Fat Heaps Without Regular Counters*

Amr Elmasry and Jyrki Katajainen

Department of Computer Science, University of Copenhagen, Denmark
{elmasry, jyrki}@diku.dk

Abstract. We introduce a variant of fat heaps that does not rely on regular counters, and still achieves the optimal worst-case bounds: $O(1)$ for *find-min*, *insert* and *decrease*, and $O(\lg n)$ for *delete* and *delete-min*. Our variant is simpler to explain, more efficient, and easier to implement. Experimental results suggest that our implementation is superior to structures, like run-relaxed heaps, that achieve the same worst-case bounds, and competitive to structures, like Fibonacci heaps, that achieve the same bounds in the amortized sense.

1 Introduction

Motivation. A numeral system is a notation for representing numbers using symbols—digits—in a consistent manner. Operations such as modifying a specified digit must obey the rules guarding the numeral system. These rules vary from simple rules as, for example, allowing only two symbols in the digit set (binary system) to more involved rules that constrain the relationship between digits. There is a connection between numeral systems and data-structural design [5, 14]. Relating the number of objects of particular type in the data structure to the value of a digit in the representation of a number may allow the performance of the operations on such objects to be improved. Caveat that complex constraints might complicate the structure and make the implementation impractical.

Many data structures—such as worst-case-efficient finger trees [11], fat heaps [12, 13], fast meldable priority queues [1], and worst-case-optimal meldable priority queues [2]—rely on numeral systems, and, as introduced, the application of numeral systems seems to be essential. In a recent paper [9] we introduced the strictly-regular numeral system, and showed how to use it to improve the constant factors in the worst-case performance of the priority queue described in [1]. We advocated that numeral systems are important when developing worst-case-efficient data structures, but we also warned against their overuse. To make the latter point clear, we show in this paper that fat heaps can be implemented, within the same asymptotic performance, without involved numeral systems.

Related work. Historically, the most significant priority queues are binary heaps [15] and Fibonacci heaps [10]. Nevertheless, binary heaps are not optimal

* © 2012 Springer-Verlag. This is the authors' version of the work. The original publication is available at www.springerlink.com.

Table 1. The asymptotic performance of some worst-case-efficient priority queues. Here n (and m) denotes the number of elements currently stored in the data structure(s) manipulated ($m \leq n$).

Data structure	*find-min*	*insert*	*decrease*	*delete*	*meld*
Binomial queues [8, 14]	$O(1)$	$O(1)$	$O(\lg n)$	$O(\lg n)$	$O(\lg m)$
Run-relaxed heaps [6]	$O(1)$	$O(1)$	$O(1)$	$O(\lg n)$	$O(\lg m)$
Fat heaps [12, 13, this paper]	$O(1)$	$O(1)$	$O(1)$	$O(\lg n)$	$O(\lg m)$
Fast meldable priority queues [1]	$O(1)$	$O(1)$	$O(\lg n)$	$O(\lg n)$	$O(1)$
Optimal meldable priority queues [2]	$O(1)$	$O(1)$	$O(1)$	$O(\lg n)$	$O(1)$

for all operations, and the bounds derived for Fibonacci heaps are amortized. Therefore, many other priority-queue structures have been developed that are efficient either in the amortized or in the worst-case sense. In Table 1 we summarize the asymptotic performance of the worst-case-efficient priority queues that are relevant for the present study.

Since the *meld* operation is seldom needed in applications, and as the structure described in [2] is complicated and impractical, run-relaxed heaps and fat heaps are the structures of choice when worst-case constant-time *decrease* has to be supported. Elsewhere it has been stated that the worst-case-efficient priority queues do not perform well in practice (see, for example, [4]). Hence, the challenge is to make such priority-queue structures practically efficient.

