DIMACS Technical Report 2004-21
May 2004

A Framework for Reducing Comparisons in Heap Operations

by

Amr Elmasry
Computer Science Department, Alexandria University,
Alexandria, Egypt
elmasry@alexeng.edu.eg

ABSTRACT

We introduce a framework for reducing the number of comparisons performed in the deletion and minimum deletion operations for priority queues. In particular, we give a priority queue with constant cost per insertion and minimum finding, and logarithmic cost with at most $\log n + O(\log \log n)$ [1] comparisons per deletion and minimum deletion, improving over the bound of $2 \log n + O(1)$ comparisons for the binomial queues and the pairing heaps. We further improve this bound to at most $\log n + O(1)$ comparisons. We also give a priority queue that supports, in addition to the above operations, the decrease-key operation. This latter priority queue achieves, in the amortized sense, constant cost per insertion, minimum finding and decrease-key operations, and logarithmic cost with at most $1.44 \log n + O(\log \log n)$ comparisons per deletion and minimum deletion.

[1] $\log x$ equals $\max(\log_2 x, 1)$.

1 Introduction

One of the major research issues in the field of theoretical Computer Science is the comparison complexity of comparison-based problems. In this paper, we consider the priority queues that have constant insertion cost, with an attempt to reduce the number of comparisons involved in the delete-min operation. Binary heaps [20] are therefore excluded, following the fact that $\log \log n$ comparisons are necessary and sufficient per insertion [12]. Gonnet and Munro [12] (corrected by Carlsson [5]) also showed that $\log n + \log^* n + \Theta(1)$ comparisons are necessary and sufficient for deleting the minimum of a binary heap.

Several priority queues that achieve constant insertion cost, and logarithmic cost per delete and delete-min appeared in the literature. Examples of such priority queues that achieve these bounds, in the amortized sense [17], are the binomial queues [2, 18] and the pairing heaps [9, 13]. The same bounds can be achieved in the worst case with a special implementation of the binomial queues. If we allow the decrease-key operation, the Fibonacci heaps [10] and the thin heaps [14] achieve, in the amortized sense, constant cost per insert, find-min and decrease-key, and logarithmic cost per delete and delete-min. Other priority queues that achieve such bounds in the worst case are the relaxed heaps [7], and the priority queues in [3, 4, 14].

Among the priority queues that achieve constant insertion cost, the number of comparisons performed in the delete-min operation of the binomial queues and the pairing heaps is bounded by $2 \log n + O(1)$. The multiplicative constant involved in the logarithmic factor exceeds 2 for the other priority queues mentioned above.

Since our new heap structures use binomial queues as a building block, we review the operations of the binomial queues in the next section. In section 3, we give a general framework for structuring priority queues that may lead to the claimed reduction in the comparisons performed. We apply this framework in three different ways as follows. In Section 4, we give a structure that achieves constant cost per insert and find-min, and logarithmic cost with at most $\log n + O(\log \log n)$ comparisons per delete and delete-min. Though, we improve this bound in Section 5, the techniques in Section 4 are more general and can be used in a wider range of heap structures, as for example in the implicit binomial queues [6]. In Section 5, we achieve the better bound of at most $\log n + O(1)$ comparisons per delete and delete-min. As an application, we show in Section 6 that using this priority queue in the adaptive Heap-sort algorithm in [15] achieves a bound of at most $n \log \frac{I}{n} + O(n)$ comparisons, where I is the number of inversions in the input sequence. This is the first heap-based adaptive sorting algorithm optimal with respect to the number of comparisons. In Section 7, we give a priority queue that supports, in addition to the above operations, the decrease-key operation. This latter priority queue achieves, in the amortized sense, constant cost per insert, find-min and decrease-key, and logarithmic cost with at most $1.44 \log n + O(\log \log n)$ comparisons per delete and delete-min.

A similar question with respect to the comparisons required by the dictionary operations was answered with the affirmative to be $\log n + O(1)$ by Andersson and Lai [1]. Still, the trees of Andersson and Lai achieve this bound only in the amortized sense. The existence of such a worst-case bound is an open problem.

