SimdQuickHeap: The QuickHeap Reconsidered

Johannes Breitling
Karlsruhe Institute of Technology, Germany

Ragnar Groot Koerkamp
Karlsruhe Institute of Technology, Germany

Marvin Williams
Karlsruhe Institute of Technology, Germany

Abstract

Motivation. Priority queues are data structures that maintain a dynamic collection of elements and allow inserting new elements and removing the smallest element. The most widely known and used priority queue is likely the implicit binary heap, even though it has frequent cache misses and is hard to optimize using e.g. SIMD instructions.

Contributions. We introduce the SimdQuickHeap, a variant of the QuickHeap that was introduced by Navarro and Paredes in 2010. As suggested by the name, the data structure bears some similarity to QuickSort. We modify the data layout of the original QuickHeap to have all *pivots* adjacent in memory, with elements between consecutive pivots stored in dedicated *buckets*. This allows efficient SIMD implementations for both partitioning of buckets and scanning the list of pivots to find the bucket to append newly inserted elements to.

The SimdQuickHeap has amortized expected complexity $O(\log n)$ per operation, which improves to $O(\frac{1}{W} \log n)$ in non-degenerate cases, where W is the number of words in a SIMD register. In this case, the I/O-complexity is amortized $O(\frac{1}{B})$ per `push` and $O(\frac{1}{B} \log_2 \frac{n}{M})$ per `pop`.

Results. In synthetic benchmarks, the SimdQuickHeap is 1.2× to 1.7× as fast as the monotone radix heap, the next-best competitor, and 1.4× to 2.8× as fast as the superscalar sample queue, the fastest comparison-based priority queue. The SimdQuickHeap needs around $1.5 \log_2 n$ comparisons and $\log_2 n$ nanoseconds per pair of push and pop operations. On graph benchmarks with Dijkstra's shortest path algorithm and Jarník–Prim's minimum spanning tree algorithm, the SimdQuickHeap is consistently the fastest.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Heap; SIMD; Priority Queue

Digital Object Identifier 10.4230/LIPIcs.ESA.2026.14

Supplementary Material *Software (Source Code)*: `https://github.com/RagnarGrootKoerkamp/quickheap`

Acknowledgements We thank Peter Sanders for discussions related to this work and feedback on a draft of this paper.

1 Introduction

The Priority Queue (PQ) is a fundamental data structure that manages a collection of elements with associated priorities, allowing the insertion of new elements and the extraction of the element with the highest priority. PQs are central to a wide range of algorithms that rely on dynamically reordering and prioritizing tasks, most famously Dijkstra's shortest-path algorithm [29], but also job scheduling, discrete event simulation, and branch-and-bound searches. As a result, PQs have been studied extensively, resulting in a large variety of priority queue designs (see Section 3).

Despite being among the oldest designs, (implicit) binary heaps are probably still the most widely used priority queues in practice[1]. There are compelling reasons for their prevalence: they are conceptually simple, easy to implement, and have $\Theta(\log_2 n)$ worst-case running time with small constants for both insertions and deletions. While many theoretically superior designs have been proposed, most of them are prohibitively complex for practical use, and surprisingly few designs that are faster in practice have emerged.

One such design is the *QuickHeap* [47], introduced in 2010 by Navarro and Paredes. It uses a recursive partitioning scheme reminiscent of quicksort [39], hence the name[2]. However, it has received little attention in the literature. Figure 1 shows a schematic overview of the QuickHeap among other priority queue designs. We show that, due to its partitioning scheme, the QuickHeap lends itself to the SIMD (*single instruction, multiple data*) paradigm much more naturally than binary heaps and most other heaps based on merging trees or streams of elements. SIMD instructions are steadily becoming faster, more efficient, more powerful, and are capable of operating on wider registers. In light of these advancements, we argue that the QuickHeap should be reconsidered.

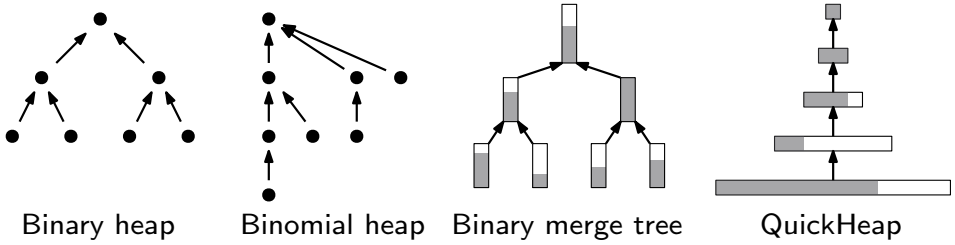

Figure 1 Overview of some heaps. Black dots indicate single elements, and arrows point from larger to smaller elements. Vertical blocks indicate a sorted list of elements, while horizontal blocks indicate an unsorted list of elements. The QuickHeap stores pivots separating the buckets explicitly.

Contributions. We present *SimdQuickHeap*, an adaptation of the QuickHeap that achieves significantly improved performance through optimized data layout and SIMD instructions. To our knowledge, it is the first practical SIMD-optimized priority queue implementation.

The `push` and `pop` operations have an amortized worst-case run time bound of $\mathcal{O}(\log n)$ and $\mathcal{O}(1)$, respectively. In most practical cases, the SIMD implementation speeds up both operations by a factor of the SIMD register width W. Like the QuickHeap, the SimdQuickHeap is cache-oblivious with an amortized I/O-complexity of amortized $\mathcal{O}(\frac{1}{B}\log\frac{n}{M})$ per operation for a main memory of M words and block size B, assuming $M = \Omega(\log n)$.

On synthetic benchmarks, the SimdQuickHeap is 1.2× to 1.7× as fast as a state-of-the-art implementation of the radix heap [4] and 1.4× to 2.8× as fast as the superscalar sample queue [61]. On real-world instances of Dijkstra's shortest path algorithm [29] and Jarník–Prim's minimum-spanning-tree algorithm [11, 41, 51], it is consistently the fastest method. Notably, while the run time of most tree-based and pointer-based priority queues grows relative to the $\Omega(\log_2 n)$ lower bound as n increases, SimdQuickHeap's run time improves, falling below $\log_2 n$ nanoseconds per `push`-`pop` pair.

[1] They are available in the standard libraries of many programming languages, including C++, Java, C#, Go, Rust, and Python.

[2] Although not a heap in the tree-like sense, the layers between the pivots (Figure 1) do form a linear tree with unordered sets of elements as nodes.

Notation and definitions. Given a universe of elements $\mathcal{U}$ and a strict weak ordering $<$ on $\mathcal{U} \cup \{-\infty, \infty\}$, the priority queue (PQ) data structure is a dynamic collection of elements $\mathcal{Q}$ over $\mathcal{U}$ (with duplicates permitted) that supports the following two operations:
push(e) Insert an element $e \in \mathcal{U}$ into $\mathcal{Q}$.
pop() Remove and return a minimal element $e \in \mathcal{Q}$, i.e., no $e' \in \mathcal{Q}$ with $e' < e$ exists.

For convenience, we assume `top` (`pop` without removing) and `size` (returns $|\mathcal{Q}|$) to also be available. We do not consider a `decreaseKey` operation that replaces arbitrary elements in the PQ by smaller ones, since it usually requires stable references to elements, adding implementation complexity and run-time overhead. While our definition admits distinct, equivalent elements, we refer to all equivalent elements as *equal* elements for simplicity.

