home heap, say $B_{a'}$, updating the maximum pointer of $B_{a'}$ if necessary. The invariants on heap B_a imply that $a' \leq a$. We do not yet update M to reflect a change in the maximum pointer of $B_{a'}$, but if this change triggers a cascade of additional movements home as in *insert* and *increase-key*, we perform these movements home. In heap B_a we proceed as in ordinary F-heap deletion (see [10]), linking trees with roots of equal rank until no such linking is possible and then resetting the maximum pointer of B_a. Finally, we process the heaps in L starting with B_a and proceeding through heaps of smaller indices (working toward the front of L), updating M if necessary to reflect any changes in the maximum pointers of these heaps. (In particular, the new maximum item of B_a may need to be inserted into M.)

On the other hand, if i is not the maximum-key item in B_a, we begin the deletion by doing an ordinary F-heap *increase-key* on i in B_a. This makes i a tree root if it was not already, and may create additional new roots. We delete i, making each of its children a new root. We move each new root home, along with its entire tree, updating the maximum pointer of the home heap if necessary but not updating M (no updating is necessary). We perform cascading movements home if necessary, as in *insert*, *increase-key*, and the other case of *delete*.

An amortized analysis, which we omit from this extended abstract, yields the claimed time bounds for the operations.

7 Remarks

The asymptotically fastest known kinetic heap structure not explicitely supporting the *decrease-key* operation is obtained by applying dynamization to either our semi dynamic structure from Section 3 or to the semi-dynamic structure of Brodal and Jacob. Either structure obtained in this way supports *insert* in $O(\log n)$ amortized time and *delete* in $O(\log n \log\log n)$ amortized time. An immediate open question is to find a structure that supports *delete* as well as *insert* in $O(\log n)$ amortized time. Designing a semi-dynamic kinetic heap with $O(n)$ amortized initialization time and $O(\log n)$ amortized deletion time would immediately yield such a result via dynamization.

If *decrease-key* is explicitly supported in $(\log n)$ amortized time and there are no additional assumptions on the keys, the best structure is our Fibonacci kinetic heap. This structure supports *delete* in $O(\log^2 n)$ amortized time. Whether this bound can be improved is another immediate open question.

Both our semi-dynamic parametric heap structure of Section 3 and the structure of Brodal and Jacob yield (via the use of complicated additional structures and dynamization [6, 5]) a parametric heap structure supporting *find-min* in $O(\log n)$ time, insertion in $O(\log n \log\log\log n)$ time, and deletion in $O(\log n \log\log n)$ time. Whether these bounds for insertion or deletion can be improved is yet another open question.

The simplicity of the data structure in Section 2 is appealing and leads to additional open questions. Can a splay tree [13] be used in place of a balanced tree without affecting the bounds ? Is there a semi-dynamic version of this structure with an $O(\log n)$ amortized deletion time ? Is there an efficient version of the structure in which the items need not be ordered by key coefficient ?

We have studied how to implement broadcast scheduling policies such as LWF and RxW efficiently using kinetic heaps. Another interesting question to ask of such policies is how well they perform as compared to an optimum off-line schedule (computed with knowledge of the future), with respect to a criterion such as minimum total waiting time.

Acknowledgment

We thank Andrew Goldberg for insightful discussions.

References

[1] D. Aksoy and M. Franklin. Scheduling for large-scale on-demand data broadcasting. In *Proc. IEEE INFOCOM Conf., San Francisco, CA*, pages 651–659. IEEE, 1998.

[2] J. Basch *Kinetic Data Structures.* PhD thesis, Stanford University, June 1999.

[3] J. Basch, L. J. Guibas, and J. Hershberger. Data structures for mobile data. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 747–756, New Orleans, Louisiana, 5–7 January 1997.

[4] J. Bentley and J. Saxe Decomposable searching problems I: Static-to-dynamic transformation". *J. of algorithms*, 1:301–358, 1980.

[5] G. Brodal and R. Jacob. Dynamic planar convex hull with optimal query time and $O(\log n \cdot \log\log n)$ update time. In *Proc. 7th Scandinavian Workshop on Algorithm Theory*, Lecture Notes in Computer Science. Springer Verlag, Berlin, 2000.

[6] T. M. Chan Dynamic planar convex hull operations in near-logarithmic amortized time. In *Proc. 40th IEEE Annual Symposium on Foundations of Computer Science*, pages 92–99. IEEE, 1999.

[7] T. M. Chan Remarks on k-level algorithms in the plane. draft, 1999.

[8] B. Chazelle. On the convex layers of a planar set. *IEEE1 Transactions on Information Theory*, 31, 1985.

[9] B. V. Cherkassky, A. V. Goldberg, and C. Silverstein. Buckets, heaps, lists, and monotone priority queues. *SIAM J. Computing*, 28:1326–1346, 1999.

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

[11] J. Hershberger and S. Suri. Applications of a semidynamic convex hull algorithm. *BIT*, 32:249–267, 1992.

[12] M. H. Overmars and J. van Leeuwen Maintenance of configurations in the plane. *J. Comp. and Syst. Sci.*, 23:166–204, 1981.

[13] D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32:652–686, 1985.

[14] R. E. Tarjan. *Data Structures and Network Algorithms* CBMS 44. Society for Industrial and applied mathematics, Philadelphia, PA, 1983.

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

[16] R. E. Tarjan and C. J. Van Wyk. An $O(n \log\log n)$-time algorithm for triangulating a simple polygon. *Siam J. Computing*, 17(1):143–173, 1988.

[17] J. W. Wong. Broadcast delivery. *Proc. IEEE*, 76(12):1566–1577, 1988.