As introduced [12, 13], fat heaps heavily rely on numeral systems. The rules governing the used numeral systems build on the notion of regular counters. An extended regular ternary counter is used to maintain the trees in their data structure and an extended regular binary counter to maintain the nodes potentially violating the heap order. Brodal [2] showed how to implement a regular binary counter (a system where the digit set has three symbols), using a structure that he calls a guide. To support constant-time decrements of any digit, he suggested to couple two such guides back to back, and hence deal with six symbols. To implement an extended regular system, Kaplan et al. [12] suggested using forward pointers. In the latter solution the digit set has four symbols in the binary case and five symbols in the ternary case.

On the other hand, run-relaxed heaps do not require regular counters; two simple counters, which allow digits to take any value and keep the sum of digits bounded, are enough. The expense is that the violation-reduction transformations are more complicated for run-relaxed heaps than their counterparts for fat heaps; namely fat heaps do not need run transformations (for details, see [6]).

Contribution. We introduce a variant of fat heaps that achieves the same asymptotic bounds, while not relying on regular counters and still using simple violation-reduction transformations (not involving run transformations). Thus, the new variant is conceptually simpler and easier to explain. In the basic implementation (Section 2) the model of computation used is a word RAM with an instruction set allowing normal array and list operations. Also, we apply several

implementation enhancements that make our priority queue easy to implement and practically efficient compared to the best implementations of the related data structures. In the enhanced implementation (Section 3) we assume the availability of an instruction that determines the position of the most-significant 1-bit in a word; this is a built-in hardware instruction in many modern computers including the one where we run our experiments (Section 4).

2 Fat Heaps Simplified

Structure. A fat heap [12, 13] is a forest of trinomial trees. A *trinomial tree* of *rank* r is composed of three trinomial trees of rank $r-1$, two of which are subtrees of the root of the third tree. A node of rank r then has two children of rank 0, rank 1, ..., rank $r-1$. A single node has rank 0. It follows that a trinomial tree of rank r has $n = 3^r$ nodes and its root has $2r = 2\log_3 n$ children.

Similar to relaxed heaps [6], the trees are *heap ordered*, i.e. an element associated with a node is not smaller than that associated with its parent, except for some special nodes that may, but not necessarily, violate the heap order; these nodes are called *potential violation nodes*. A fat heap storing n elements consists of $O(\lg n)$ trinomial trees, which may contain $O(\lg n)$ violation nodes.

The children of a node are maintained in non-decreasing rank order to form a *child list*. The roots of the trees are linked in a similar manner to form a *root list*. Each node has two pointers to its siblings to keep it on one of these lists, and a pointer to its last child. The right-sibling pointer of the last child points to the parent. Additionally, each node stores its rank and a *root-indicator bit* that distinguishes roots from other nodes.

The key issue in a fat heap is how to keep track of the trees and the violation nodes, and how to make sure that their count never gets too high. In the original articles [12, 13] numeral systems were used for both of these purposes. Our point here is to show that much simpler tool, a resizable array collecting duplicates at each rank, is enough in both cases.

Tree reductions. If a fat heap stores n elements, the largest trinomial tree is of rank at most $\lfloor \log_3 n \rfloor$. When the number of trees becomes larger than $2\lfloor \log_3 n \rfloor + 2$, there must be at least three trees having the same rank. Three trinomial trees of rank r can be joined to form a trinomial tree of rank $r+1$ by letting the root that stores the smallest element be the new root and making the other two trees the last two subtrees of the new root. This operation, called a *tree reduction*, requires two element comparisons and a few pointer updates.

Tree inventory. In order to detect at which ranks we have more than two trees, we maintain a *tree inventory* which consists of two parts: 1) a resizable array where the rth entry stores the number of trees of rank r as well as a pointer to the first root of rank r, and 2) a linked list on the entries of the array indicating the ranks at which there are more than two trees. Using this linked list and the array, three trees to be joined can be easily located. See Section 3 for a different

treatment that uses a bit vector, instead of the linked list, to indicate the ranks at which there are more than two trees.

Violation inventory. For the violation nodes we use a similar arrangement. The so-called *violation inventory* also has two parts: 1) a resizable array where the rth entry records the number of violation nodes of rank r as well as a pointer to one violation of rank r, and 2) a doubly-linked list on the entries of the array indicating the ranks at which a violation reduction is possible (details are given below). See Section 3 for a different treatment using a bit vector instead.

