A Back-to-Basics Empirical Study of Priority Queues*

Daniel H. Larkin† Siddhartha Sen‡ Robert E. Tarjan§

November 11, 2013

Abstract

The theory community has proposed several new heap variants in the recent past which have remained largely untested experimentally. We take the field back to the drawing board, with straightforward implementations of both classic and novel structures using only standard, well-known optimizations. We study the behavior of each structure on a variety of inputs, including artificial workloads, workloads generated by running algorithms on real map data, and workloads from a discrete event simulator used in recent systems networking research. We provide observations about which characteristics are most correlated to performance. For example, we find that the L1 cache miss rate appears to be strongly correlated with wallclock time. We also provide observations about how the input sequence affects the relative performance of the different heap variants. For example, we show (both theoretically and in practice) that certain random insertion-deletion sequences are degenerate and can lead to misleading results. Overall, our findings suggest that while the conventional wisdom holds in some cases, it is sorely mistaken in others.

1 Introduction

The priority queue is a widely used abstract data structure. Many theoretical variants and implementations support a varied set of operations with differing guarantees. We restrict our attention to the following base set of commonly used operations:

- Insert (Q, x, k) — insert item x with key k into heap Q and return a handle $\bar{x}$
- DeleteMin (Q) — remove the item of minimum key from heap Q and return its corresponding key k
- DecreaseKey $(Q, \bar{x}, k')$ — given a handle $\bar{x}$, change the key of item x belonging to heap Q to be k', where k' is guaranteed to be less than the original key k

It has long been known that either Insert or DeleteMin must take $\Omega(\log n)$ time due to the classic lower bound for sorting [23], but that the other operations can be done in $\mathcal{O}(1)$ time. In practice, the worst-case of $\log n$ is often not encountered or can be treated as a constant, and for this reason simpler structures with logarithmic bounds have traditionally been favored over more complicated, constant-time alternatives. In light of recent developments in the theory community [2, 5, 11, 12, 19] and the outdated nature of the most widely cited experimental studies on priority queues [24, 26, 30], we aim to revisit this area and reevaluate the state of the art. More recent studies [3, 11, 13] have been narrow in focus with respect to the implementations considered (e.g., comparing a single new heap to a few classical ones), the workloads tested (e.g., using a few synthetic tests), and the metrics collected (e.g., measuring wallclock time and element comparisons). In addition to the normal metric of wallclock time, we have collected additional metrics such as branching and caching statistics. Our goal is to identify experimentally verified trends which can provide guidance to future experimentalists and theorists alike. We stress that this is not the final word on the subject, but merely another line in the continuing dialogue.

In implementing the various heap structures, we take a different approach from the existing algorithm engineering literature, in that we do not perform any algorithm engineering. That is, our implementations are intentionally straightforward from their respective descriptions in the original papers. The lack of considerable tweaking and algorithm engineering in this study is, we believe, an example of naïveté as a virtue. We expect that this would accurately reflect the strategy of a practitioner seeking to make initial comparisons between different heap variants. As a sanity check, we also compare our implementations with a state-of-the-art, well-engineered implementation often cited in the literature.

Our high-level findings can be summarized as follows. We find that wallclock time is highly correlated

*Research at Princeton University partially supported by NSF grant CCF-0832797 and a Google PhD Fellowship.

†Princeton University, Department of Computer Science. Email: dhlarkin@cs.princeton.edu.

‡Microsoft Research SVC. Email: sisen@microsoft.com

§Princeton University, Department of Computer Science and Microsoft Research SVC. Email: ret@cs.princeton.edu.

with the cache miss rate, especially in the L1 cache. High-level theoretical design decisions—such as whether to use an array-based structure or a pointer-based one—have a significant impact on caching, and which decisions fare best is dependent on the specific workload. For example, Fibonacci heaps sometimes outperform implicit d-ary heaps, in contradiction to conventional wisdom. Even a well-engineered implementation like Sanders' sequence heap [29] can be bested by our untuned implementations if the workload favors a different method.

Beyond caching behavior, those heaps with the simplest implementations tend to perform very well. It is not always the case that a theoretically superior or simpler structure lends itself to simpler code in practice. Pairing heaps dominate Fibonacci heaps across the board, but interestingly, recent theoretical simplifications to Fibonacci heaps tend to do worse than the original structure.

Furthermore we found that a widely-used benchmarking workload is degenerate in a certain sense. As the sequence of operations progresses, the distribution of keys in the heap becomes very skewed towards large keys, contradicting the premise that the heap contains a uniform distribution of keys. This can be shown both theoretically and in practice.

Our complete results are detailed in Sections 4 and 5. We first describe the heap variants we implemented in Section 2, and then discuss our experimental methodology and the various workloads we tested in Section 3. We conclude in Section 6 with some remarks.

2 Heap Variants

Aiming to be broad, but not necessarily comprehensive, this study includes both traditional heap variants and new variants which have not previously undergone much experimental scrutiny. We have implemented the following structures, listed here in order of program length: implicit d-ary heaps, pairing heaps, Fibonacci heaps, binomial queues, explicit d-ary heaps, rank-pairing heaps, quake heaps, violation heaps, rank-relaxed weak queues, and strict Fibonacci heaps. Table 1 lists the logical lines of code; in our experience, this order corresponded exactly to perceived programming difficulty. There are several other heap variants which may be worth investigating, but which have not been included in this study. Among those not included are the 2-3 heap [31], thin/thick heaps [22], and the buffer heap [6].

Table 1: Programming effort

Heap variant	Logical lines of code (lloc)
implicit simple	184
pairing	186
implicit	194
Fibonacci	282
binomial	317
explicit	319
rank-pairing	376
quake	383
violation	481
rank-relaxed weak	638
strict Fibonacci	1009

Williams' binary heap [33] is the textbook example of a priority queue. Lauded for its simplicity and taught in undergraduate computer science courses across the world, it is likely the most widely used variant today. Storing a complete binary tree whose nodes obey the heap order gives a very rigid structure; indeed, the heap supports all operations in worst-case $\Theta(\log n)$ time. The tree can be stored explicitly using heap-allocated nodes and pointers, or it can be encoded implicitly as a level-order traversal in an array. We refer to these variations as *explicit* and *implicit* heaps respectively. The implicit heap carries a small caveat, such that in order to support DecreaseKey efficiently, we must rely on a level of indirection: encoding the tree's structure as an array of node pointers and storing the current index of a node's pointer in the node itself (allowing us to return the node pointer to the client as $\bar{x}$). This study includes two versions of implicit heaps—one that supports DecreaseKey through this indirection, and one that doesn't. We refer to the latter as the *implicit-simple* heap.