2 The Original QuickHeap

In 2006, Paredes and Navarro presented *incremental quick select* [49], an algorithm that takes a list of n elements and then repeatedly returns the next-smallest element, such that returning the k smallest elements in an online setting takes $\mathcal{O}(n + k \log k)$ time. The authors note that their construction can be generalized to a proper priority queue. Consequently, in follow-up work from 2010, they present the QuickHeap [47]. The QuickHeap organizes the elements in disjoint *buckets* $B_1, \ldots, B_\ell$ that are separated by *pivots* $p_1 > \ldots > p_{\ell-1}$ with

$$\infty \succ B_1 \succeq p_1 \succ B_2 \succeq p_2 \succ \cdots \succeq p_{\ell-1} \succ B_\ell \succ -\infty,$$

where $\succeq$ ($\succ$) indicates that the $\geq$ ($>$) inequality holds for all elements in the sets. For the sake of simplicity, in the following we assume that the PQ contains no two equal elements. See Section 4.1 for details on how equal elements are handled in our design. The `pop` operation picks a new pivot p_ℓ (e.g., uniformly at random) from the *bottom* bucket B_ℓ and moves all elements in B_ℓ smaller than p_ℓ into a new bucket $B_{\ell+1}$. If the new bucket is empty, it is discarded and a new pivot is selected. This process is repeated recursively in $B_{\ell+1}$ until the bottom bucket contains only one (the smallest) element. Finally, this element is returned and its bucket is removed. The `push` operation simply inserts the element into the correct bucket according to the pivots.

The original implementation [47] stores all buckets and pivots in an interleaved fashion in a flat array, with an additional array to track the positions of the pivots. Splitting a bucket is implemented analogously to partitioning in quicksort. To make room for a new element in a bucket B_i, all buckets with index smaller than i are shifted one slot to the right by moving two elements per bucket. The data structure is stored in a circular buffer, so that removing the smallest element only requires incrementing a pointer. Figure 2 shows the data layout and illustrates the `push` and `pop` operations.

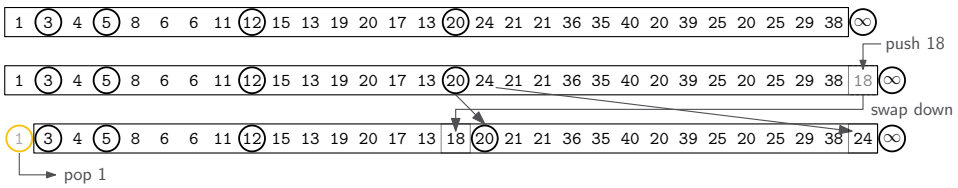

Figure 2 Overview of the QuickHeap data structure [47]. The data is stored in a single buffer. Pivots are circled and partition the data into smaller (left) and larger (right) elements. Their positions are stored in a separate list (not shown). An element is pushed by appending it at the end and then sifting it down until it is less than the corresponding pivot. An element is popped by repeatedly partitioning the bottom layer and then removing the smallest element.

The `push` and `pop` operations of the QuickHeap are both in $\mathcal{O}(\log n)$ *in expectation over uniformly distributed input* [47]: the original QuickHeap can degenerate under specific input distributions. For example, when the inserted elements are decreasing, the number of buckets/pivots grows linearly over time, causing a run time of $\mathcal{O}(n)$ for insertions. Follow-up work introduces *Randomized QuickHeaps* [48], which addresses this issue by probabilistic repartitioning: each time a bucket of size s is shifted during an insertion, with probability $1/s$, the Randomized QuickHeap unites the bucket with all newer buckets by dropping the corresponding pivots. While the Randomized QuickHeap achieves $\mathcal{O}(\log n)$ operations in expectation for *any* input, it adds significant overhead for benign cases. Very recently, Brodal et al. [19] introduced more sophisticated rebalancing schemes (see Section 3). However, these are not directly practical since they rely on linked lists to split and join buckets efficiently.

3 Related work

There is a vast body of literature on priority queues that dates back to the 1950s. Here, we highlight work on practical priority queues and focus on RAM-model complexity, I/O-complexity, the number of comparisons, and the running time. For a broader overview, we refer to the survey by Brodal [15] from 2013. A survey on the performance of classic priority queues is given by Larkin et al. [45]. The QuickHeap does not appear in either of these surveys, highlighting the little attention it has received.

Complexity lower bounds. There is a strong connection between sorting and priority queues [58]. Due to the lower bound for comparison based sorting, n insertions followed by n deletions must take at least $n \log_2 n$ comparisons for a comparison-based priority queue. A `pop` followed by a `push` requires at least $\log_2 n$ comparisons in the worst case amortized over many operations [20].

Binary and d-ary heap. A popular priority queue implementation is the binary heap using an implicit array representation [65]. The binary heap organizes the elements as a binary tree, maintaining the *heap invariant* that no element can be larger than its parent in the tree (unless it is the root). Thus, no element can be smaller than the root element. A binary heap can be implemented efficiently using the *Eytzinger* or *heap layout*, where the elements are stored consecutively in a flat array level by level starting from the root. Consequently, the children of node i are stored at positions $2i$ and $2i + 1$ in the array (1-based indexing). Both the `push` and `pop` operations take $\mathcal{O}(\log_2(n))$ time.

The d-ary heap generalizes the binary heap so that each node has up to d children. The main benefits are better cache locality and fewer levels to traverse. This comes at the cost of needing to find the minimum of d elements on each level when deleting an element.

Amortized heaps. A number of heaps reduce the $\mathcal{O}(\log n)$ cost of binary heap insertions. Binomial heaps (Figure 1) [62, 21] have amortized $\mathcal{O}(1)$ insertions. Fibonacci heaps [36] and pairing heaps [35] reduce this to truly constant time insertions, but have only amortized constant time deletions. The Brodal heap [14] (1996) has $\mathcal{O}(1)$ insertions and non-amortized $\mathcal{O}(\log n)$ deletions, but has a large hidden constant. Other non-amortized structures are the cascade heap [8], which has $\mathcal{O}(1)$ insertions and $\mathcal{O}(\log^* n + \log k)$ for the k'th delete, and the logarithmic funnel [46], which has $\mathcal{O}(\log^* n)$ insertions and $\mathcal{O}(\log n)$ deletions.

Unfortunately, all these structures are pointer-based and inefficient in practice. The weak heap and weak queue are more practical variants of these ideas that can be implemented using a flat array [30, 22, 31, 32].

decreaseKey. Some of the priority queues mentioned above also support `decreaseKey`, which lowers the key associated with a given element. Typically (but not always [24]), data structures supporting this require *pointer stability* of the elements and thus an additional level of indirection, hurting performance. We do not consider these further, but note that in the pointer-based model there is a lower bound of $\Omega(\log \log n)$ for `decreaseKey` [40] that is attained by slim and smooth heaps [55, 56].

I/O-model. Given that optimal priority queues for the RAM-model were already found early on, further research went into obtaining a good I/O-complexity [2]. In this setting, we are given an M-word internal memory and must minimize the number of transfers of B-word blocks to/from external memory. The lower bound for priority queues matches that of sorting, $\Omega(\frac{1}{B} \log_{M/B} \frac{n}{B})$ amortized transfers per operation [2], and a data structure matching this is given by Jiang and Larsen [42]. Practical implementations of I/O-optimal priority queues are the array heap [13] and sequence-heap [52], which improves the constant factor. The QuickHeap has slightly higher I/O cost of $\mathcal{O}((1/B) \log_2(n/M))$ for the `push` and `pop` operations [47]. With additional support for `decreaseKey`, the lower bound increases to $\Omega(\frac{1}{B} \log B / \log \log n)$ [33], and $\mathcal{O}(\frac{1}{B} \log \frac{n}{B} / \log \log n)$ is achieved by [42]. A *cache-oblivious* [37] priority queue with optimal amortized complexity that does not rely on the value of M and B is given by [6, 5], with improvement to $\mathcal{O}(1/B)$ insertions in [18]. The buffer heap [25, 24] supports `decreaseKey` in $\mathcal{O}(\frac{1}{B} \log_2 \frac{n}{M})$ transfers and was discovered in parallel with the bucket heap [17]. The funnel heap [16] is another optimal cache-oblivious queue that relies on *binary mergers* (Figure 1) with strategically chosen buffer sizes.