To keep track of the violation nodes that have the same rank, we maintain a doubly-linked list for each rank. We do this indirectly so that the list contains objects each of which has a pointer to a priority-queue node, which has a pointer back to this list object. We need this back pointer when a node becomes non-violating and has to be removed from its list in constant time. Observe that in their original form [12, 13] fat heaps do not need this back pointer, as the number of violation nodes per rank is a constant; this overhead is due to our simplified construction. On the other hand, similar to our treatment, run-relaxed heaps do need these back pointers. See Section 3 for a different treatment that uses arena-based memory management for the intermediate list objects; in effect, the storage overhead at the priority-queue nodes can be reduced.

Storage requirements. Each priority-queue node has an element, a rank, a root-indicator bit, and four pointers: two sibling pointers, a last-child pointer, and a pointer to the violation inventory. The amount of storage used by the tree and violation inventories is $O(\lg n)$.

Violation reductions. A *violation reduction* at rank r makes two rank-r violation nodes non-violating at the expense of possibly introducing one new violation node at rank $r+1$. The main task is to keep the number of violation nodes logarithmic. An implementation relying on a numeral system [12, 13] achieves this by guaranteeing that the number of violation nodes per rank is at most a constant. To avoid run transformations [6], each of the two rank-r violation nodes involved in a reduction must either be one of the last two children of its parent or have a non-violating rank-$(r+1)$ sibling. The aforementioned restriction is also guaranteed when using a numeral system (the reductions are performed on violation nodes of rank r whose count corresponds to a 2, which is followed by a 0 or 1 indicating that at most one violation node of rank $r+1$ exists).

We do not restrict the number of violation nodes per rank. Instead, similar to run-relaxed heaps, we maintain the number of violation nodes below $\lfloor \log_3 n \rfloor + 1$. Our first observation is that when the number of violation nodes becomes larger than this, we can find a rank r such that there exist at least two violation nodes of rank r and at most one violation node of rank $r+1$; at such rank a violation reduction is possible. This observation follows from the fact that a root cannot be violating, and hence no violation nodes with the highest rank exist. Our second observation is that we can efficiently keep track of ranks where such violation

reductions are possible, using an array of double pointers and a doubly-linked list in which viable ranks appear in arbitrary order.

When a non-violating rank-r node becomes violating or vice versa, it is a routine matter to accordingly update the violation inventory in constant time. We also need to update the doubly-linked list having the ranks for which violation reductions are possible. This is done by first updating the number of violation nodes at rank r, then checking the number of violation nodes at ranks $r-1$, r, $r+1$ within the corresponding array entries. Next, we update the double pointers to maintain the doubly-linked list accordingly. The details are simple and are left for the reader. See Section 3 for a different treatment that uses bit operations to detect at which rank such a reduction would be possible.

Assume that we have two violation nodes u and v of rank r and at most one violation node at rank $r+1$. If u is not one of the last two children of its parent, we know that among the two siblings of u of rank $r+1$ at most one can be violating. Let x be a non-violating sibling of u of rank $r+1$, and let y be one of the last two children of x. We make u one of the last two children of x by swapping the subtree rooted at u with the subtree rooted at y. Similarly, we ensure that v is one of the last two children of its parent. Assume without loss of generality that the element at the parent of u is not larger than that at the parent of v. If u and v are not siblings, we swap the subtree rooted at u and the subtree rooted at the rank-r sibling of v to make u and v children of the same parent. Note that in this process no new violation nodes are created. Hereafter, we cut the three subtrees rooted at u, v, and their parent. We then join them in one tree of rank $r+1$ and attach that tree in the place of the subtree rooted at the parent. Unless the root of the resulting tree is a tree root, it is made violating and added to the violation inventory. Note that the last two children of the root of the resulting tree are now non-violating. In total, each violation reduction requires at most three element comparisons and a few pointer updates.