Explicit and implicit heaps can be generalized beyond the binary case to have any fixed branching factor d. We refer to these heaps collectively as d-ary heaps; this study examines the cases where $d = 2, 4, 8, 16$. To distinguish between versions with different branching factors, we label the heaps in this fashion: *implicit-2*, *explicit-4*, *implicit-simple-16*, and so forth.

Beyond the d-ary heaps, all other heap variants are primarily pointer-based structures, though some make use of small auxiliary arrays. All are conceptual successors to Vuillemin's binomial queue [32]. Originally developed to support efficient melding (which takes linear time in d-ary heaps), we have included it in our study due to its simplicity. The binomial queue stores a forest of perfect, heap-ordered binomial trees of unique rank. This uniqueness is maintained by linking trees of equal rank such that the root with lesser key becomes the new parent. To support deletion of a node, each of its children is made into a new root, and the resulting forest is then processed to restore the unique-rank invariant. This can lead to a fair amount of structural

rearrangement, but the code to do so is rather simple. Key decreases are handled as in d-ary heaps by sifting upwards. Like d-ary heaps, binomial queues support all operations in worst-case $\Theta(\log n)$ time.

Most other heap variants can be viewed as some sort of relaxation of the binomial queue, with the chronologically first one being the Fibonacci heap [17]. The Fibonacci heap achieves amortized $\mathcal{O}(1)$-time Insert and DecreaseKey by only linking after deletions and allowing some imperfections in the binomial trees. The imperfections are generated by key decreases: instead of sifting, a node is cut from its parent as soon as it loses a second child, and then made into a new root. This can lead to a series of upwardly cascading cuts. The violation heap [12] and rank-pairing heaps [19] can be viewed as further relaxations of the Fibonacci heap. The rank-pairing heaps allow rank differences greater than one, and propagate ranks instead of cascading cuts so that at most one cut is made per DecreaseKey. Two rank rules were proposed by the authors, leading to our implementations being labeled *rank-pairing-t1* and *rank-pairing-t2*. The violation heap also propagates ranks instead, only considering rank differences in the two most significant children of a node. It allows two trees of each rank and utilizes a three-way linking method. The pairing heap [16] is essentially a self-adjusting, single-tree version of the Fibonacci heap, where ranks are not stored explicitly, and linking is done eagerly. Its amortized complexity is still an open question, though it has been shown that DecreaseKey requires $\Omega(\log \log n)$ time if all other operations are $\mathcal{O}(\log n)$ [15]. Two different amortization arguments can be used to prove either $\mathcal{O}(1)$ and $\mathcal{O}(\log n)$ bounds for Insert and DecreaseKey respectively [20] or $\mathcal{O}\left(2^{2\sqrt{\log \log n}}\right)$ for both operations [27]. It remains an open question to prove an $o(\log n)$ bound for DecreaseKey simultaneously with $\mathcal{O}(1)$-time Insert. All three relaxations are intended to be in some way simpler than Fibonacci heaps, with the hope that this makes them faster in practice.

The strict Fibonacci heap [2] on the other hand, intends to match the Fibonacci time bounds in the worst case, rather than in an amortized sense. This leads to a fair amount of extra code to manage structural imperfections somewhat lazily. Rank-relaxed weak queues [11] are essentially a tweaked version of rank-relaxed heaps, with an emphasis on minimizing key comparisons. They mark nodes as potentially violating after a DecreaseKey operation and clean them up lazily. Quake heaps [5] are a departure from the Fibonacci model, but are still vaguely reminiscent of binomial queues. A forest of uniquely-ranked tournament trees is maintained. Subtrees may be missing, but the number of nodes at a given height decays exponentially in the height, a property guaranteed through a set of global counters and a global rebuilding process triggered after deletions. There are multiple implementation strategies mentioned in the original paper, but only the one that was fully detailed (the full tournament representation) has been implemented here. It is possible that the other implementations would be more efficient.

3 Experimental Design and Workloads

Our codebase is written primarily in C99 and is available online for inspection, modification, and further development [1]. As we stated earlier, our implementations are intentionally straightforward from their respective descriptions in the original papers, or use only the most basic, well-known optimizations for the more studied structures. Further optimization is left to the compiler (`gcc -O4`) so as not to unfairly bias toward one variant or another.

Keys are 64-bit unsigned integers (`uint64_t`), while the items themselves are 32-bit unsigned integers (`uint32_t`). In most cases, the key actually consists of a 32-bit key in the high-order bits and the item identifier in the low-order bits, in order to break ties during comparisons.

We experimented with different memory allocation schemes using our own simple fixed-size memory pool implementation. This abstraction layer allowed us to allocate all memory eagerly using a single `malloc`, allocate lazily by doubling when space fills, or allocate completely on the fly using a `malloc` for each Insert. In our experiments, the memory allocation scheme made very little difference regardless of heap variant, indicating that this layer of optimization was superfluous. Thus, all the results in this paper use the eager strategy.

The workloads we tested are described in the subsections below. These include workloads generated by code sourced (with modifications) from DIMACS implementation challenges [9, 10], as well as workloads generated by a packet-level network simulator [7]. All experiments use trace-based simulation. More specifically, a workload is generated once using a reference heap and the sequence of operations and values is recorded in a trace file. This trace file can then be executed against each of many drivers—one for each heap variant included in the study—as well as a dummy driver that simply parses the trace file but does not execute any heap operations. The dummy driver captures the overhead of the simulation and its collected metrics are subtracted from those of the other drivers before any comparisons are done. Wallclock time is measured by the driver itself. For purposes of timing, each execution

of a trace file is run for a minimum of five iterations and two seconds of wallclock time (whichever takes longer), and the time is averaged over all iterations. Other metrics are collected over the course of a single iteration using `cachegrind` [4], a cache and branch-prediction profiler. The profiler simulates actual machine parameters and does not vary between executions, providing accurate measurements that are isolated from other system processes. We have used it to collect dynamic instruction and branching counts as well as reads, writes and misses for both the L1 and L2 caches. Additionally, `cachegrind` allows for simulating branch prediction in a basic model (that does not correspond exactly to the real machines); we have collected this misprediction count as well.

All experiments were run on a high-performance computing cluster in Princeton consisting of Dell PowerEdge SC1435 nodes with dual AMD Opteron 2212 processors (dual-core, 2.0GHz, 64KB L1 cache and 1MB L2 cache per core) and 8GB of RAM (DDR2-667). The machines ran Springdale/PUIAS Linux (a Red-Hat Enterprise clone) with kernel version 2.6.32. All executions remained in-core.