Other priority queues. There are also data structures that exploit integer keys. These data structures are typically limited to workloads where the deleted elements increase monotonically. We refer to the survey by Costa et al. [26] for an overview. The radix heap [3] inserts keys into a bucket based on the largest bit in which they differ from the current minimum, and has amortized complexity $\mathcal{O}(\log C)$ per operation when inserted keys are at most C larger than the current minimum. Like the QuickHeap, bucket sizes grow roughly exponentially. Further examples are bucket queues [28, 27] (similar to bucket sort) and $\mathcal{O}(\log w)$ dynamic predecessor structures for w-bit keys like Van Emde Boas trees [59], Y-fast tries [64], and fusion trees [34].

Practical/engineered priority queues. Given the plethora of theoretical results, there are surprisingly few engineered implementations. Sequence heaps (2000) [52] provide an optimized I/O-efficient implementation based on merging sorted lists, with the drawback that elements are eagerly sorted, even when they are never removed. The QuickHeap (2010) [47] is indeed much faster when only few elements are removed, and competitive on synthetic benchmarks, but in theory is less I/O-efficient than the sequence heap. The superscalar sample queue (S^3Q, 2021) [61] is a much more recent engineered data structure based on super scalar sample sort [54], that uses k-way partitioning instead of k-way merging to reach the optimal I/O-complexity. In particular, partitioning tends to be easier to optimize than merging, even though k-way splitting is harder to do evenly than data-independent k-way merging.

One further engineered data structure is the B-heap [44], which implements an implicit binary heap using a recursive layout like the Van Emde Boas tree [59].

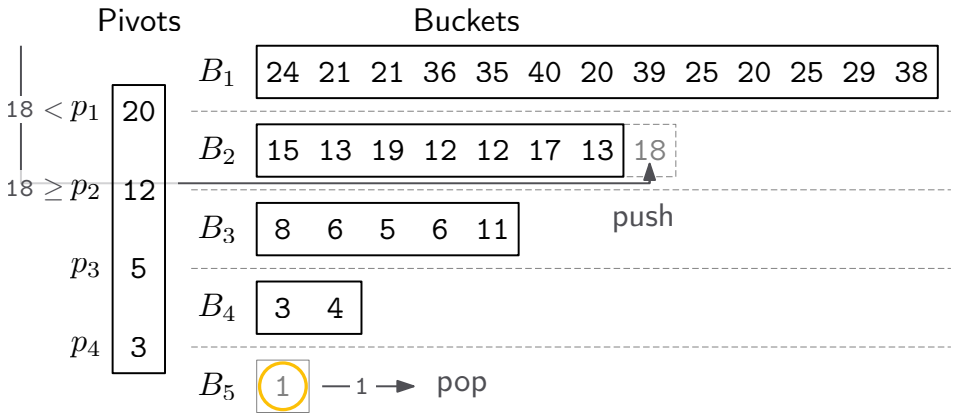

Figure 3 A schematic overview of the SimdQuickHeap: each bucket is a separate array, and the pivots are stored in their own array in sorted order.

Partition-based heaps. Most methods above are based on variously shaped trees with the heap property. Some methods, like the QuickHeap, instead use a path-shaped (linear) heap of buckets, where pivots (implicitly) partition the data into buckets. The priority queue of [6, 5] uses exponentially growing layers where each layer contains an increasing number of sorted buffers of increasing size. The buffer heap [25, 24] is conceptually simpler: each layer contains an unsorted list of elements. The superscalar sample queue combines these ideas, and each level contains k unsorted buffers. Brodal et al. [19] recently revisited this concept and provide *rebalancing strategies* to ensure the number of layers is guaranteed to be logarithmic. In one strategy, two consecutive buckets B_{i-1} and B_i are merged after a `pop` if they contain fewer elements in total than all previous buckets combined, i.e., $|B_i| + |B_{i-1}| < |B_1| + |B_2| + \cdots + |B_{i-2}|$. Under this rule, the number of layers is bounded by $1 + 2\log_2 n$. Another strategy is to maintain the invariant $|B_i| < 3 \cdot 2^i$, resulting in at most $1 + \log_2 n$ layers. Both strategies use a linear time pivot selection algorithm to partition the buckets into the right sizes. Elements are stored in a single linked list so that merging partitions takes constant time.

4 The new SimdQuickHeap

We first describe the basic scalar design of the *SimdQuickHeap* based on the original QuickHeap (Section 2) and analyze its run time. We then discuss improvements using SIMD.

4.1 Scalar Design

Our variant of the QuickHeap keeps an explicit array of all pivots in decreasing order, as shown on the left in Figure 3. The buckets are stored in separate buffers, rather than as a single allocation. The `push` operation classifies each element using a binary search on the pivot array and then appends it to the respective bucket. To partition the bottom bucket for the `pop` operation, a *pivot* is selected and all elements smaller than the pivot are moved to a newly allocated bucket, while compactifying the remaining elements in-place, as shown in Algorithm 1. In case we selected the smallest element in the bucket as a pivot, no element moves to the new bucket, so we pick a new pivot and start over. However, if all elements in the bucket are equal, this does not terminate. For this reason, we treat elements equal to the pivot specially.

Algorithm 1 Pseudo code for the `SimdQuickHeap` data structure. The `pop` operation assumes that it is not empty.

```
structure SimdQuickHeap⟨T⟩
    pivots : Vec⟨T⟩, a vector of decreasing pivots
    buckets : Vec⟨Vec⟨T⟩⟩, a vector of bucket vectors
1:  procedure push(x)
2:      i ← classify(x, pivots)                    ▷ Binary search the first pivot ≤ x.
3:      buckets[i].pushBack(x)
1:  procedure pop
2:      while buckets.last().len() > 1 do          ▷ Partition the bottom layer repeatedly.
3:          (p, B′) ← Partition(buckets.last())
4:          if B′ ≠ ∅ then
5:              buckets.pushBack(B′)
6:              pivots.pushBack(p)
7:      x ← buckets.last().popBack()
8:      buckets.popBack()
9:      pivots.popBack()
10:     return x
1:  procedure partition(B)
2:      (i, p) ← SelectPivot(B)                    ▷ p = B[i]
3:      k ← 0
4:      B′ ← Vec⟨T⟩()
5:      for j ∈ [1, B.len()] do
6:          if B[j] < p or (j > i and B[j] ≤ p) then
7:              B′.pushBack(B[j])
8:          else
9:              k ← k + 1
10:             B[k] ← B[j]
11:     B.Resize(k)
12:     return (p, B′)
```

Equal elements. If all elements in a bucket are equal, we need to ensure that partitioning terminates and does not degenerate to splitting off one element at a time. To achieve this, elements in the bucket that are on the left of the pivot are moved down when they are *strictly* smaller than the pivot, while elements on the right of the pivot are moved down when they are smaller *or equal*. This way, in expectation half the (non-pivot) elements are moved to the new bucket. Since the pivot itself remains in its bucket, we maintain the invariant that no bucket is ever empty, unless the data structure is completely empty, in which case there is only one empty bucket. Alternatively, one could use dedicated buckets for elements equal to the pivot, but we did not pursue this strategy further. Note that our strategy allows for elements equal to the pivot p_i to be in buckets B_i and B_{i+1}.

Pivot selection. As in quicksort, the performance of the SimdQuickHeap depends on the pivot quality. The median would be the ideal pivot, as it splits the bucket into two equal parts. Experimentally, the median of three random samples offers good performance across all tested workloads. In particular, it is typically a few percent faster than choosing a random pivot or the median of five. Stronger strategies, such as the true median or median-of-medians [9], did not justify their additional overhead.