Operations. When the reduction techniques are available, it is quite straightforward to implement the priority-queue operations.

find-min. Let all the other operations maintain a pointer, called the *minimum pointer*, to a node that stores the current minimum; this node can be a root or a violation node. Hence, this operation can be easily supported in constant time. See Section 3 for a slower version of this operation.

insert. The given node that already stores an element is added to the tree inventory as a trinomial tree of rank 0. If the number of trees exceeds the threshold, their number is reduced by executing a single tree reduction. Since the node may contain the current minimum, a further check is needed to ensure that the minimum pointer is up to date. In total, a constant amount of work is done including at most three element comparisons.

decrease. After making the element replacement, the accessed node is made violating (without any checks), and one violation reduction is executed if necessary. The new value is compared with the minimum, and the minimum pointer is updated if necessary. Hence, this operation also involves a constant

amount of work including at most four element comparisons (three for a violation reduction and one for a minimum-pointer update).

delete. A simple way of accomplishing this operation is to make the node to be deleted a root by bubbling it up (repeatedly swapping the node with its current parent) until a root is met. Since we are only storing parent pointers for the last children, a parent is reached by repeatedly traversing right-sibling pointers. When the nodes on the path to the root are moved one level down, their rank in the violation inventory is to be updated as well. Thereafter, this root can be removed and its subtrees can be inserted into the tree inventory. Accordingly, the number of trees in the tree inventory may increase by at most $2\log_3 n + O(1)$, and several tree reductions may be necessary. The number of element comparisons involved in such tree reductions is at most $2\log_3 n + O(1)$. If the deleted element is the minimum, a new one is found by scanning all the roots and violation nodes, and the minimum pointer is updated; this requires at most another $3\log_3 n + O(1)$ element comparisons. As a result of a deletion n goes down by 1. This may result in the number of violation nodes exceeding the threshold, and a violation reduction is to be performed. In total, the number of element comparisons performed by *delete* is at most $5\log_3 n + O(1) \approx 3.16 \lg n$, and the work done is $O(\lg n)$. See Section 3 for a different implementation that relies on borrowing a node to replace the deleted node.

meld. Assume that the queues to be melded contain m and n elements, and $m \leq n$. There are two tasks to be done: to merge the tree inventories and to merge the violation inventories. The first task is done by moving the trees, one by one, from the smaller tree inventory to the larger. Thereafter, the number of trees is reduced by repeated tree reductions until the number of trees is within the allowable threshold. The amount of work involved is proportional to the number of element comparisons performed, which is at most $2\log_3 m + O(1)$. For the second task the larger of the two resizable arrays is kept as the basis of the violation inventory of the combined priority queue, and the other array is disposed after all the violation nodes are moved from there to the other inventory. Each time a violation node is moved from one inventory to another it may be necessary to execute a violation reduction. The amount of work involved is proportional to the number of element comparisons performed, which is at most $3\log_3 m + O(1)$. In total, the number of element comparisons performed in the *meld* operation is at most $5\log_3 m + O(1) \approx 3.16 \lg m$, and the work done is $O(\lg m)$.

3 Implementation Enhancements

Genericity. We relied on policy-based design as recommended, for example, in [4]. The implementation accepted several parameters, making it possible to change policies in use whenever desired. The parameters passed included the type of elements stored, comparison function used in element comparisons, allocator used for memory management, nodes used for encapsulating the elements, tree

inventory used for keeping track of the trees, and violation inventory used for keeping track of the violation nodes.

Duplicate grouping. In principle, we used almost identical arrangements when implementing the two inventories. Both included a preallocated array that stored pointers to the beginning of each duplicate list (roots or violation nodes of the same rank), and two bit vectors: the *occupancy vector* that indicated the ranks where there are at least one entity, and the *threshold vector* that indicated the ranks where the number of entities is or exceeds the threshold. Since there is a logarithmic number of ranks, each of these bit vectors could be maintained in a single word. There were four main differences between the two inventories: 1) What was the threshold making a reduction possible (two vs. three)? 2) How were reductions accomplished (tree reduction vs. violation reduction)? 3) How were the duplicates linked? 4) What was stored at the priority-queue nodes?