3.1 Artificial randomized workloads. The first and least controversial workload we consider is sorting sequences of n uniformly random integers. This translates to n random insertions followed by n deletions in the trace files.

The next type of sequence intermixes insertions and deletions, but in a very structured way which turns out to be degenerate. It is a very natural sequence to test, and due to its presence in the DIMACS test set, we worry that its use in benchmarks may be more widespread than one might hope of a broken test. The sequence begins with n random insertions as in the sorting case. It is then followed by cn repetitions of the following: one random insertion followed by one deletion. It is not hard to show that the evolving distribution of keys remaining in the heap is far from uniform.

LEMMA 3.1. *After the initial n insertions and cn iterations of insert-delete, the items remaining in the heap consist of the n largest keys inserted thus far. The next item inserted has roughly a $c/c+1$ probability of becoming the new minimum.*

Proof. For the purpose of this analysis, we consider the inserted keys to be reals distributed uniformly at random in the range $[0, 1]$, rather than 32-bit integers. The pattern of operations leaves the n largest keys inserted thus far in the heap, as the following simple inductive argument shows. Initially, n keys are inserted; being the only keys thus far, they are trivially the largest. Then, each iteration consists of a single insertion followed by a minimum deletion. Since there are $n + 1$ keys in the heap after the insertion, and the minimum is deleted, the remaining n keys are the largest thus far.

We can view the random variables of all keys inserted thus far to be the collection $X_1, \ldots, X_{(c+1)n}$, and the current minimum in the heap to be the $(cn + 1)^{th}$ order statistic, $X_{(cn+1)}$. The expectation of this variable is well-known: $\mathbf{E}\left[X_{(cn+1)}\right] = {cn+1}/{(c+1)n} \approx {c}/{c+1}$. From this we deduce that the probability p of the next inserted key becoming the new minimum is roughly ${c}/{c+1}$.

As c grows, the most recent insertion becomes exceedingly likely to be the next deleted item. In other words, the behavior of the queue becomes increasingly stack-like as the sequence lengthens. On the other hand, if we introduce DECREASEKEY operations to the sequence, we can ameliorate the degeneracy. This brings us to our third type of artificial sequence. We again build an initial heap of size n with random insertions. We then perform cn repetitions of the following: one random insertion, k key decreases on random nodes, and one deletion. We also consider two cases for the k key decreases. In the first, we decrease the key to some random number between its current value and the minimum. In the second, we decrease it so that it becomes the new minimum. We refer to these options as "middle" and "min", respectively.

In both the insertion-deletion workloads and the key-decrease workloads we consider $c \in \{1, 32, 1024\}$, while in the key-decrease workloads we also consider $k \in \{1, 32, 1024\}$.

3.2 More realistic workloads. Of our remaining workloads, some are still artificial in the sense that they are generated by running real algorithms on artificial inputs, but others make use of real inputs.

The first two of these are Dijkstra's algorithm for single source shortest paths and the Nagamochi-Ibaraki algorithm for the min-cut problem. We run both algorithms against well-structured or randomly generated graphs. Dijkstra's algorithm in particular is run on several classes of graphs, including some which guarantee a DECREASEKEY operation for each edge. Additionally, we run Dijkstra's algorithm on real road networks of different portions of the United States.

Our final set of trace files is generated from the `htsim` packet-level network simulator [7], written by the authors of the multipath TCP (MPTCP) protocol. The simulator models arbitrary networks using pipes (that add delays) and queues (with fixed processing capacity and finite buffers), and implements both TCP

and MPTCP. One of these workloads is based on real traffic traces from the VL2 network [18].

4 Results

The results reveal a more nuanced truth than that which has been traditionally accepted. It is not true that implicit-4 heaps are optimal for all workloads, nor is it true that Fibonacci heaps are always exceptionally slow. We focus on the most interesting cases here, and include the remaining results in the full version of our paper [25]. We present most of our data in tables sorted in ascending order of wallclock time. Each table is for a single, large input file. The tables represent raw metrics divided by the minimum value attained by any heap, such that a highlighted value of **1.00** is the minimum, while a value c is c times the minimum. These ratios make it easier to interpret relative performance instead of the full counts. In order to keep the tables compact, the column titles have been abbreviated: `time` is wallclock time, `inst` is the dynamic instruction count, `l1_rd` and `l1_wr` are the number of L1 reads and writes respectively, `l2_rd` and `l2_wr` are the L2 reads and writes respectively, `br` is the number of dynamic branches, and `l1_m`, `l2_m` and `br_m` are the number of L1 misses, L2 misses, and branch mispredictions.

We initially ran each experiment on many problem sizes. We found that in most cases the relative performance stabilized very quickly, so from here on we only present data for the largest problem size. See Figures 1 and 2 for some evidence of this stabilization. The heaps are separated into two classes so as to unclutter the plots and give a consistent axis. The operation counts are the sum of the counts of Insert, DecreaseKey, and DeleteMin operations. In Figure 1, all operation counts are scaled by $\log n$, where n is the average size of the heap. In Figure 2, only the DeleteMin count is scaled by $\log n$. This scaling approximately reflects the amortized bounds for each heap.

Before diving into the results, we first make a high-level observation. The number of L1 cache misses appears to be the metric most strongly correlated with wallclock time. It is not a perfect predictor, and inversions in ordering certainly exist. Some of these inversions can be explained by L2 cache misses, write counts, or branch misprediction. Others appear to be outliers or are otherwise yet unexplained.

4.1 Conventional wisdom holds. We first examine two cases where the conventional wisdom holds. As seen in Table 2, the implicit-simple heaps handle sorting workloads very well. The best performance is achieved by the implicit-simple-4 heap. The Fibonacci heap is almost seven times as slow as the fastest, which does indeed echo old complaints about its speed. The pairing heap and binomial queue fair better here, but still poorly at at least four times as slow as the fastest. Without any key decreases, the rank-relaxed weak queue is essentially just an alternate implementation of a binomial queue, so it is not terribly surprising that it does better than the Fibonacci heap.

Similarly with Dijkstra's algorithm on the full USA road map (Table 3), we see implicit-4 heaps performing quite well, while Fibonacci heaps are roughly three times as slow. The explicit heaps are noticeably slower even than Fibonacci heaps, and the only Fibonacci relaxation to perform well here is the pairing heap. The others are in fact *slower* than their conceptual ancestor. Although they exhibit similar caching behavior, their code is somewhat more complicated, which may be contributing to the slowdown.

Both of the above workloads are very well-studied, and as such the relative performance of the older heap variants should not be very surprising.