Rebalancing. The SimdQuickHeap does not suffer from degenerate inputs as much as the original QuickHeap (see Section 2): While the original QuickHeap "bubbles down" elements to insert them, the SimdQuickHeap uses binary search to classify elements, which takes $\mathcal{O}(\log n)$ time no matter the number of pivots. Thus, we do not employ any rebalancing strategies in our implementation. However, we will consider rebalancing in future work, since it still has many potential benefits: When there are $\mathcal{O}(\log n)$ buckets, the binary search has complexity $\mathcal{O}(\log \log n)$ [19]. Alternatively, this allows for a simple linear scan in $\mathcal{O}(\log n)$.

I/O-complexity. When the number of layers is indeed $\mathcal{O}(\log n)$, the SimdQuickHeap has the same amortized I/O-complexity as the QuickHeap: $\mathcal{O}(\frac{1}{B}\log_2 \frac{n}{M})$ per operation, assuming that the first size-B block of each bucket fits in the cache, i.e., $M = \Omega(B \log n)$. Specifically, classifying a pushed element then does not require any memory transfers since the pivots are in memory, and appending it to a block incurs amortized $\mathcal{O}(\frac{1}{B})$ transfers by buffering the last block of elements appended to each bucket. The memory access patterns for the partitioning step are similar to the original QuickHeap, and the original analysis applies.

4.2 Amortized analysis

For simplicity, we also assume that the true median is used as a pivot (as this can be found in linear time), and that no two elements are equal. Using the median of three random elements yields the same bounds for `push` and, in expectation over the random pivot selection, also for `pop`. This can be shown with a similar, more technical, proof that uses the fact that median of three random elements gives a constant-factor split in expectation.

▶ **Theorem 1.** *For the SimdQuickHeap containing n elements, the `push` operation takes $\mathcal{O}(\log n)$ amortized and worst-case time, the `pop` operation takes $\mathcal{O}(1)$ amortized time.*

Proof. The `push` operation can classify the element to insert in $\mathcal{O}(\log n)$ time via a binary search on the pivots, since there are at most $\ell \leq n-1$ pivots. Appending the element to the correct bucket takes constant time, resulting in a total worst-case run time of $\mathcal{O}(\log n)$.

For the amortized run time analysis, we define the *potential function*

$$\Phi := c \cdot \sum_{i=1}^{\ell} (|B_i| + 1) \ln(|B_i| + 1)$$

for some sufficiently large constant c. Let $k_i = |B_i| + 1$ be the *weight* of the bucket B_i. Then, inserting an element into bucket B_i increases the potential by

$$\begin{aligned}
\Delta_{\text{push},i}\Phi &= c(k_i+1)\ln(k_i+1) - ck_i\ln(k_i) \\
&= ck_i \ln\left(1 + k_i^{-1}\right) + c\ln(k_i+1) \\
&\leq c(1 + \ln(k_i+1)) && \text{using } \ln(1+x) \leq x \\
&\leq 2c\ln(n+2) \in \mathcal{O}(\log n).
\end{aligned}$$

Thus, the `push` operation takes $\mathcal{O}(\log n)$ amortized time.

The `pop` operation first finds the median of the bottom bucket of size k_ℓ as pivot and partitions it, resulting in two buckets of equal size $k_\ell/2$. This takes $\mathcal{O}(k_\ell)$ time and changes the potential (ignoring rounding errors) by

$$\Delta_{\text{part}}\Phi = 2\left(c\frac{k_\ell}{2}\ln\left(\frac{k_\ell}{2}\right)\right) - ck_\ell\ln(k_\ell) = -ck_\ell\ln(2).$$

Repeatedly partitioning until the bottom bucket contains only one element changes the potential (again, ignoring rounding errors) by $\Delta_{\text{pop}}\Phi = -c\ln(2)(k_\ell + \frac{k_\ell}{2} + \frac{k_\ell}{4} + \ldots + 2) = -\Omega(ck_\ell)$ and takes $\mathcal{O}(k_\ell)$ time. Thus, with c sufficiently large, the potential can "pay" for the $\mathcal{O}(k_\ell)$ actual cost of the `pop` operation, giving $\mathcal{O}(1)$ amortized time. ◀

4.3 Practical Optimizations

Here, we describe practical optimizations that take advantage of the data layout of the SimdQuickHeap. Let W be the number of elements that fit into one SIMD register. When the number of layers is indeed $\mathcal{O}(\log n)$, the optimizations result in an $\mathcal{O}(\frac{1}{W}\log n)$ algorithm.

Classification. If the number of pivots is small, a linear scan can be faster than a binary search, thanks to better cache efficiency and branch predictions. We therefore employ a SIMD-optimized linear scan unless the number of pivots surpasses some threshold (64 by default), above which we fall back to binary search. In practice, the linear scan is used for all feasible inputs where the number of buckets is in $\mathcal{O}(\log n)$ with this threshold. We compare W consecutive pivots at once with SIMD, resulting in a run time of $\mathcal{O}(\frac{1}{W}\log n)$ when the number of pivots is in $\mathcal{O}(\log n)$.

Partitioning. Multiple partitioning schemes using SIMD have been proposed [12, 63]. We use a straightforward mechanism: We load W values from the bucket and compare them against the pivot at once in $\mathcal{O}(1)$ time. Using AVX2 `permutevar`[3] (~ 3 cycles) or AVX-512 `compress` (~ 6 cycles) instructions, we can then efficiently move the values smaller (or larger) than the pivot to the front of the register. Finally, we append these values to the back of the bucket array. This way, partitioning a bucket of size m takes $\mathcal{O}(m/W)$ time.

In practice, the constant overheads of this partitioning scheme become too large when the bucket is small (of length $\mathcal{O}(W)$). Thus, we stop the iterative partitioning when the bucket size falls below length 16. To be able to delete the minimum in the last bucket efficiently, we keep it sorted in descending order as long as it is smaller than the threshold. Consequently, when a new element is inserted into the last bucket and the bucket size is below the threshold, the element is inserted at the appropriate position using a linear scan.

Further implementation details. We pre-allocate the pivot array with 128 entries, which is enough for most non-degenerate workloads. We do not pre-allocate the bucket arrays but ensure that the capacity is always a multiple of the SIMD width to avoid special cases when reading the last elements. Logically removed buckets are not freed but re-used when new buckets are needed.

5 Experiments

The implementation of the SimdQuickHeap and the evaluations are written in Rust and are available at github:ragnargrootkoerkamp/quickheap. We use an AMD Zen 4 EPYC 9684X (92 cores, 32 KiB L1 data cache, 1 MiB L2 cache, and 32+64 MiB L3 cache for each package of 8 cores) with a 12-channel DDR5 RAM running Rocky Linux 9.4 for the experiments. All experiments are run using a single thread and repeated three times after one warm-up iteration. We report the median of these runs but note that the measurements generally had very little variance. We compile the rust code in release mode with `target-cpu=native` and

[3] See `https://lemire.me/blog/2017/04/10/removing-duplicates-from-lists-quickly/`.

enable full link-time optimization with a single codegen unit to minimize the impact of cross-language function calls. The `C++` code is compiled with `Clang` using the flags `-std=c++17 -O3 -DNDEBUG -march=native -flto`. We further reimplemented some of the benchmarks in `C++` for the `C++` implementations, but measured no difference in performance.

Compared implementations. For SimdQuickHeap, we test both 256-bit AVX2 and 512-bit AVX-512 implementations. We compare against a number of external implementations:

- BinaryHeap: the Rust standard library `std::collections::BinaryHeap`.
- Orx-d-aryHeap: implementation from the `orx_priority_queue` Rust crate [7], the fastest d-ary heap rust implementation we found.
- WeakHeap: from the `weakheap` Rust crate [57].
- RadixHeap: the fastest radix heap we found, in `C++` [4].
- SequenceHeap: the original `C++` implementation [52, 53].
- S3qHeap: the original super-scalar sample queue `C++` implementation [61, 60].
- OriginalQuickHeap: the Randomized QuickHeap implementation of [48].