2 Binomial queues

A binomial tree [2, 18] of rank r is constructed recursively by making the root of a binomial tree of rank $r-1$ the leftmost child of the root of another binomial tree of rank $r-1$. A binomial tree of rank 0 is a single node. The following properties follow from the definition:

- The rank of an n-node (assume n is a power of 2) binomial tree is $\log n$.
- The root of a binomial tree of rank r has r sub-trees each of which is a binomial tree, having respective ranks $0, 1, \ldots, r-1$ from right to left.

A binomial queue is a set of binomial trees with at most one tree of any given rank. In addition, the value of every node is smaller than or equal to the values of its children.

Each binomial tree within a binomial queue is implemented using the binary representation. In such an implementation, every node has two pointers, one pointing to its left sibling and the other to its leftmost child. The sibling pointer of the leftmost child points to the rightmost child forming a circular list. Given a pointer to a node, both its rightmost and leftmost children can be accessed in constant time. The list of its children can be sequentially accessed right-to-left. To support the delete operation, each node includes a pointer to its parent. The roots of the binomial trees within a binomial queue are organized in a linked list right-to-left, which is referred to as the *root-list*.

Two binomial trees of the same rank can be merged in constant time, by making the root of the tree that has the larger value the leftmost child of the other root. The following operations are defined for binomial queues:

insert. The new element is added to the forest as a tree of rank 0, and successive merges are performed until there are no two trees of the same rank. A pointer to the node with the minimum value is maintained.

delete-min. The root with the smallest element is removed, thus leaving all the sub-trees of that element as independent trees. Trees of equal ranks are then merged until no two trees of the same rank remain. The new minimum among the current tree roots is then found and maintained.

delete. The key of the node to be deleted is repeatedly swapped with its parents up to the root of its tree. A delete-min is then performed to delete this node.

For an n-node binomial queue, the worst-case cost per insert, delete-min and delete is $O(\log n)$. The amortized bound on the number of comparisons per insert is 2, and per delete-min is $2 \log n$. To see that this bound is tight, consider a binomial queue with n one less than a power of 2. Consider an alternating sequence of a delete-min that is followed by an insert, such that the minimum element is always the root of the tree with the largest rank. For every delete-min in such a case, we need $\lfloor \log n \rfloor$ comparisons for merging trees with equal ranks and $\lfloor \log n \rfloor$ comparisons to find the new minimum.

3 The framework

For the binomial queues, there are two major procedures that contribute to the multiplicative factor of 2 in the bound on the number of comparisons for the delete-min operation. The first is merging the trees with equal ranks, and the second is maintaining the new minimum element. The basic idea of our framework is to reduce the number of comparisons involved in finding the new minimum, after the deletion of the current minimum. This is achieved by composing our priority queues from the following main components:

3.1 The lower layer

This layer is basically the main priority queue, implemented as a forest of heap-ordered structures, with a pointer to the structure that has the overall minimum value maintained by the upper layer.

3.2 The upper layer

This layer is implemented as another priority queue that only contains the minimum element of each of the lower layer structures. Its main purpose is to maintain a pointer to the structure, among the structures of the lower layer, which has the overall minimum value.

3.3 The reservoir

The basic idea is to keep one structure in the reservoir and treat it in a special way. Whenever an element is deleted from the lower layer, as a result of a delete or a delete-min operation, we borrow an element from the reservoir. Using this borrowed element, the construction of the structure that lost an element is readjusted gaining the same properties as before the deletion.

4 Our first priority queue

The main priority queue resembles the lower layer, and is implemented as a binomial queue. The upper layer is implemented as another binomial queue that only contains the roots of the binomial trees of the lower layer. The minimum element of this upper layer is, therefore, the overall minimum element. The size of the upper layer is $O(\log n)$ and the delete-min requires $O(\log \log n)$ comparisons for this layer. The challenge is how to maintain the upper layer and how to efficiently implement the priority queue operations at the lower layer to reduce the work to be done at the upper layer, achieving the claimed bounds. If the delete-min operation is implemented the same way as that of the standard binomial queues, there would be a logarithmic number of new roots that need to be inserted at the upper layer. Hence, a new implementation of the delete-min operation, that does not alter the current roots of the trees, is introduced. We maintain one tree in the reservoir, not necessarily binomial. Each time we perform a delete or a delete-min, a node is borrowed from the reservoir. Next, we show how different priority queue operations are handled.

delete-min. To borrow a node from the tree of the reservoir, we detach the rightmost child of its root, making the children of the detached node the rightmost children of the root in the same order. Note that this can be done with constant cost and involves no comparisons. If there are no nodes left in the reservoir, a binomial tree is moved from the lower layer to the reservoir. A crucial constraint is that the rank of the tree being moved to the reservoir is not the largest among the trees of the lower layer. The corresponding node to the root of this tree is deleted from the upper layer. A special case is when there is only one tree at the lower layer. In such case, we split that tree by cutting the leftmost sub-tree of its root, and move this sub-tree to the reservoir.