4.2 Degenerate results. We now turn to our randomized insertion-deletion workload. The results here are more surprising. Recall from Lemma 3.1 that this workload is degenerate, in that as the sequence goes on, the most recently inserted item is very likely to be the next item deleted. Nevertheless, this sequence is commonly used in empirical studies. The shortest sequence we tested, $c = 1$, remains rather close to the sorting workload. On the other hand, when the sequence is very long ($c = 1024$), as shown in Table 4, we see a very different picture. The queue-based structures outperform the implicit heaps by a factor of at least two. Under these assumptions about the distribution, an Insert operation in a d-ary heap results in the node being sifted all the way to the top, and the subsequent deletion on average results in another long sifting sequence. In a queue structure with lazy insertion, the Insert commonly results in a singleton node which is simply removed afterwards with little to no restructuring.

Although degenerate in the above case, a generalization of this sequence becomes a natural sequence for which efficient structures have been designed. Consider workloads which frequently insert new items near the minimum rather than toward the bottom of the heap. Let $r(x)$ denote the rank of x among the items in the heap, such that the rank of the minimum is 1 and the maximum is n. Similarly let $m(x)$ be the maximum value of $r(x)$ over the lifetime of x in the heap. Then there are structures which are optimized for both the case of frequently deleting small-rank items and the

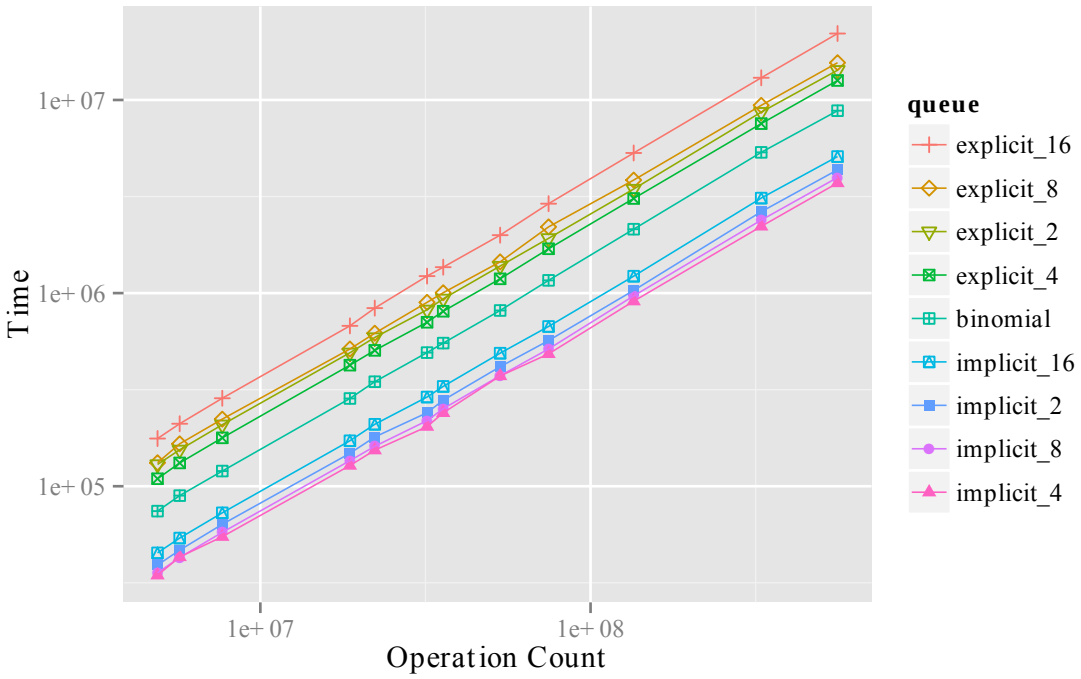

Figure 1: Dijkstra on the full USA road map. All operation counts are scaled by $\log n$.

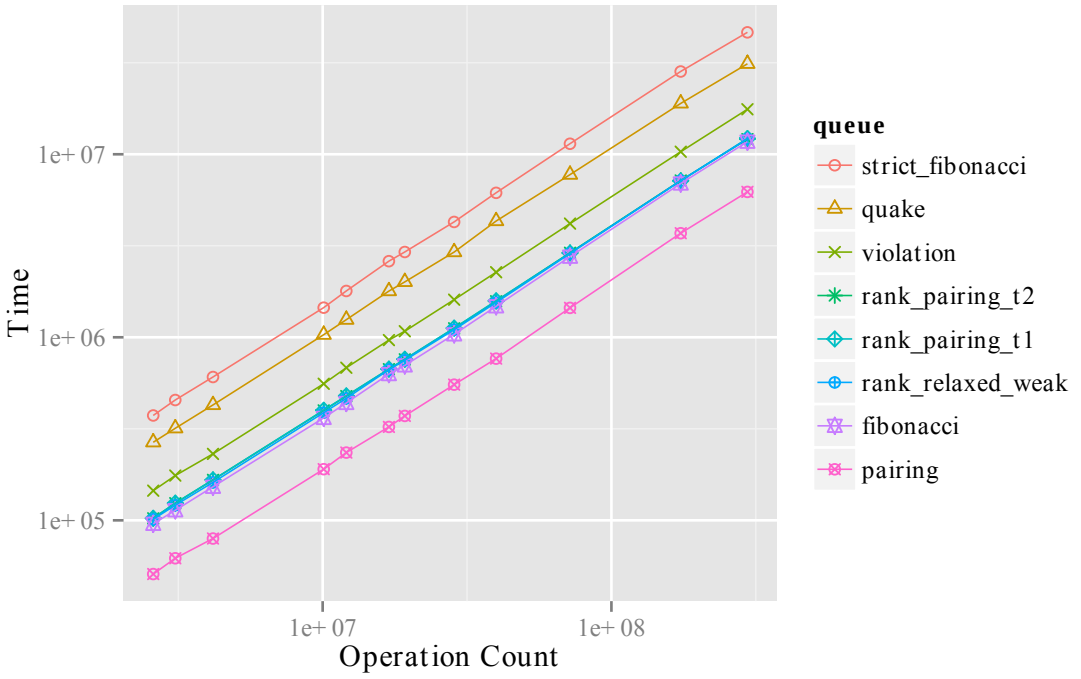

Figure 2: Dijkstra on the full USA road map. The DeleteMin count is scaled by $\log n$.