Further implementations that are less efficient than the ones above are:

- d-aryHeap: implementation from the `dary_heap` Rust crate [43].
- Boost-d-ary heap, BoostFibonacciHeap, BoostPairingHeap, BoostBinomialHeap, BoostSkewHeap: various heap implementations from the `C++` Boost library [10].
- RadixHeapMap: from the `radix_heap` Rust crate [50].
- PairingHeap: from the `pheap` Rust crate [1].
- FibonacciHeap: from the `fibonacci_heap` Rust crate [66].
- ReimplementedQuickHeap: our reimplementation of the QuickHeap.

We further implemented a ScalarQuickHeap variant of the SimdQuickHeap that does not use SIMD instructions and can be instrumented to count internal metrics such as the number of comparisons.

5.1 Synthetic benchmarks

We tested each implementation on several synthetic workloads, for both 32-bit and 64-bit keys. We vary n, the maximum size of the heap, from $2^{10} = 1024$ to $2^{25} \approx 32$ million.

Workloads. The HeapSort workload first generates n random values and then measures how long it takes to push all of them and then pop all of them: $\mathsf{push}^n \circ \mathsf{pop}^n$. In the RandomWiggle workload, we grow the structure to size n via a sequence of $3n$ operations $(\mathsf{push} \circ \mathsf{pop} \circ \mathsf{push})^n$ and then empty it via $(\mathsf{pop} \circ \mathsf{push} \circ \mathsf{pop})^n$. The MonotoneConstantSize workload is initialized by growing the data structure to contain n elements via $(\mathsf{push} \circ \mathsf{pop} \circ \mathsf{push})^n$ as well. Compared to just inserting n values, this ensures that the internal state of each data structure is more randomized. Then, we measure how long it takes to do $10n$ pairs of $\mathsf{pop} \circ \mathsf{push}$ operations. In MonotoneWiggle and MonotoneConstantSize, the pushed value is chosen as the last popped value plus a uniform random constant between 0 and n. In the non-monotone RandomWiggle, the pushed value is a uniform random (32 or 64-bit) integer. We note that this non-monotone case is degenerate, in that most values are pushed to the front of the queue.

For each workload, the number N of `push` operations is the same as the number of pop operations: $N = n$ for HeapSort, $N = 10n$ for MonotoneConstantSize, and $N = 3n$ for MonotoneWiggle and RandomWiggle. We report the average time *per element*, which is the total running time divided by N.

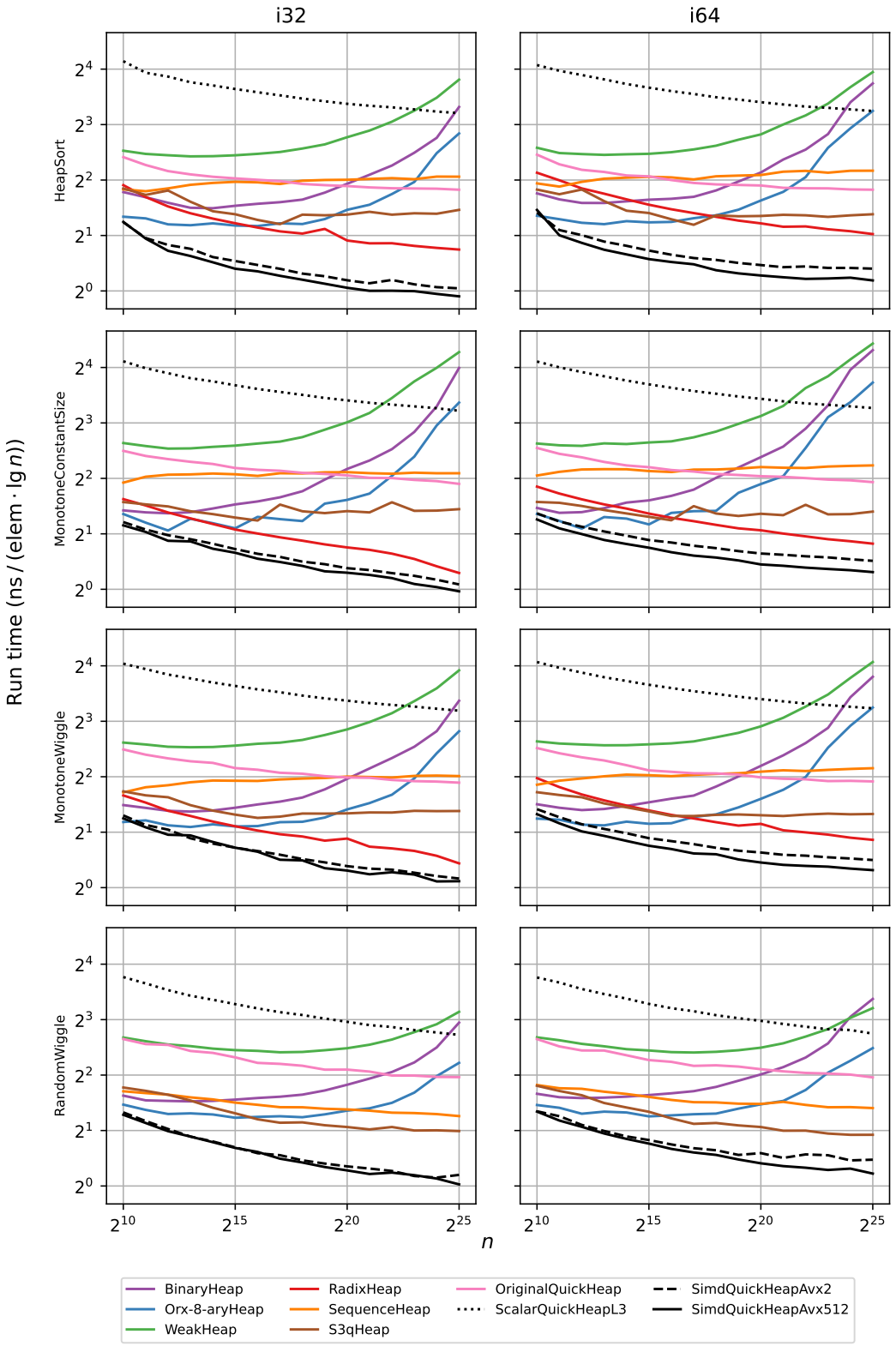

Figure 4 Log-log plots showing for a variety of workloads (rows, see Section 5.1), `i32` and `i64` inputs (columns), and increasing maximum number of elements stored in the data structure (x-axis), the time in nanoseconds to push and pop each element, divided by $\log_2 n$. That is, the time is relative to the $\Omega(\log_2 n)$ lower bound per operation.

	Time (ns/elem)				L2 misses/elem
Implementation	$n = 2^{10}$	2^{15}	2^{20}	2^{25}	$n = 2^{25}$
BinaryHeap	27.4	45.8	104.2	439.6	13.26
Orx-8-ary heap	25.7	33.2	73.4	297.7	8.27
WeakHeap	61.7	92.8	175.9	482.0	20.12
RadixHeap	36.6	39.0	42.6	44.9	0.07
SequenceHeap	41.7	65.5	92.0	117.2	0.43
S3qHeap	30.1	37.1	53.2	68.1	0.35
OriginalQuickHeap	57.7	68.5	82.3	97.2	0.07
ScalarQuickHeap	173.3	194.7	215.4	242.3	0.64
SimdQuickHeapAvx2	27.1	29.0	32.3	35.6	0.09
SimdQuickHeapAvx512	23.5	**26.3**	**29.4**	**32.2**	0.11
Orx-4-ary heap	**20.3**	35.3	64.1	348.5	8.03
Boost-4-aryHeap	25.7	43.6	90.3	419.8	8.07
4-ary heap	23.8	43.5	73.8	320.5	8.08
8-ary heap	46.2	79.7	122.6	258.8	21.01
RadixHeapMap	44.1	45.9	48.5	51.2	0.05
PairingHeap	179.3	300.4	750.3	1 963.7	26.25
FibonacciHeap	389.2	609.0	1 270.5	2 313.3	82.47
BoostFibonacciHeap	122.4	216.5	719.2	2 027.9	54.30
BoostPairingHeap	260.5	409.8	1 112.4	2 981.7	52.29
BoostBinomialHeap	465.5	739.3	1 525.9	2 671.6	34.02
BoostSkewHeap	103.4	229.0	764.1	2 754.2	35.84
ReimplementedQuickHeap	58.7	76.9	95.8	115.9	**0.04**

Table 1 Average time in nanoseconds for pushing and popping each element for all implementations on the `MonotoneConstantSize` workload with `i64` keys, for increasing heap size n. The last column gives the number of L2 cache misses per element at the largest size $n = 2^{25}$, excluding cache lines prefetched by the hardware prefetcher. The implementations above the rule are shown in Figure 4. The best value in each column is set in bold.