Using the pointer from the upper layer to the tree that has the root with the minimum value at the lower layer, this root is deleted. The node that is borrowed from the reservoir is incrementally merged with the sub-trees of the deleted root right-to-left (sub-trees with smaller ranks first). In other words, every sub-tree is merged with the tree resulting from the merging of the sub-trees to its right, in an incremental fashion. The merging is done similar to that of the binomial trees; the root of the tree that has the larger value becomes the leftmost child of the other root. This results in a new binomial tree with the same structure as before the deletion. (Since merging two binomial trees of rank i results in a binomial tree of rank $i + 1$, successively merging two binomial trees of rank 0 and then merging the resulting tree with another binomial tree of rank 1 and so on up to $r - 1$ results in a binomial tree of rank r.) The number of comparisons performed by this incremental merging step is at most $\log n$. At the upper layer, the new root of this tree is inserted and the corresponding node to the old root is deleted with an extra $O(\log \log n)$ cost.

If the root of the tree of the reservoir is the node with the minimum value, this root is removed and an incremental merging procedure is performed on its children right-to-left. Starting with a binomial tree of rank r in the reservoir, the only guarantee is that the number of children of the root of this tree at any moment is at most r. This follows from Lemma 1 below. Since during the lifespan of the current tree of the reservoir there is another binomial tree at the lower layer whose rank is at least r, the number of comparisons involved for this case is at most $\log n$.

lemma 1 *Starting with a binomial tree of rank r in the reservoir, the number of children of the root of the reservoir at any moment is at most r.*

Proof. Consider any node x of the reservoir. Let A_x be the set of ancestors of x, including x. Let cls_x be the count of the left siblings of x, including x. Let d_x be $\sum_{y \in A_x} cls_y$. A property of binomial trees implies that d_x is initially at most r. When the root of the reservoir is deleted, the value of d_x drops by 1. The right-to-left incremental merging of the children of the root may increase d_x by at most 1. When the rightmost child of the root is detached from the reservoir, the value of d_x may only decrease. Hence, d_x will not exceed r. The lemma follows by noting that the number of children of the root equals the value of d_x of its rightmost child. □

delete. If the deleted node is a node of the reservoir, its sub-trees are merged right-to-left. Otherwise, the node that is borrowed from the reservoir is incrementally merged with the sub-trees of the deleted node right-to-left. The key of the root of the resulting sub-tree is repeatedly compared with its parents and swapped if necessary. The number of comparisons involved in this procedure is at most $\log n$. If the value of the root of the tree that involves the deletion (except for the reservoir) changes, the new root is inserted at the upper layer and the corresponding node to the old root is deleted.

insert. We first show how to perform the insertion in constant amortized cost. Similar to the binomial queues, the insert is performed by adding a new node of rank 0 to the lower layer. If there are two binomial trees of the same rank r in the queue, the two trees are to be merged forming a binomial tree of rank $r + 1$. In the amortized sense, this accounts for a constant number of merges. As a result of the merge operations, some of the roots of the trees at the lower layer are linked to other roots, and should be deleted from the upper layer. We can only afford to spend a constant time with each of these deletions. Hence, a method of lazy deletions is applied, where a node to-be-deleted is only marked until we have time for bulk deletions. If any of these marked nodes is again promoted as a root at the lower layer (as a result of a delete or delete-min), its mark at the upper layer is removed. When the number of marked nodes reaches a constant factor of the nodes of the upper layer (say half), the upper layer is rebuilt in linear time getting rid of the marked nodes. The cost of the rebuilding is amortized over the merges that took place at the lower layer, for a constant cost per merge. What makes the scheme of lazy deletions work is the fact that none of the marked nodes could possibly become the minimum node of the upper layer, since the upper layer must have an unmarked node smaller than each marked node.