Table 2: Sorting

Heap Size – max = 4194304, average = 2097152

Ratio of Operations – INSERT : DELETEMIN : DECREASEKEY = 1.00 : 1.00 : 0.00

queue	time	inst	l1_rd	l1_wr	l2_rd	l2_wr	br	l1_m	l2_m	br_m
implicit_simple_4	**1.00**	**1.00**	**1.00**	1.33	**1.00**	1.01	**1.00**	**1.00**	**1.00**	**1.00**
implicit_simple_8	1.05	1.23	1.09	1.11	1.01	1.00	1.12	1.01	1.01	1.12
implicit_simple_16	1.25	1.78	1.39	**1.00**	1.28	**1.00**	1.53	1.19	1.27	1.53
implicit_simple_2	1.59	1.12	1.23	1.98	1.01	1.04	1.29	1.31	1.01	1.29
implicit_8	3.30	1.16	2.17	1.79	3.94	2.05	1.61	1.90	3.90	1.61
implicit_4	3.66	1.04	1.88	2.01	3.26	2.04	1.65	1.76	3.23	1.65
implicit_16	3.91	1.54	2.84	1.68	5.58	2.09	1.94	2.33	5.50	1.94
pairing	4.29	1.06	2.08	6.78	2.35	38.14	2.04	3.05	3.13	2.04
binomial	4.63	1.73	4.22	9.45	1.51	44.62	2.58	5.15	2.44	2.58
implicit_2	4.75	1.25	1.98	2.66	3.82	2.07	2.41	1.99	3.78	2.41
explicit_4	5.10	3.11	6.52	10.60	2.06	117.30	7.07	6.99	4.56	7.07
rank_relaxed_weak	5.14	2.46	6.06	9.86	2.43	10.69	4.85	6.50	2.61	4.85
explicit_2	5.69	4.13	8.43	13.60	1.51	113.31	8.87	9.01	3.93	8.87
fibonacci	6.81	2.37	4.80	13.53	1.54	101.32	5.09	6.53	3.70	5.09
explicit_8	7.93	3.38	7.48	12.49	3.71	162.24	8.22	8.10	7.14	8.22
rank_pairing_t2	8.35	2.39	5.21	7.37	1.96	47.12	5.70	5.32	2.94	5.70
rank_pairing_t1	8.40	2.39	5.21	7.37	1.96	47.12	5.70	5.32	2.94	5.70
violation	10.13	3.39	6.23	10.29	3.45	6.46	7.52	6.71	3.51	7.52
explicit_16	12.62	4.41	10.17	16.95	5.90	250.87	11.49	11.00	11.21	11.49
quake	13.76	6.84	16.98	31.33	3.10	123.25	10.40	19.10	5.71	10.40
strict_fibonacci	14.91	11.60	31.49	62.56	4.33	52.71	19.47	36.49	5.38	19.47

Table 3: Dijkstra – full USA road map

Heap Size – max = 4200, average = 2489

Ratio of Operations – INSERT : DELETEMIN : DECREASEKEY = 13.98 : 13.98 : 1.00

queue	time	inst	l1_rd	l1_wr	l2_rd	l2_wr	br	l1_m	l2_m	br_m
implicit_4	**1.00**	**1.00**	**1.00**	1.10	1.35	1.06	**1.00**	**1.00**	1.00	**1.00**
implicit_8	1.07	1.12	1.12	1.03	1.61	1.18	1.01	1.07	1.20	1.01
implicit_2	1.17	1.10	1.01	1.27	1.35	**1.00**	1.33	1.05	**1.00**	1.33
implicit_16	1.37	1.42	1.38	**1.00**	2.20	1.35	1.21	1.24	1.63	1.21
pairing	1.68	1.09	1.12	2.95	1.71	28.57	1.39	1.60	1.75	1.39
binomial	2.37	1.49	1.83	3.49	1.30	34.57	1.49	2.24	1.56	1.49
fibonacci	3.15	2.00	2.09	5.03	1.73	79.53	2.91	2.85	2.67	2.91
rank_pairing_t2	3.26	1.98	2.16	2.85	1.34	35.46	3.19	2.29	1.61	3.19
rank_relaxed_weak	3.27	2.21	2.72	3.62	2.34	10.01	3.08	2.90	1.89	3.08
rank_pairing_t1	3.29	1.98	2.16	2.85	1.33	35.35	3.19	2.29	1.60	3.19
explicit_4	3.39	2.69	2.83	4.11	1.97	104.57	4.22	3.11	3.29	4.22
explicit_2	3.84	3.35	3.39	4.84	**1.00**	74.61	5.01	3.71	2.05	5.01
explicit_8	4.20	3.01	3.32	5.00	4.50	168.91	5.04	3.70	6.28	5.04
violation	4.74	2.85	2.67	3.92	2.60	4.24	4.38	2.95	1.97	4.38
explicit_16	5.94	3.94	4.56	6.81	8.02	276.59	7.13	5.06	10.76	7.13
quake	8.40	5.84	6.82	10.69	3.45	137.91	6.90	7.72	4.97	6.90
strict_fibonacci	12.49	9.47	12.50	22.07	6.96	84.51	11.47	14.83	6.58	11.47

Table 4: Randomized Insert–DeleteMin (Degenerate) – $c = 1024$
Heap Size – max = 131073, average = 131041
Ratio of Operations – Insert : DeleteMin : DecreaseKey = 1.00 : 1.00 : 0.00

queue	time	inst	l1_rd	l1_wr	l2_rd	l2_wr	br	l1_m	l2_m	br_m
pairing	**1.00**	**1.00**	**1.00**	1.08	1.60	4.96	1.17	**1.00**	1.34	1.17
binomial	1.14	1.02	1.08	1.12	**1.00**	5.44	**1.00**	1.06	**1.00**	**1.00**
rank_relaxed_weak	1.48	1.43	1.50	1.49	1.62	1.09	1.91	1.45	1.07	1.91
fibonacci	1.72	1.86	1.41	1.50	1.01	11.85	4.09	1.40	1.46	4.09
rank_pairing_t2	2.06	2.25	1.58	1.30	1.08	5.55	5.52	1.43	1.06	5.52
rank_pairing_t1	2.07	2.25	1.58	1.30	1.08	5.55	5.52	1.43	1.06	5.52
implicit_8	4.90	4.68	3.03	1.66	86.87	93.72	9.64	2.42	60.22	9.64
implicit_simple_8	5.25	4.77	1.65	1.11	119.49	173.98	6.09	1.40	86.00	6.09
implicit_simple_4	6.12	4.30	1.70	1.42	119.46	347.11	6.09	1.54	98.18	6.09
implicit_4	6.64	4.61	2.82	1.97	132.53	264.11	11.15	2.42	100.40	11.15
implicit_simple_16	6.76	7.11	2.11	**1.00**	40.98	**1.00**	8.46	1.62	25.37	8.46
implicit_16	6.78	6.26	4.03	1.55	108.93	96.37	11.50	2.97	74.02	11.50
quake	7.37	5.86	2.97	2.11	1.79	14.30	17.39	2.55	2.11	17.39
violation	7.88	5.52	3.91	3.17	2.56	1.76	16.97	3.51	1.70	16.97
strict_fibonacci	9.76	2.47	2.46	2.88	47.26	9.78	4.48	2.54	29.86	4.48
implicit_simple_2	11.04	5.32	2.34	2.26	139.20	866.81	8.62	2.23	146.96	8.62
implicit_2	11.87	6.07	3.29	2.81	159.38	785.54	17.73	3.01	153.69	17.73
explicit_4	17.50	18.16	13.38	12.77	5.11	37.38	59.93	12.73	5.79	59.93
explicit_8	20.14	19.69	14.98	14.51	14.47	95.52	68.80	14.33	15.66	68.80
explicit_2	23.58	23.13	16.78	16.13	4.13	30.65	70.99	16.01	4.71	70.99
explicit_16	44.51	27.29	21.42	20.86	55.83	403.73	100.53	20.54	62.89	100.53