The implemented tree inventory kept the duplicates grouped, but the roots were not fully sorted with respect to their ranks. Since the sibling pointers of the roots were unused, we reused these pointers when linking the duplicates.

Arena. As for the doubly-linked lists that hold objects of the same ranks (see Section 2), the implemented violation inventory used an extra preallocated array that was used to store these objects. We call this array an *arena*, and the resulting violation inventory *arena-based*. Yet another bit vector was used to keep track of the free places on the arena. Each entry of the arena stored a pointer to the priority-queue node it represents. In addition, each such entry stored two indexes to a next and a previous entry of the same rank within the arena, if any. Now the back pointer from the priority-queue node could be replaced by the index of the corresponding entry in the arena. The index stored is called a *violation index*; if a node is non-violating, its violation index had some fixed sentinel value. The good news is that the violation index, the root-indicator bit, and the rank each can be stored in one byte, and hence all three can be stored in one word. Other than the data element, including the sibling and child pointers, this sums up to four words per node. In addition to saving storage and avoiding pointer manipulation, two important other benefits are that we improve cache efficiency and avoid additional memory allocation and deallocation.

Bit hacks. To perform a tree reduction the threshold vector of the tree inventory is consulted to pick any rank where the threshold is reached (at least three trees of that rank exist). This is realized by relying on an instruction that determines the position of an arbitrary 1-bit in a word. For a violation reduction the situation is more restricted. Actually, a rank having at least two violation nodes does not indicate that a violation reduction is possible at that rank. We are looking for a rank r where the threshold is reached (at least two violation nodes of that rank exist) and in the meantime rank $r + 1$ has at most one violation node. This is realized by relying on an instruction that determines the position of the most-significant 1-bit in the threshold vector of the violation inventory. The use

of this special instruction is crucial, as the position of the most-significant 1-bit always corresponds to a rank where a violation reduction is possible.

Borrow-based delete. Following [3, 6] we did not implement *delete* by bubbling elements up (see Section 2). Instead, we used a local strategy where we borrowed a node from the smallest tree after splitting it up repeatedly, and then reconstructed the subtree that lost its root by starting with the borrowed node and repeatedly joining the subtrees of the deleted node. The new root of the reconstructed tree becomes a violation node, unless it is a root of an entire tree. For random deletions this would reduce the expected cost of *delete* to a constant. To efficiently locate the smallest tree we again used our bit-manipulation tools, by determining the position of the least-significant 1-bit in the occupancy vector of the tree inventory.

Event-driven reductions. We chose to perform reductions in an aggressive manner. Instead of maintaining exact counts for the number of trees and violation nodes and repeatedly checking these counts, we opted for performing reductions whenever a threshold is possibly exceeded. A tree reduction is performed by every *insert* if possible. A violation reduction is performed by every *decrease* and *delete* if possible. During *meld* a tree reduction is performed after every tree move, and a violation reduction is performed after every violation-node move, if either is possible. It is worth mentioning that although our implementation of *delete* would result in increasing the number of trees because of the borrowing of the root of the smallest tree, we do not need to perform a tree reduction in *delete*. Performing a tree reduction only in *insert* would still guarantee that the number of trees is at most $2\lfloor\log_3 n\rfloor + 2$. In fact, we can prove by induction a stronger bound; after each operation the number of trees is at most $2\lfloor\log_3 n\rfloor + 2 - 2r'$, where r' is the rank of the smallest tree.

Slow minimum finding. In some applications it is advantageous to keep *find-min* fast, while in other applications it is advantageous to support *find-min* in logarithmic time and remove the burden of updating the minimum pointer from other operations. To facilitate this we implemented a decorator that could transform any priority queue supporting logarithmic *find-min* to one that supports constant-time *find-min*; the only overhead caused by this is that each *delete-min* has to find the minimum element before each element removal instead of updating the minimum pointer after the operation. Hence, we can support both variants in a flexible manner. In our experiments we used the logarithmic (slow) version of *find-min* for all the priority queues under investigation.