For the worst-case constant bound, we cannot afford to perform all the necessary merges at once. Instead, we do a constant number of merges with each insertion. Hence, merges will be left partially completed to pick up the work on the next operation. To facilitate this, we maintain a logarithmic number of pointers to the merges in progress and their structures, kept as a stack of pointers. With every insertion we make progress on the merge of the two smallest trees with the same rank. Similar to our amortized solution, we need to do the insertion and the lazy deletion (marking) at the upper layer. From the amortized analysis of binomial queues it is clear that we must do at least two comparisons per insertion, performing the merges. See [6] for a similar treatment. A nice result of [6] implies that the number of pointers can be reduced to $\log^* n$ if three units of work are done with every insertion, instead of two.

Still, we cannot afford to spend a linear time rebuilding the upper layer when the number of marked nodes reaches a constant factor of the total number of nodes. Instead, we use a technique similar to the global rebuilding technique in [16]. When the number of unmarked nodes of the upper layer goes below $\frac{m}{c}$, where m is the number of nodes at the upper layer and $c \geq 2$ is some constant, we start rebuilding the whole layer. The work is distributed over the next operations. We still use and update our original upper layer, but in parallel we also build a new heap structure. If a node to-be-marked also exists in the new structure it has to be marked there, as well. Whenever a new node is inserted in the current upper layer we insert it in the new structure, as well. Whenever we mark a node for deletion or insert a new node in the current upper layer, we copy two of the unmarked nodes from the current structure to the new structure. It follows that within the next at most $\frac{m}{2c}$ operations, all the unmarked nodes must have been copied to the new structure. At this point, we can dismiss the current structure and use the new one instead. At this point, the new structure will have at least half the nodes unmarked. Since $c \geq 2$, we are only busy constructing at most one new structure at a time.

Theorem 1 *Our first priority queue achieves constant cost per insert and find-min, and logarithmic cost with at most* $\log n + O(\log \log n)$ *comparisons per delete and delete-min.*

The bound on the number of comparisons for the delete and delete-min operations can be further reduced as follows. Instead of having two layers we may have several layers. The delete-min operation on the second layer is implemented the same way as that of the first layer. In each layer, other than the highest layer, a pointer for the minimum element at that layer is maintained from the next higher layer. Except for the highest layer, we need a constant of one for the logarithmic factor as a bound for the number of comparisons performed by the delete-min applied at this layer. Therefore, the bound on the delete and delete-min operations is at most $\log n + \log \log n + \ldots + 2 \log^{(k)} n + O(1)$ comparisons, where $\log^{(k)}$ is the logarithm taken k times and k is a constant representing the number of layers. An insertion of a new element would result in a constant amortized number of insertions and markings per layer. The number of layers k should be constant to achieve the constant amortized insertion cost.

5 Our second priority queue

The lower layer is divided into the following two components:

The main heap

The main heap is implemented as a binomial queue, with an extra pointer accompanying the root of each tree. The pointer of each root points to the root with the smallest value among the roots to its right, including itself. It follows that the pointer of the root with the largest rank points to the minimum value of the main heap. We call these pointers the prefix-minimum pointers (*pm* pointers). These pointers resemble the upper layer in our general framework.

The auxiliary heap

The auxiliary heap is implemented as another binomial queue, with one pointer pointing to the root with the minimum value. All the insertions are performed in this heap, until trees are moved later to the main heap.

5.1 Achieving the amortized bounds

Next, we show how heap operations are performed for this priority queue to achieve the claimed bounds in the amortized sense.

insert. A newcomer is inserted in the auxiliary heap in the same way insertions are performed in binomial queues. The minimum of the auxiliary heap is maintained. In the amortized sense, this requires 2 comparisons.

delete-min. Let r be the rank of the tree that has the minimum value. Three possibilities popup depending on whether the node with the minimum value is:

- the minimum of the main heap,
- the minimum of the auxiliary heap, or
- the root of the reservoir.

For the first two cases, we start by borrowing a node from the reservoir as before. If there were no nodes in the reservoir we move a tree from the main heap to the reservoir. (One possibility is to move the tree with the second highest rank from the main heap, and update the pm pointer of the tree with the highest rank. If there is only one tree in the two heaps altogether, we split that tree by cutting the leftmost sub-tree of its root, and move this sub-tree to the reservoir. Other special cases should also be treated.) After the minimum node is removed, the node that is borrowed from the reservoir is incrementally merged with the sub-trees of the deleted root right-to-left. The number of comparisons performed by this incremental merging step is r.