case of frequently deleting large-rank items. The fishspear data structure achieves an $\mathcal{O}(\log m(x))$ bound for deletion while rank-sensitive priority queues achieve an $\mathcal{O}(\log({}^{n}/_{r(x)}))$ bound [8, 14]. Additonally, pairing heaps have been shown to support DeleteMin in $\mathcal{O}(\log k)$ time where k is the number of heap operations since the minimum item was inserted [20].

The event simulation literature, largely orthogonal to the theory literature, includes more sophisticated random models for generating insertion-deletion workloads. One in particular to note is the so-called "classic hold" model which is essentially the same as the degenerate model, except that instead of inserting a completely random key in each iteration, the new key is equal to the most recently deleted key plus a positive random value. This avoids the degeneracy. This and other models were explored in a previous experimental study [28]. That study also considers several special-case priority queues with poor theoretical bounds (e.g., $\omega(\log n)$) which nonetheless perform quite well for event simulation workloads.

4.3 Surpising results. As noted in our discussion of the workloads, adding even a single key decrease per iteration to the random sequences lessens the degeneracy. Furthermore in Table 5 we see that if the key decreases do not always generate a new minimum, as would be the case in many applications such as graph search, then implicit heaps with large branching factors continue to perform well. When the key decreases always produce new minima, the amortized structures come out ahead, while worst-case structures (implicit heaps and binomial queues included) fair poorly, as shown in Table 6. As these sequences get longer, e.g. $c = 1024$ and $k = 1$, the Fibonacci relaxations gain ground, with rank-pairing-t1 heaps surpassing Fibonacci heaps. We note that the change in performance coincides with a large gap in L2 cache misses, and is likely due to the long sifting process in d-ary heaps.

If we increase the density of key decreases in the sequence, then we see something strange (Table 7). Suddenly the d-ary heaps are doing well, and in particular the explicit heaps outperform the implicit ones. One possible explanation for this is that the level of indirection in the implicit heap implementations requires them to not only touch the same allocated nodes that the explicit heaps touch, but also to jump around in the structural array while doing path traversals. Noting that implicit and explicit heaps have a similar number of L1 misses, this is one of the few other workloads for which L2 behavior is a better performance predictor.

As to why the d-ary heaps outperform the amor-

Table 5: Randomized DecreaseKey – Middle, $c = 1$, $k = 1$
Heap Size – max = 8388609, average = 7340032
Ratio of Operations – Insert : DeleteMin : DecreaseKey = 2.00 : 1.00 : 1.00

queue	time	inst	l1_rd	l1_wr	l2_rd	l2_wr	br	l1_m	l2_m	br_m
implicit_8	**1.00**	1.23	1.14	1.07	2.91	1.21	**1.00**	1.04	1.58	**1.00**
implicit_4	1.05	1.15	1.03	1.19	2.57	1.63	1.07	**1.00**	1.43	1.07
binomial	1.07	1.42	1.84	3.99	1.28	6.09	1.24	2.26	1.05	1.24
pairing	1.09	**1.00**	**1.00**	2.90	1.52	4.92	1.04	1.41	1.10	1.04
implicit_16	1.15	1.53	1.41	**1.00**	3.98	1.02	1.13	1.21	2.12	1.13
rank_relaxed_weak	1.38	2.06	2.57	3.74	1.68	2.28	2.39	2.69	1.02	2.39
implicit_2	1.47	1.41	1.12	1.55	3.05	2.91	1.61	1.15	1.77	1.61
fibonacci	1.58	1.98	2.02	5.23	**1.00**	11.07	2.39	2.70	1.23	2.39
explicit_4	1.86	3.82	3.77	5.89	1.40	13.92	5.10	4.05	1.63	5.10
rank_pairing_t2	1.98	2.04	2.16	2.98	1.29	5.18	2.74	2.22	1.00	2.74
rank_pairing_t1	1.98	2.04	2.16	2.98	1.29	5.18	2.74	2.22	**1.00**	2.74
explicit_2	2.22	5.00	4.80	7.44	1.09	12.73	6.32	5.14	1.39	6.32
explicit_8	2.29	4.08	4.24	6.80	2.48	19.90	5.83	4.60	2.57	5.83
violation	2.30	2.73	2.52	3.85	2.13	**1.00**	3.49	2.69	1.17	3.49
quake	2.96	5.19	6.07	10.51	1.88	12.47	4.70	6.78	1.78	4.70
strict_fibonacci	3.28	8.40	11.05	20.87	2.76	6.29	8.08	12.79	1.83	8.08
explicit_16	3.94	5.20	5.59	8.85	4.00	31.73	7.98	6.03	4.12	7.98