4 Experiments

Competitors. We implemented fat heaps as discussed in the previous section. Our programs have been made part of the CPH STL (`www.cphstl.dk`), from where we also picked two competitors for comparison: a run-relaxed weak queue

[7] (binary version of a run-relaxed heap [6]), which has the same worst-case behaviour as a fat heap, and a Fibonacci heap [10], which has the same bounds (*meld* is even amortized constant) as a fat heap but in the amortized sense. Both of these existing implementations were highly tuned. The implementation of a run-relaxed weak queue relies on bit-compaction techniques to keep the tree and violation inventories small and efficient. Such data structure is documented in [4]. The variant of the Fibonacci heap that we used was extremely lazy: *insert*, *decrease*, and *delete* operations only append some nodes to the root list and leave all the actual work for the forthcoming *find-min* operations (which have to consolidate the root list). Because of this laziness we do not expect that any of the worst-case-efficient priority queues can beat this lazy variant for other than *find-min* and *delete-min* operations. However, it is still interesting to see what the cost of worst-case efficiency is in practice.

Alternatives. We implemented the array-based tree inventory that keeps the roots of the same rank on doubly-linked lists (as proposed in [6]). As an alternative we tested the inventory relying on a special regular counter (as proposed in [7]). Such counter should support digit injections and ejections at one end, and satellite-data updates at any position. A stack and an array would be needed for its implementation. In our experiments both types of inventories worked well. Of the tested structures our fat heap uses the array-based solution, whereas the run-relaxed weak queue uses the numeral-system-based solution.

We tried three different violation-inventory realizations. The first was almost identical to the array-based tree inventory. Instead of storing roots the inventory stored violation nodes, and in the nodes space for two additional pointers was reserved. Interestingly, in spite of the larger memory footprint of the nodes, this version worked well in practice. As we expect that the amount of extra space used will be significant in some applications, we also considered the compact violation inventory described in [4] and the arena-based solution described in Section 3. Of the tested structures our fat heap uses the arena-based solution, whereas the run-relaxed weak queue uses the more complex bit-packing techniques of [4].

Lines of code. By looking at the code available at the repository of the CPH STL we can quantitatively verify that a fat heap is simpler than other structures that achieve the same worst-case performance (see Table 2). Even if the lines-of-code (LOC) metric may be considered questionable, the numbers extracted clearly suggest that the transformations needed for performing a violation reduction are simpler for fat heaps than for run-relaxed weak queues. The LOC counts also indicate that the array-based tree inventory is simpler than the numeral-system-based tree inventory, and that a worst-case-efficient priority queue requires more code than a Fibonacci heap.

Environment. In our tests we used a laptop with the following configuration:
CPU: Intel Core 2 Duo (model T5600 at 1.83GHz)
Memory size: 1 GB
Cache size: 2 MB

Table 2. LOC counts for some priority-queue realizations in the CPH STL. All comments, lines only having a single parenthesis, debugging code, and assertions are excluded from these counts.

Data structure	**Component**	**LOC**
Fat heap	kernel	136
	node	199
	tree inventory	116
	violation inventory	301
	Total	752
Run-relaxed weak queue	kernel	166
	node	253
	tree inventory	238
	violation inventory	550
	Total	1 207
Fibonacci heap	kernel	192
	node	104
	Total	296

Operating system: Ubuntu 11.10 (Linux kernel 3.0.0-12-generic)
Compiler: `g++` (`gcc` version 4.6.1)
Compiler options: `-DNDEBUG -Wall -std=c++0x -pedantic -x c++ -O3`

Experimental set-up. In all experiments the input was of type `long long`. We considered tests involving the following operation sequences:

insert **test.** Starting from an empty data structure perform n *insert* operations. The elements were given in decreasing order.

decrease **test.** Starting from a data structure of size n, created by n *insert* operations followed by one *find-min* operation, perform n *decrease* operations. The value of each element was arbitrarily decreased once (the new value does not affect the outcomes) and decreases were performed in random order.

delete **test.** Starting from a data structure of size n, created by n *insert* operations followed by one *find-min* operation, perform n *delete* operations. The elements were deleted in random order.

delete-min **test.** Starting from a data structure of size n, created by n *insert* operations, perform n *delete-min* operations. The elements were inserted into the data structure in random order.