Results. Figure 4 shows the average time per element normalized by $\log_2 n$ for the best performing priority queue implementations on various workloads. The normalization results in the time per *fundamental operation*, since $\log_2$ is the minimum number of comparisons needed per element when the queue contains n elements.

The absolute time per element for all implementations on the `MonotoneConstantSize` workload is given in Table 1. We see that the BinaryHeap and d-aryHeap slow down significantly as the data grows beyond the size of the CPU caches. The d-aryHeap is consistently around 1.4× as fast as the BinaryHeap. For large n, the WeakHeap is surprisingly competitive with the BinaryHeap.

The RadixHeap benefits from high cache-locality and gets faster rather than slower (relative to the lower bound) as n grows. It is faster for 32-bit than for 64-bit data. Similarly, as predicted by the theory, the SequenceHeap is I/O-efficient and does not slow down on larger inputs. The more modern and optimized S3qHeap is up to 2× faster for 64-bit data.

Our reimplementation of the original QuickHeap is on-par with the sequence heap, while the scalar version of our new heap is 3× *slower*, possibly because the original QuickHeap uses more efficient in-place partitioning. The SimdQuickHeap is around 8× as fast as the

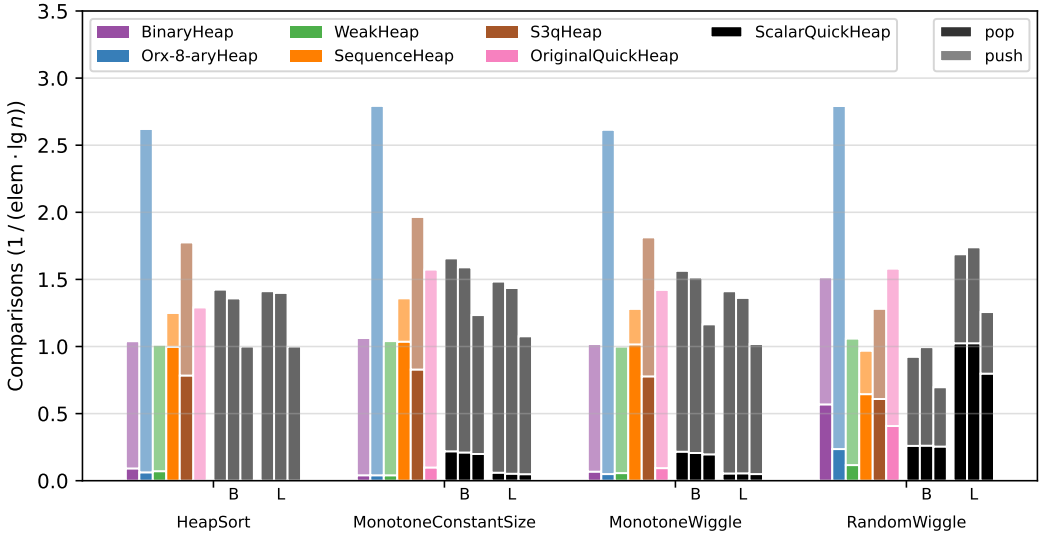

Figure 5 The number of comparisons per operation relative to the $\lg_2 n$ lower bound in different workloads. The number of comparisons during push operations is highlighted on the bottom, with the number of comparisons during pop operations shown on top. The SimdQuickHeap variants differ in using binary search (B) or linear search (L), and the pivot selection (in order: random, median of 3, true median oracle).

scalar version. Like the other engineered heaps, it gets faster relative to the lower bound as n increases, indicating that the constant overhead of each `push` and `pop` is relatively large compared to the $\log_2 n$ pivot steps required for each element. The AVX-512 version is up to 1.2× as fast as the AVX2 version, and up to 2× as fast as the S3qHeap. Surprisingly, it also significantly outperforms the RadixHeap in all tested cases.

The SimdQuickHeap reaches below $\log_2 n$ nanoseconds per pop ∘ push pair. This means that it requires around 1 nanosecond for each comparison in the lower bound. Since each nanosecond corresponds to 3.7 clock cycles, each of which can contain multiple SIMD instructions on 8 words, the overhead of moving data around is still very large compared to just the comparisons.

Number of comparisons. On the MonotoneConstantSize workload (Figure 5), the BinaryHeap needs less than $1.1 \log_2 n$ comparisons per element, while the 8-aryHeap needs around $2.7 \log_2 n$ comparisons. The BinaryHeap needs more comparisons on degenerate input, while the WeakHeap consistently needs $1.0 \log_2 n$. The ScalarQuickHeap needs $1.7 \log_2 n$ comparisons when using random pivots in combination with a linear scan over the pivots to insert elements. This decreases to $1.2 \log_2 n$ when using an oracle that gives the exact median for free, showing that there is some room for improvement by using a more accurate partitioning scheme.

Running time distribution. When running SimdQuickHeap with $n = 2^{25}$ on the MonotoneConstantSize workload, 21% of the time is spent pushing elements, with 14% (of the total) scanning the list of pivots to find the right layer to append to. The other 78% of time is spent on pop: 59% partitioning the input, 11% finding the position of the smallest element in the bottom bucket, and 3% removing and returning this element.

Name	Description	$\lvert V\rvert$ ($\times 10^6$)	$\lvert E\rvert$ ($\times 10^6$)	max degree	median weight	max weight	N_{D} ($\times 10^6$)	N_{JP} ($\times 10^6$)
CAL	Rd. California	1	4	8	3115	54k	2	2
CTR	Rd. Central USA	14	34	9	3467	54k	15	17
GER	Rd. Germany	20	41	9	39	62k	21	22
USA	Rd. USA	23	58	9	3473	92k	26	29
RHG_{20}	Rand. hyperbolic	1	10	90k	230	999	2	5
RHG_{22}	Rand. hyperbolic	4	41	941k	230	999	7	21
RHG_{24}	Rand. hyperbolic	16	159	595k	232	999	28	80

Table 2 Number of edges and vertices of the graph instances that have been used for benchmarking, as well as the median and maximal edge weight. N_{D} and N_{JP} are the numbers of processed queue elements during Dijkstra's and Jarník–Prim's algorithm, respectively.

Memory usage. For most workloads, the total capacity of the vectors allocated by the SimdQuickHeap is around 5× the space required just to hold the elements. For the MonotoneConstantSize workload, this factor grows to around 11. The increased memory consumption is likely due to the higher number of operations ($10n$ vs. $\leq 3n$), so that the data structure has more time to accumulate large temporary buckets due to bad pivots. In comparison, binary heaps and the original QuickHeap can be implemented on top of a growing vector, resulting in an overhead of just the geometric growing factor (typically 2). Possible mitigations to lower the memory consumption would be to periodically shrink vectors with excessive capacity or to use a more memory-efficient data structure like a list of (reusable) blocks.