Table 6: Randomized DecreaseKey – Min, $c = 1$, $k = 1$
Heap Size – max = 8388609, average = 7340032
Ratio of Operations – Insert : DeleteMin : DecreaseKey = 2.00 : 1.00 : 1.00

queue	time	inst	l1_rd	l1_wr	l2_rd	l2_wr	br	l1_m	l2_m	br_m
pairing	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	1.65	**1.00**	**1.00**	**1.00**	**1.00**
fibonacci	3.04	2.62	2.21	2.58	3.00	3.52	4.91	2.37	2.43	4.91
rank_relaxed_weak	3.65	3.16	3.05	2.18	4.66	2.01	5.89	2.68	2.40	5.89
rank_pairing_t2	5.43	3.42	2.57	1.96	5.94	1.58	7.54	2.32	2.67	7.54
rank_pairing_t1	5.95	3.39	2.56	1.95	5.94	1.57	7.46	2.30	2.67	7.46
violation	6.32	4.54	3.32	2.40	5.93	1.04	10.47	2.93	2.46	10.47
quake	6.55	6.59	4.37	3.53	5.28	2.61	13.31	4.02	2.86	13.31
implicit_8	6.98	3.58	2.42	1.24	40.06	1.02	5.55	1.92	14.19	5.55
implicit_4	7.25	3.47	2.25	1.41	33.17	**1.00**	6.20	1.90	11.81	6.20
strict_fibonacci	7.39	8.44	8.38	8.31	9.76	4.81	13.05	8.35	5.27	13.05
implicit_16	9.40	4.35	2.94	1.14	56.29	1.05	6.02	2.18	19.79	6.02
implicit_2	10.17	4.48	2.59	1.93	38.66	1.01	9.67	2.31	13.70	9.67
binomial	12.14	5.90	5.55	6.54	30.10	14.21	10.45	5.97	16.01	10.45
explicit_4	14.95	12.84	9.64	8.33	24.42	18.16	30.31	9.09	15.63	30.31
explicit_2	16.24	17.12	12.53	10.60	24.32	12.53	38.42	11.71	13.36	38.42
explicit_8	21.07	13.80	10.88	9.74	39.17	28.94	34.46	10.40	25.00	34.46
explicit_16	31.01	17.68	14.40	12.85	60.51	48.34	47.08	13.74	40.06	47.08

Table 7: Randomized DecreaseKey – Min, $c = 1$, $k = 1024$
Heap Size – max = 262145, average = 262017
Ratio of Operations – Insert : DeleteMin : DecreaseKey = 2.00 : 1.00 : 1024.00

queue	time	inst	l1_rd	l1_wr	l2_rd	l2_wr	br	l1_m	l2_m	br_m
explicit_4	**1.00**	1.07	1.04	1.07	1.23	5.92	1.17	1.05	1.18	1.17
explicit_16	1.07	1.11	1.08	1.11	1.50	6.22	1.30	1.09	1.36	1.30
implicit_16	1.16	1.01	1.01	**1.00**	2.16	5.71	1.01	1.00	1.71	1.01
implicit_2	1.22	1.01	1.00	1.01	2.09	5.71	1.03	1.00	1.67	1.03
binomial	1.23	1.44	1.89	1.31	1.79	5.83	2.46	1.71	1.50	2.46
explicit_8	1.26	1.07	1.05	1.08	1.47	6.03	1.20	1.06	1.32	1.20
explicit_2	1.27	1.10	1.06	1.09	**1.00**	5.88	1.24	1.07	1.03	1.24
implicit_8	1.28	1.00	1.00	1.00	2.17	5.71	**1.00**	1.00	1.71	**1.00**
implicit_4	1.31	**1.00**	**1.00**	1.00	2.15	5.71	1.00	**1.00**	1.70	1.00
pairing	1.39	3.58	2.36	5.77	3.57	14.48	7.18	3.41	3.20	7.18
strict_fibonacci	1.41	1.43	1.61	2.22	1.56	**1.00**	1.58	1.80	**1.00**	1.58
violation	1.82	2.37	2.04	1.73	2.85	2.37	4.85	1.94	1.87	4.85
rank_pairing_t1	1.84	4.97	3.80	4.05	6.54	11.89	12.30	3.87	4.77	12.30
rank_pairing_t2	1.84	5.06	3.81	4.05	6.56	11.89	12.64	3.88	4.79	12.64
fibonacci	2.82	7.09	5.36	11.55	3.68	31.54	12.86	7.26	4.55	12.86
rank_relaxed_weak	2.99	11.32	8.93	11.76	4.04	14.28	25.57	9.79	3.47	25.57
quake	5.07	16.04	14.80	20.94	7.26	39.64	20.19	16.68	7.28	20.19

tized structures, consider the overall pattern. If many nodes have their keys decreased in a pairing heap, for instance, then their subtrees are simply reattached underneath the root. Each node access is likely to trigger a cache miss, as there will be little revisiting of nodes other than the root, and the subsequent deletion will have to examine each of these nodes again in order to restructure the tree. On the other hand, in the d-ary heaps, all the sifting is along ancestral paths which share many nodes between operations, and hence the caching effects are more favorable.

4.4 Other workloads. The full version of our paper contains a complete set of data and further discussion of results [25]. Among the other workloads we tested, those that generate relatively small heap sizes favor the implicit heaps, while those operating on larger heap sizes favor amortized structures, especially the pairing heap. Very dense key-decrease workloads, including some "bad" inputs for Dijkstra's algorithm and the Nagamochi-Ibaraki workloads, favor implicit and explicit heaps. The network simulation workloads, which produce relatively small heap sizes and have no key decreases, all favor implicit-simple heaps.

5 Sanity Checks

We performed a few auxiliary experiments to verify our findings in the previous section.

Table 8: Tweaking node size to test caching effects.

	implicit			pairing		
node size	time	rd	wr	time	rd	wr
1.00	1.00	1.00	1.00	1.00	1.00	1.00
2.00	1.08	1.10	1.67	1.36	1.08	1.45
4.00	1.29	1.32	3.00	1.84	1.08	2.24

5.1 Testing the caching hypothesis. In order to lend some credence to our claim that caching is the primary predictor of performance in many of these test cases, we ran a few additional tests, tweaking the parameters of our implementations. We added an extra padding field to the node in our pairing heap and implicit-4 heap implementations. The extra field does not generate additional instructions in the code other than in the original memory allocation process (not included in the timing procedures) and thus the only change should be in the memory address allocated to the nodes. This can affect both caching and branch prediction. Through repeated doubling of node size, we find that even though the dynamic instruction count does not grow, the wall-clock time does—in fact, it grows roughly in proportion to the cache miss rate. A less-pronounced effect also accompanies the growth of the misprediction rate.

One potentially interesting observation from these experiments is this: the instruction patterns of pairing heaps is *write-first*, while that of implicit heaps is *read-*

first. By this we mean that typically, whenever an implicit heap touches a node, it does so first via a read, while a pairing heap quite often simply overwrites data in the node without reading it. This means that the cache behavior for pairing heaps is skewed toward write misses, while implicit heaps are skewed toward read misses. Table 8 shows the read and write miss rates for both heaps.

5.2 Comparison to an existing implementation.

We ran a few experiments against Sanders' implementation of the sequence heap [29], which has a reputation of being hard to beat in practice. This gives us an easy way to benchmark our own untuned implementations to see how they compare against a well-engineered one. The results were encouraging.