If the minimum was from the main heap, the pm pointer of the tree of the deleted minimum and those of the trees with larger ranks need to be updated. This is accomplished by proceeding right-to-left; the pm pointer of a given node x is updated to point to the smaller of the value of x and the value of the node pointed to by the pm pointer of the right sibling of x. The number of comparisons performed by this procedure is at most $\lg n - r + 1$.

If the minimum was from the auxiliary heap, the tree of the deleted minimum and the trees with larger ranks are moved to the main heap. Trees with the same ranks in the main heap are then merged, until no two trees with the same rank remain. The cost of these merges is charged to the insertions performed. The pm pointers of the trees of the main heap whose ranks are higher or equal to r need to be updated using at most $\lg n - r + 1$ comparisons, as explained above. The new minimum of the auxiliary heap is found in $i - 1$ comparisons, where i is the number of trees remaining in the auxiliary heap. These $i - 1$ comparisons are also charged to the insertions performed, as stated in Lemma 2 below.

lemma 2 *The amortized bound on the number of comparisons performed per insertion is* $2 + o(1)$*.*

Proof: Similar to the standard binomial queues, we maintain one credit per tree. Whenever a node is inserted two credits are enough to cover the costs of the merges performed in both the auxiliary and the main heaps. When the root with the minimum value is deleted from the auxiliary heap, the cost of finding the new minimum of the auxiliary heap is charged to the nodes that have just been moved to the main heap. If such actual cost is $i - 1$, this means that at least 2^i nodes have been moved to the main heap. Since every node is moved once, this accounts for $(i - 1)/2^i = o(1)$ extra cost per insertion (if i is a constant, this constant cost is charged to the delete-min). □

If the minimum is the root of the reservoir, the same procedure as that implemented for our first priority queue is used.

delete. If the deleted node is a node of the reservoir, its sub-trees are merged right-to-left. Otherwise, the node that is borrowed from the reservoir is incrementally merged with the sub-trees of the deleted node right-to-left. The key of the root of the resulting sub-tree is repeatedly compared with its parents and swapped if necessary. If the value of the root of the tree that involves the deletion (except for the reservoir) changes, the procedure continues to update the minimum pointers as if a delete-min is performed on the root of this tree.

5.2 Achieving the worst-case bounds

First, we show how to achieve a worst-case bound of $\log n + O(1)$ comparisons per delete and delete-min and constant amortized cost per insertion. Let h be the highest rank of the main heap. We impose the rule that the highest rank of the auxiliary heap is at most $\lceil \frac{h}{2} \rceil - 1$. Insertions are done in the auxiliary heap the same way as that of the standard binomial queues. The only difference pops up when a tree of rank $\lceil \frac{h}{2} \rceil$ is produced as a result of a merge. In such case, this tree is immediately moved to the main heap. The way we implement the heap operations insures that there is no tree in the main heap with smaller rank that this tree. This tree is repeatedly merged with a tree of the main heap if they have the same rank. If the root of the originally moved tree wins all these possible comparisons and is still a root of a tree, the pm pointers of its left siblings are updated. This requires at most $\lfloor \frac{h}{2} \rfloor + 1$ comparisons to finish the merges and update the pm pointers. There should have been $2^{\lceil \frac{h}{2} \rceil}$ insertions since another tree has been moved from the auxiliary heap to the main heap. This accounts for an extra $o(1)$ amortized cost per insertion. Deleting a node from either the main heap or the reservoir is implemented as before with at most $\log n + O(1)$ comparisons. Deletions from the auxiliary heap are performed the same way as that of the standard binomial queues. This accounts for at most $2(\lceil \frac{h}{2} \rceil - 1)$ comparisons, which is at most $\log n$.