5.2 Graph benchmarks

Two textbook graph algorithms that require a priority queue are Dijkstra's algorithm [29] to compute shortest paths on a weighted directed graph with non-negative edge weights, and Jarník–Prim's algorithm [41, 51] to compute minimum-spanning-trees on weighted, undirected graphs. Keys are non-decreasing for Dijkstra's algorithm, while keys may decrease for Jarník–Prim's algorithm, thus excluding the radix heap. In this section, we compare running times of the two algorithms on multiple graph instances using different priority queues.

Setup. We use 32-bit identifiers for vertices/edges and 32-bit non-negative edge weights. We pack these into a single 64-bit unsigned integer that we use as elements for the priority queues. For Dijkstra's algorithm, we re-insert a vertex when a new shorter path is found and discard stale versions of the vertex when popped. We do not implement a variant that updates vertices instead of re-inserting them since re-inserting generally results in better performance [23] and no priority queue implementation that supports `decreaseKey` offers competitive performance in the synthetic benchmarks (see Table 1).

Graph types. We used different types of graphs to benchmark the performance of the SimdQuickHeap (Table 2). We test on four road networks of varying sizes with edge weights representing travel times, all but the German[4] network downloaded from 9th DIMACS

[4] `https://i11www.iti.kit.edu/resources/roadgraphs.php`

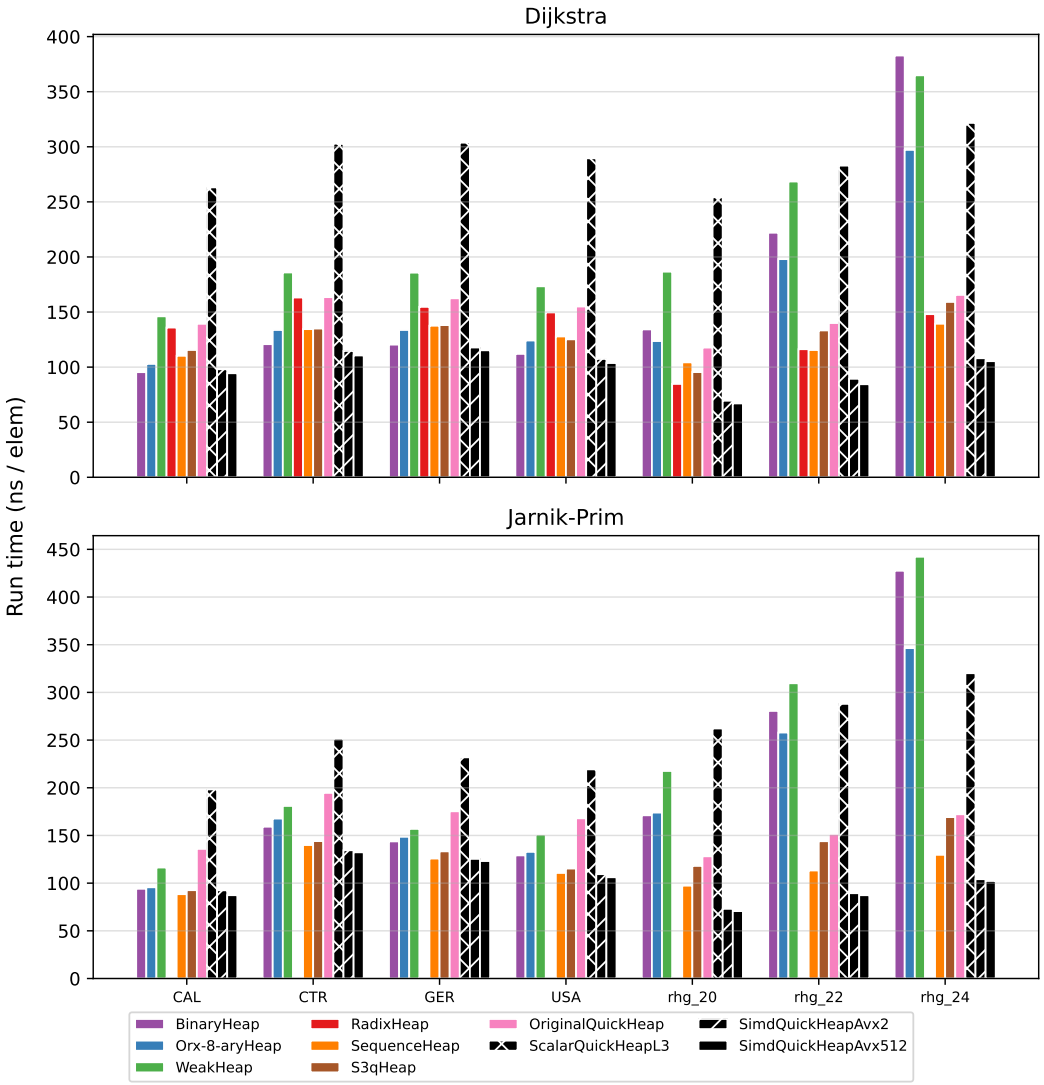

Figure 6 Time per processed queue element for the different heaps on multiple graph instances. The radix heap requires non-decreasing keys and is excluded from the Jarník–Prim benchmark.

implementation challenge[5]. We also test on random hyperbolic graphs generated with KaGen [38] with 2^{20}, 2^{22}, and 2^{24} nodes, an average degree of 16 and a power law exponent of $\gamma = 2.3$. Edge weights represent the hyperbolic distance.

Results. We report the average running time per processed queue element, i.e., the number of elements that are pushed into and subsequently popped from the queue. In Figure 6, we see that the SimdQuickHeap is the fastest on all graphs for both Dijkstra and Jarník–Prim workloads. Speedups on road networks are small since the cache misses involved with traversing the graph are likely the bottleneck. On the hyperbolic graphs, we achieve speedup factors of up to 4× compared to the binary heap.

[5] `http://www.diag.uniroma1.it/challenge9/download.shtml`

6 Conclusion

Due to its conceptually simple design, the SimdQuickHeap allows an efficient implementation using SIMD instructions that has both a good theoretical complexity and great performance in practice. As predicted by the improved I/O-complexity, the SimdQuickHeap vastly outperforms the binary and d-ary heap, as well as the weak heap. This new design gives a 4× speedup over I/O-efficient structures such as the sequence heap (2000) [52] and QuickHeap (2010) [47], and a 1.4× to 2.8× speedup over the much more recent superscalar sample queue [61]. Maybe more surprisingly, the SimdQuickHeap is also up to 1.7× as fast as the non-comparison-based radix heap.

Future work. Future work will be to implement the recently introduced rebalancing strategies of [19] to limit the number of buckets, and to evaluate if and how much they affect performance in both (currently) degenerate and non-degenerate cases. This will also make the running time and I/O-complexity hold unconditionally. Furthermore, the current $\mathcal{O}(\frac{1}{B}\log_2\frac{n}{M})$ I/O-complexity is missing the more efficient $\log_{M/B}$ that is obtained by multi-way merging/splitting in the sequence heap and superscalar sample queue, and so the question is whether the QuickHeap naturally extends to multi-way splitting.

Another option is to make a SIMD-optimized version of the radix-heap, as it has a similar structure with exponentially growing buckets.

From the engineering side, optimizing the partitioning might provide further gains. A drawback of the SimdQuickHeap is the larger memory usage compared to the original QuickHeap and simple binary heaps. Using a list of reusable blocks rather than single vectors could reduce memory usage. Furthermore, the implementation could be expanded to support key-value pairs that are both 64 bits, bulk-insertions, and multi-threading.

References

1 1crcbl. Pairing heap, 2021. URL: `https://github.com/1crcbl/pheap-rs`.

2 Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988. `doi:10.1145/48529.48535`.

3 Ravindra K. Ahuja, Kurt Mehlhorn, James Orlin, and Robert E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 37(2):213–223, April 1990. `doi:10.1145/77600.77615`.