Of the four workloads we tested, the sequence heap was faster than any of our implementations on two of them, while it was slower on the other two. More specifically, the sequence heap was 1.97 times faster than the implicit-simple-4 heap on the sorting workload, and a significant 3.69 times faster than the pairing heap on the randomized insertion-deletion workload with $c = 32$. Our pairing heap implementation performed 1.36 times faster than the sequence heap on the insertion-deletion workload with $c = 1024$, and the implicit-simple-2 heap was 1.15 times faster on one of the network simulator workloads.

6 Remarks

As declared in the introduction, this is by no means a final study. The push in the past decade for better-performing Fibonacci-like heaps, while it may have led to theoretical simplifications, does not seem to have yielded obvious practical benefits. The results show that the optimal choice of implementation is strongly input-dependent. Furthermore, it shows that care must be taken to optimize for cache performance, primarily at the L1-L2 barrier. This suggests that complicated, cache-oblivious structures are unlikely to perform well compared to simpler, cache-aware structures. Some obvious candidates for renewed testing are sequence heaps and B-heaps [21]. Another obvious direction for future work is to explore other classes of workloads.

We hope that our study gives future theorists and practitioners a new outlook on the state of affairs. Unfortunately, there is no simple answer to which heap should be used when. Picking the best tool for the job will likely require experimentation between existing implementations or careful analysis of the expected workload's caching behavior against each heap. To this end, we hope that our simple implementations of various heap structures will serve as a useful resource.

Acknowledgments. We would like to thank Jeff Erickson for the considerable guidance he provided the first author in the initial stages of this project. Some of the compute cluster resources at Princeton were donated by Yahoo!.

References

[1] Priority queue testing. `http://code.google.com/p/priority-queue-testing/`.

[2] Gerth S. Brodal, George Lagogiannis, and Robert E. Tarjan. Strict Fibonacci heaps. In *Proc. 44th Annual ACM Symposium on Theory of Computing*, pages 1177–1184, 2012.

[3] Asger Bruun, Stefan Edelkamp, Jyrki Katajainen, and Jen Rasmussen. Policy-based benchmarking of weak heaps and their relatives. In *Proc. 9th Annual International Symposium on Experimental Algorithms*, pages 424–435. 2010.

[4] Cachegrind: A cache and branch-prediction profiler. `http://valgrind.org/docs/manual/cg-manual.html`.

[5] Timothy M. Chan. Quake heaps: a simple alternative to Fibonacci heaps, 2009.

[6] Rezaul A. Chowdhury. *Cache-efficient Algorithms and Data Structures: Theory and Experimental Evaluation*. PhD thesis, The University of Texas at Austin, 2007.

[7] Raiciu Costin and Mark Handley. Multipath TCP implementations. `http://nrg.cs.ucl.ac.uk/mptcp/implementation.html`.

[8] Brian C. Dean and Zachary H. Jones. Rank-sensitive priority queues. In *Proc. of the 11th International Symposium on Algorithms and Data Structures*, pages 181–192, 2009.

[9] DIMACS. 5th DIMACS challenge: Priority queues. `http://www.cs.amherst.edu/~ccm/challenge5/p_queue/index.html`.

[10] DIMACS. 9th DIMACS implementation challenge: Shortest paths. `http://www.dis.uniroma1.it/~challenge9/download.shtml`.

[11] Stefan Edelkamp, Amr Elmasry, and Jyrki Katajainen. The weak-heap family of priority queues in theory and praxis. In *Proc. of the 18th Computing: The Australasian Theory Symposium*, pages 103–112, 2012.

[12] Amr Elmasry. The violation heap: A relaxed Fibonacci-like heap. In *Proc. 16th Annual International Conference on Computing and Combina-*

torics, pages 479–488, 2010.

[13] Amr Elmasry and Jyrki Katajainen. Fat heaps without regular counters. In *Proc. 6th Workshop on Algorithms and Computation*, pages 173–185. Springer, 2012.

[14] Michael J. Fischer and Michael S. Paterson. Fishspear: a priority queue algorithm. *J. ACM*, 41(1):3–30, 1994.

[15] Michael L. Fredman. On the efficiency of pairing heaps and related data structures. *J. ACM*, 46(4):473–501, 1999.

[16] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1–4):111–129, 1986.

[17] Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[18] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A scalable and flexible data center network. *ACM SIGCOMM Computer Communication Review*, 39(4):51–62, 2009.

[19] Bernhard Haeupler, Siddhartha Sen, and Robert E. Tarjan. Rank-pairing heaps. *SIAM J. Computing*, 40(6):1463–1485, 2011.

[20] John Iacono. Improved upper bounds for pairing heaps. *CoRR*, abs/1110.4428, 2011.

[21] Poul-Henning Kamp. You're doing it wrong. *ACM Queue: Tomorrow's Computing Today*, 8(6), 2010.

[22] Haim Kaplan and Robert E. Tarjan. Thin heaps, thick heaps. *ACM Trans. on Algorithms*, 4(1), 2008.

[23] Donald E. Knuth. *The art of computer programming, volume 3: sorting and searching (2nd ed.)*. Addison Wesley Longman Publishing Co., Inc., 1998.

[24] Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of heaps. *ACM J. of Experimental Algorithmics*, 1(4), 1996.

[25] Daniel H. Larkin, Siddhartha Sen, and Robert E. Tarjan. A back–to–basics empirical study of priority queues. CoRR, 2013.

[26] Bernard M. E. Moret and Henry D. Shapiro. An empirical analysis of algorithms for constructing a minimum spanning tree. In *Algorithms and Data Structures*, volume 519 of *LNCS*, pages 400–411. Springer, 1991.

[27] Seth Pettie. Towards a final analysis of pairing heaps. In *Proc. of 46th Annual IEEE Symposium on Foundations of Computer Science*, pages 174–183, 2005.

[28] Robert Rönngren and Rassul Ayani. A comparative study of parallel and sequential priority queue algorithms. *ACM Trans. on Modeling and Computer Simulation*, 7(2):157–209, 1997.

[29] Peter Sanders. Fast priority queues for cached memory. *ACM J. of Experimental Algorithmics*, 5(7), 2000.

[30] John T. Stasko and Jeffrey S. Vitter. Pairing heaps: experiments and analysis. *Commun. ACM*, 30(3):234–249, 1987.

[31] Tadao Takaoka. Theory of 2-3 heaps. In *Proc. 5th Annual International Conference on Computing and Combinatorics*, pages 41–50, 1999.

[32] Jean Vuillemin. A data structure for manipulating priority queues. *Commun. ACM*, 21:309–315, 1978.

[33] John W. J. Williams. Algorithm 232 heapsort. *Commun. ACM*, 7(6):347–349, 1964.