We measured the work per operation by dividing the total work used for the whole sequence by n. We repeated each individual experiment $10^7/n$ times and took the average. The work used for all initializations is excluded from the measurement results. When analysing the performance of different data structures on these operation sequences we considered two performance indicators: running time and number of element comparisons.

Table 3. Performance of fat heaps versus their competitors.

(a) Average running time (in microseconds) per operation.

Test	**Data structure**	$n = 10\,000$	$n = 100\,000$	$n = 1\,000\,000$
insert	Fat heap	0.12	0.12	0.12
	Run-relaxed weak queue	0.16	0.16	0.15
	Fibonacci heap	0.10	0.09	0.09
decrease	Fat heap	0.28	0.52	0.78
	Run-relaxed weak queue	0.48	0.73	1.02
	Fibonacci heap	0.01	0.13	0.22
delete	Fat heap	0.33	0.36	0.42
	Run-relaxed weak queue	0.58	0.63	0.68
	Fibonacci heap	0.02	0.05	0.07
delete-min	Fat heap	0.48	1.08	2.07
	Run-relaxed weak queue	0.52	1.10	2.19
	Fibonacci heap	0.71	1.41	2.85

(b) Average number of element comparisons performed per operation.

Test	**Data structure**	$n = 10\,000$	$n = 100\,000$	$n = 1\,000\,000$
insert	Fat heap	0.99	0.99	0.99
	Run-relaxed weak queue	0.99	0.99	0.99
	Fibonacci heap	0	0	0
decrease	Fat heap	2.96	2.99	2.99
	Run-relaxed weak queue	1.99	1.99	1.99
	Fibonacci heap	0	0	0
delete	Fat heap	3.04	3.12	3.17
	Run-relaxed weak queue	2.40	2.44	2.44
	Fibonacci heap	0	0	0
delete-min	Fat heap	20.3	26.6	32.8
	Run-relaxed weak queue	16.3	21.2	26.2
	Fibonacci heap	16.2	21.2	26.2

Outcomes. The results in Table 3 indicate that Fibonacci heaps are performing better than both fat heaps and run-relaxed weak queues as for *insert*, *decrease*, and *delete*. The reason is that the lazy version of Fibonacci heaps postpones the actual work to *find-min* and *delete-min*. Although fat heaps require more element comparisons than run-relaxed weak queues per *decrease*, our fat-heap implementation is faster; this is due to the simpler transformations, the avoidance of numeral-system complications, and other implementation enhancements. For *delete-min* the other two heaps are superior to Fibonacci heaps. The reason is that here Fibonacci heaps pay for the laziness of the other operations.

We also performed some experiments with intermixed operation sequences. One such sequence considered involved rounds of k *decrease* operations followed by one *find-min*, one *delete*, and one *insert* operation. By varying the parameter

k, once k was larger than a small constant (5 for our experiments), the effect of decreases was dominating. This particular experiment, together with some other similar experiments, gave us a consistent picture of the behaviour of the three data structures: Fibonacci heaps were the fastest, run-relaxed weak queues the slowest, and fat heaps landed there between. Some divergences are possible, but in general the behaviour of these data structures is predictable; the structures are more sensitive to the sequence of operations than to the data values.

5 Conclusion

Progress. First, we thought that Brodal's guides [2] would be needed to implement the extended regular counters. Later, we found a simpler way of implementing those counters. Then we observed that the extended regular counters are actually not needed, and the inventories could be implemented using the non-extended regular counters [5]; the key is that, for digit d_i, the general $\texttt{--}d_i$ operation is not needed; it is enough to support the operation **if** $d_i > 0$: $\texttt{--}d_i$, which the non-extended regular counter can handle efficiently. Finally, we ended up with the solution reported in this paper; fat heaps can be implemented with similar counters as those used in the implementation of run-relaxed heaps.