To achieve the worst-case constant insertion cost, we cannot afford to perform all the necessary merges at once. Instead, we do a constant number of merges with each insertion, similar to the insertions in our first priority queue. If a tree is moved from the auxiliary heap to the main heap we perform one unit of work in the main heap (either a merge or an update to a pm pointer) with every insertion. From the above amortized analysis, it is clear that we must do at least $2 + o(1)$ comparisons per insertion, performing the merges and fixing the pm pointers. We need to be careful that the pm pointers are not necessarily correct. To find the minimum of the main heap, we need to compare the value pointed to by the pm pointer of the leftmost root and the value of the root of the tree that has been moved from the auxiliary heap to the main heap. Another issue is the possibility that some trees of the auxiliary heap will have the same ranks. As a result, the number of trees of the auxiliary heap is not bounded by $\lceil \frac{h}{2} \rceil$ anymore, but by twice that much. To overcome this deficiency, the highest allowable rank of the auxiliary heap should be $\lceil \frac{h}{3} \rceil - 1$, instead. This reestablishes the bound on the number of comparisons for deleting a node from the auxiliary heap to be less than h, which is at most $\log n$. On the other hand, after a tree is moved from the auxiliary heap to the main heap, it requires at most $\lfloor \frac{2h}{3} \rfloor + 1$ comparisons to perform the merges and update the pm pointers of the main heap. In this case, there should have been at least $2^{\lceil \frac{h}{3} \rceil - 1}$ insertions since another tree has been moved from the auxiliary heap to the main heap. (Immediately after the previous tree had been moved to the main heap, the highest rank of the auxiliary heap must have been at most $\lceil \frac{h}{3} \rceil - 2$, with no two trees with the same rank; for at most $\sum_{i=0}^{\lceil \frac{h}{3} \rceil - 2} 2^i < 2^{\lceil \frac{h}{3} \rceil - 1}$ nodes. To produce a tree with rank $\lceil \frac{h}{3} \rceil$, at least $2^{\lceil \frac{h}{3} \rceil - 1}$ more insertions are needed.) Hence, performing $o(1)$ comparisons per insertion to maintain the pm pointers guarantees that, at a time, the pm pointers are correct except, possibly, for only the rightmost tree that may have been moved from the auxiliary heap.

Theorem 2 *Our second priority queue achieves constant cost per insert and find-min, and logarithmic cost with at most* $\log n + O(1)$ *comparisons per delete and delete-min.*

6 Application - Adaptive Heap-sort

An adaptive sorting algorithm is a sorting algorithm that benefits from the presortedness in the input sequence and sorts faster. There are plenty of adaptive sorting algorithms in the literature, and many measures defining how the input sequence is presorted. One of the simplest adaptive sorting algorithms introduced in the literature is the adaptive Heap-sort algorithm [15].

The first step of the algorithm is to build a Cartesian tree from the input sequence. Given a sequence $X =< x_1, \ldots, x_n >$, the corresponding Cartesian tree [19] is a binary tree with root $x_i = \min(x_1, \ldots, x_n)$. Its left sub-tree is the Cartesian tree for $< x_1, \ldots, x_{i-1} >$ and its right sub-tree is the Cartesian tree for $< x_{i+1}, \ldots, x_n >$. A Cartesian tree can be constructed in linear time [11]. After building a Cartesian tree, the adaptive Heap-sort proceeds by inserting the smallest element of the Cartesian tree in a heap. During the iterative step, the minimum of the heap is deleted and printed, and the children of this element in the Cartesian tree are inserted in the heap. The total work done by the algorithm is, therefore, n insertions and n minimum deletions plus the linear work involving building and querying the Cartesian tree. Levcopoulos and Petersson [15] showed that the number of elements of the heap at step i is not greater than $\lfloor \frac{||Cross(x_i)||}{2} \rfloor + 2$, where x_i is the smallest element of the heap at step i and

$$Cross(x_i) = \{j | 1 \leq j \leq n \text{ and } \min(x_j, x_{j+1}) < x_i < \max(x_j, x_{j+1})\}.$$

They [15] suggested using a binary heap, and showed that their algorithm runs in $O(n \log \frac{Osc(X)}{n})$, where

$$Osc(X) = \sum_{i=1}^{n} ||Cross(x_i)||.$$

They also showed that, using a binary heap, the number of comparisons performed by the n insertions and the n minimum deletions is at most $2.5 n \log n$. Using the fact that $Osc(X) \leq 4 \cdot Inv(X)$ [15], it follows that adaptive Heap-sort runs in $O(n \log \frac{Inv(X)}{n})$, where $Inv(X)$ is the number of inversions in X.

$$Inv(X) = ||\{(i,j) \mid 1 \leq i < j \leq n \text{ and } x_i > x_j\}||.$$

Using our second priority queue instead of the binary heap, we achieve a bound of $\sum_{i=1}^{n} \log ||Cross(x_i)|| + O(n)$ comparisons which is at most $n \log \frac{Osc(X)}{n} + O(n)$, and hence a bound of $n \log \frac{Inv(X)}{n} + O(n)$ comparisons, which is optimal with respect to the number of comparisons. Other adaptive sorting algorithms that achieve the same bound [8] are either based on Insertion-sort or Merge-sort.