Summary. No knowledge of numeral systems is needed to understand the functioning of our version of fat heaps. Thus, our description is conceptually simpler than the original descriptions. A straightforward implementation of simplified fat heaps would require more space. However, for an arena-based inventory the memory overhead is only one byte per element. Due to the alignment enforced by most modern compilers, the space requirements of the two versions—the original and ours—are the same, i.e. $4n + O(\lg n)$ words plus the space used by the n elements themselves. Also, the time complexity of all priority-queue operations is the same for the two versions. Compared to run-relaxed heaps the transformations needed for reducing the number of potential violation nodes are simpler for fat heaps; our lines-of-code counts witness this clearly. As far as we know our implementation of fat heaps is the first fully functional, publicly available implementation. Our experiments showed that the actual runtime performance of fat heaps is good. At the same time, fat heaps provide strict worst-case guarantees for all priority-queue operations. However, in some cases, binomial trees used by run-relaxed heaps are better than trinomial trees used by fat heaps; and in typical applications data structures that are efficient in the average-case sense or in the amortized sense can be faster.

Source code

The programs used in the experiments are available via the home page of the CPH STL (`http://cphstl.dk/`) in the form of a PDF document and a `tar` file.

References

1. Brodal, G.S.: Fast meldable priority queues. In: 4th International Workshop on Algorithms and Data Structures. LNCS, vol. 955, pp. 282–290. Springer-Verlag (1995)
2. Brodal, G.S.: Worst-case efficient priority queues. In: 7th Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 52–58. ACM/SIAM (1996)
3. Brown, M.R.: Implementation and analysis of binomial queue algorithms. SIAM Journal on Computing 7(3), 298–319 (1978)
4. Bruun, A., Edelkamp, S., Katajainen, J., Rasmussen, J.: Policy-based benchmarking of weak heaps and their relatives. In: 9th International Symposium on Experimental Algorithms. LNCS, vol. 6049, pp. 424–435. Springer-Verlag (2010)
5. Clancy, M.J., Knuth, D.E.: A programming and problem-solving seminar. Technical Report STAN-CS-77-606, Department of Computer Science, Stanford University (1977)
6. Driscoll, J.R., Gabow, H.N., Shrairman, R., Tarjan, R.E.: Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. Communications of the ACM 31(11), 1343–1354 (1988)
7. Elmasry, A., Jensen, C., Katajainen, J.: Relaxed weak queues: An alternative to run-relaxed heaps. CPH STL Report 2005-2, Department of Computer Science, University of Copenhagen (2005)
8. Elmasry, A., Jensen, C., Katajainen, J.: Multipartite priority queues. ACM Transactions on Algorithms 5(1), 14:1–14:19 (2008)
9. Elmasry, A., Jensen, C., Katajainen, J.: Strictly-regular number system and data structures. In: 12th Scandinavian Symposium and Workshops on Algorithm Theory. LNCS, vol. 6139, pp. 26–37. Springer-Verlag (2010)
10. Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. Journal of the ACM 34(3), 596–615 (1987)
11. Guibas, L.J., McCreight, E.M., Plass, M.F., Roberts, J.R.: A new representation for linear lists. In: 9th Annual ACM Symposium on Theory of Computing. pp. 49–60. ACM (1977)
12. Kaplan, H., Shafrir, N., Tarjan, R.E.: Meldable heaps and Boolean union-find. In: 34th Annual ACM Symposium on Theory of Computing. pp. 573–582. ACM (2002)
13. Kaplan, H., Tarjan, R.E.: New heap data structures. Technical Report TR-597-99, Department of Computer Science, Princeton University (1999)
14. Vuillemin, J.: A data structure for manipulating priority queues. Communications of the ACM 21(4), 309–315 (1978)
15. Williams, J.W.J.: Algorithm 232: Heapsort. Communications of the ACM 7(6), 347–348 (1964)