7 Our third priority queue

We introduce the notion of an F-queue that is used as our basic structure.

F-queues

An F-tree is recursively defined as follows. An F-tree of rank 0 is a single node. An F-tree of rank r consists of a root node that has r or $r-1$ sub-trees. These sub-trees, from right to left, are F-trees with consecutive ranks $0, 1, \ldots, r-1$ or $0, 1, \ldots, r-2$. We call an F-tree whose rank is r a heavy F-tree if its root has r sub-trees, and light if its root has $r-1$ sub-trees. Each node of an F-tree is implemented with three pointers that point to its left sibling, right sibling and leftmost child. The left pointer of the leftmost child points to the parent instead of its nonexistent left sibling.

lemma 3 *The number of descendants of an F-tree of rank r is at least* Φ^{r-1}, *where* $\Phi = \frac{1+\sqrt{5}}{2}$ *is the golden ratio.*

Proof: Let T_r be the size of an F-tree of rank r. It follows from the definitions that $T_0 = 1, T_1 \geq 1$, and $T_r \geq 1 + \sum_{i=0}^{r-2} T_i$ for $r \geq 2$. Consider the Fibonacci numbers defined as $F_0 = F_1 = 1$, and $F_r = F_{r-1} + F_{r-2}$ for $r \geq 2$. It follows by induction that $T_r \geq F_r$ for all r. The inequality $F_r \geq \Phi^{r-1}$ is well known. It follows that $r < 1.44 \log T_r$. □

An F-queue is a forest of F-trees with the additional constraint that the value of every node is smaller than or equal to the value of its children. The main F-trees are all heavy, a condition that is not necessarily true for sub-trees. Note the similarity between the definition of the F-queues and the thin heaps [14]. We define the following operations on F-trees:

split. A heavy F-tree of rank r can be split into two F-trees by cutting the leftmost sub-tree of the root from the rest of the tree. This leftmost sub-tree will form an F-tree of rank $r-1$ (heavy or light), while the rest of the tree forms a light F-tree of rank r. No comparisons are performed in this operation.

merge. Two F-trees of the same rank r can be merged, in constant time and one comparison, resulting in an F-tree of rank $r+1$. Let x be the root of the tree that has the larger value.

1. If the two trees are heavy: Link x as the leftmost child of the other root, forming a heavy F-tree.
2. If the two trees are light: Decrement the rank of the tree of x to $r-1$, converting it to a heavy F-tree. Link x as the leftmost child of the other root, forming a light F-tree.
3. If one tree is light and the other is heavy:
   (a) If the tree of x is the light tree: Link x as the leftmost child of the other root, forming a heavy F-tree.
   (b) If the tree of x is the heavy tree: Split the tree of x to two F-trees of ranks $r-1$ and r, as above. Link these two trees, the one of rank $r-1$ first, as the leftmost children of the other root, forming a heavy F-tree.

The lower layer, as well as the upper layer, is implemented as an F-queue. A pointer to the node with the minimum value of the upper F-queue is also maintained. Next, we show how different heap operations are handled.

delete-min. Given a pointer from the upper layer to the tree that has the root with the minimum value, the delete-min is implemented as follows. After this minimum node is removed, we perform an incremental merging procedure on its children, from right to left. We may think about the single node that is the rightmost child of this root as a light tree of rank 1. It follows that the resulting tree will still be an F-tree of the same rank. If the resulting F-tree is light, make it heavy by decrementing its rank. At the upper layer, the new root of this tree is inserted and the corresponding node to the old root is deleted. We then proceed by merging any two F-trees that have the same rank, until there are no two trees of the same rank exist. This is done in a similar way to that of the Fibonacci heaps and the thin heaps. With each of these merges, a corresponding node at the upper layer is marked for lazy deletion. The amortized number of comparisons performed by this operation is at most $1.44 \log n + O(\log \log n)$.

decrease-key. Let x be the node whose value is decreased. If x is a root of a tree at the lower layer, a decrease-key operation is performed on the corresponding node at the upper layer and the procedure terminates. Otherwise, the sub-tree of x is cut and made a new tree. If this tree is light, make it heavy by decrementing its rank. The node x is then inserted at the upper layer. Consider the position of x before the cut, and let y be its parent. The left siblings of x are traversed from right to left, until a heavy sub-tree is encountered (if at all such a sub-tree exists). The rank of the root of each of the traversed light sub-trees is decremented, making these sub-trees heavy. If we encountered a heavy sub-tree, it is split into two sub-trees as mentioned above, and the procedure terminates. If all the left siblings of x were light and y was heavy, the procedure terminates and y becomes light. If the left siblings of x were light and y was also light, the sub-tree of y is cut and made a new tree. The rank of y is adjusted, making its tree heavy. The node y is then inserted at the upper layer. The procedure continues considering the left siblings of y, and the process is repeated until either no structural problem exists, or until we reach a root of a tree. The amortized cost of this operation is constant [14].

delete. To delete a node, its value is decreased to become the smallest among the nodes of the heap. It is then deleted by applying a delete-min operation.

Theorem 3 *Our third priority queue achieves, in the amortized sense, constant cost per insert, find-min and decrease-key, and logarithmic cost with at most* $1.44 \log n + O(\log \log n)$ *comparisons per delete and delete-min.*

References

[1] A. Andersson and T. W. Lai. *Fast updating of well-balanced trees.* 2nd SWAT, LNCS 447 (1990), 111-121.

[2] M. Brown. *Implementation and analysis of binomial queue algorithms.* SIAM J. Comput. 7 (1978), 298-319.

[3] G. Brodal. *Fast meldable priority queues.* 4th WADS, LNCS 955 (1995), 282-290.

[4] G. Brodal. *Worst-case efficient priority queues.* 7th ACM SODA (1996), 52-58.

[5] S. Carlsson. *An optimal algorithm for deleting the root of a heap.* Inf. Proc. Lett. 37 (1991), 117-120.

[6] S. Carlsson, J. Munro and P. Poblete. *An implicit binomial queue with constant insertion time.* 1st SWAT, LNCS 318 (1988), 1-13.

[7] J. Driscoll, H. Gabow, R. Shairman and R. Tarjan. *Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation.* Comm. ACM 31(11) (1988), 1343-1354.

[8] A. Elmasry and M. Fredman, *Adaptive sorting and the information theoretic lower bound.* 20th STACS, LNCS 2607 (2003), 654-662.

[9] M. Fredman, R. Sedgewick, D. Sleator and R. Tarjan. *The pairing heap: a new form of self_adjusting heap.* Algorithmica 1(1) (1986), 111-129.

[10] M. Fredman and R. Tarjan. *Fibonacci heaps and their uses in improved network optimization algorithms.* J. ACM 34(3) (1987), 596-615.

[11] H. Gabow, J. Bentley and R. Tarjan. *Scaling and related techniques for computational geometry.* 16th ACM STOC (1984), 135-143.

[12] G. Gonnet and J. Munro. *Heaps on heaps.* SIAM J. Comput. 15(4) (1986), 964-971.

[13] J. Iacono. *Improved upper bounds for pairing heaps.* 7th SWAT, LNCS 1851 (2000), 32-45.

[14] H. Kaplan and R. Tarjan. *New heap data structures.* TR-597-99, Princeton University (1999).

[15] C. Levcopoulos and O. Petersson. *Adaptive Heapsort.* J. Alg. 14 (1993), 395-413.

[16] M. Overmars and J. van Leeuwen. *Worst-case optimal insertion and deletion methods for decomposable searching problems.* Inf. Proc. Lett. 12 (1981), 168-173.

[17] R. Tarjan. *Amortized computational complexity.* SIAM J. Alg. Disc. Meth. 6 (1985), 306-318.

[18] J. Vuillemin. *A data structure for manipulating priority queues.* Comm. ACM 21(4) (1978), 309-314.

[19] J. Vuillemin. *A unifying look at data structures.* Comm. ACM 23 (1980), 229-237.

[20] J. Williams. *Algorithm 232, Heapsort.* Comm. ACM 7(6) (1964), 347-348.