4 Takuya Akiba. Radix-heap, 2015. URL: `https://github.com/iwiwi/radix-heap`.

5 Lars Arge, Michael A. Bender, Erik D. Demaine, Bryan Holland-Minkley, and J. Ian Munro. An optimal cache-oblivious priority queue and its application to graph algorithms. *SIAM Journal on Computing*, 36(6):1672–1695, January 2007. `doi:10.1137/s0097539703428324`.

6 Lars Arge, Michael A. Bender, Erik D. Demaine, Bryan Holland-Minkley, and J. Ian Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, STOC02, pages 268–276. ACM, May 2002. `doi:10.1145/509907.509950`.

7 Ugur Arikan. Orx-priority-queue: Priority queue traits and efficient d-ary heap implementations, 2023. URL: `https://github.com/orxfun/orx-priority-queue/`.

8 Maxim Babenko, Ignat Kolesnichenko, and Ivan Smirnov. Cascade heap: Towards time-optimal extractions. In *Computer Science – Theory and Applications*, pages 62–70. Springer International Publishing, 2017. `doi:10.1007/978-3-319-58747-9_8`.

9 Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, August 1973. `doi:10.1016/S0022-0000(73)80033-9`.

10 Boost. Boost C++ libraries, 2026. URL: https://www.boost.org/.

11 Otakar Borůvka. O jistém problému minimálním (On a certain problem of minimization). *Práce Moravské přírodovědecké společnosti, sv. III, spis 3, 1926*, pages 36–58, 1926. URL: https://dml.cz/handle/10338.dmlcz/500114.

12 Berenger Bramas. A Novel Hybrid Quicksort Algorithm Vectorized using AVX-512 on Intel Skylake. *International Journal of Advanced Computer Science and Applications (ijacsa)*, 8(10), October 2017. doi:10.14569/IJACSA.2017.081044.

13 Klaus Brengel, Andreas Crauser, Paolo Ferragina, and Ulrich Meyer. An experimental study of priority queues in external memory. In *Algorithm Engineering*, pages 345–359. Springer Berlin Heidelberg, 1999. doi:10.1007/3-540-48318-7_27.

14 Gerth Stølting Brodal. Worst-case efficient priority queues. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '96, pages 52–58. Society for Industrial and Applied Mathematics, January 1996. URL: https://dl.acm.org/doi/10.5555/313852.313883.

15 Gerth Stølting Brodal. A Survey on Priority Queues. In *Space-Efficient Data Structures, Streams, and Algorithms: Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, pages 150–163. Springer, 2013. doi:10.1007/978-3-642-40273-9_11.

16 Gerth Stølting Brodal and Rolf Fagerberg. Funnel Heap-A Cache Oblivious Priority Queue. In *Algorithms and Computation*, pages 219–228. Springer, 2002. doi:10.1007/3-540-36136-7_20.

17 Gerth Stølting Brodal, Rolf Fagerberg, Ulrich Meyer, and Norbert Zeh. Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths. In *Algorithm Theory - SWAT 2004*, pages 480–492. Springer Berlin Heidelberg, 2004. doi:10.1007/978-3-540-27810-8_41.

18 Gerth Stølting Brodal, Michael T. Goodrich, John Iacono, Jared Lo, Ulrich Meyer, Victor Pagan, Nodari Sitchinava, and Rolf Svenning. External-Memory Priority Queues with Optimal Insertions. In *33rd Annual European Symposium on Algorithms (ESA 2025)*, volume 351 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.ESA.2025.5.

19 Gerth Stølting Brodal, John Iacono, Casper Moldrup Rysgaard, and Sebastian Wild. Partition-based simple heaps. *arXiv*, 2026. doi:10.48550/ARXIV.2603.01206.

20 Mark R. Brown. The complexity of priority queue maintenance. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing - STOC '77*, Stoc '77, pages 42–48. ACM Press, 1977. doi:10.1145/800105.803394.

21 Mark R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM Journal on Computing*, 7(3):298–319, August 1978. doi:10.1137/0207026.

22 Asger Bruun, Stefan Edelkamp, Jyrki Katajainen, and Jens Rasmussen. Policy-based benchmarking of weak heaps and their relatives,. In *Experimental Algorithms*, pages 424–435. Springer Berlin Heidelberg, 2010. doi:10.1007/978-3-642-13193-6_36.

23 Mo Chen, Rezaul Alam Chowdhury, Vijaya Ramachandran, David Lan Roche, and Lingling Tong. Priority Queues and Dijkstra's Algorithm, 2007. URL: https://www.cs.utexas.edu/ftp/techreports/tr07-54.pdf.

24 Rezaul A. Chowdhury and Vijaya Ramachandran. Cache-oblivious buffer heap and cache-efficient computation of shortest paths in graphs. *ACM Transactions on Algorithms*, 14(1):1–33, January 2018. doi:10.1145/3147172.

25 Rezaul Alam Chowdhury and Vijaya Ramachandran. Cache-oblivious shortest paths in graphs using buffer heap. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA04, pages 245–254. ACM, June 2004. doi:10.1145/1007912.1007949.

26 Jonas Costa, Lucas Castro, and Rosiane de Freitas. Exploring monotone priority queues for Dijkstra optimization. *RAIRO - Operations Research*, 59(5):2419–2436, September 2025. doi:10.1051/ro/2025082.

27 Eric V. Denardo and Bennett L. Fox. Shortest-route methods: 1. Reaching, pruning, and buckets. *Operations Research*, 27(1):161–186, February 1979. doi:10.1287/opre.27.1.161.

28 Robert B. Dial. Algorithm 360: Shortest-path forest with topological ordering [H]. *Communications of the ACM*, 12(11):632–633, November 1969. doi:10.1145/363269.363610.

29 Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, December 1959. doi:10.1007/bf01386390.

30 Ronald D. Dutton. Weak-heap sort. *Bit Numerical Mathematics*, 33(3):372–381, September 1993. doi:10.1007/bf01990520.

31 Stefan Edelkamp, Amr Elmasry, and Jyrki Katajainen. The weak-heap data structure: Variants and applications. *Journal of Discrete Algorithms*, 16:187–205, October 2012. doi:10.1016/j.jda.2012.04.010.

32 Stefan Edelkamp, Amr Elmasry, and Jyrki Katajainen. Weak heaps engineered. *Journal of Discrete Algorithms*, 23:83–97, November 2013. doi:10.1016/j.jda.2013.07.002.

33 Kasper Eenberg, Kasper Green Larsen, and Huacheng Yu. DecreaseKeys are expensive for external memory priority queues. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, Stoc '17, pages 1081–1093. ACM, June 2017. doi:10.1145/3055399.3055437.

34 M. L. Fredman and D. E. Willard. BLASTING through the information theoretic barrier with FUSION TREES. In *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing - STOC '90*, Stoc '90, pages 1–7. ACM Press, 1990. doi:10.1145/100216.100217.

35 Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica. An International Journal in Computer Science*, 1(1–4):111–129, November 1986. doi:10.1007/bf01840439.

36 Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, July 1987. doi:10.1145/28869.28874.

37 Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms*, 8(1):1–22, January 2012. doi:10.1145/2071379.2071383.

38 Daniel Funke, Sebastian Lamm, Ulrich Meyer, Manuel Penschuck, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. Communication-free massively distributed graph generation. *Journal of Parallel and Distributed Computing*, 131:200–217, September 2019. doi:10.1016/j.jpdc.2019.03.011.

39 C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, July 1961. doi:10.1145/366622.366644.

40 John Iacono and Özgür Özkan. Why some heaps support constant-amortized-time decrease-key operations, and others do not. In *Automata, Languages, and Programming*, pages 637–649. Springer Berlin Heidelberg, 2014. doi:10.1007/978-3-662-43948-7_53.

41 Vojtěch Jarník. O jistém problému minimálním (On a certain problem of minimization). *Práce Moravské přírodovědecké společnosti, sv. VI., spis 4, 1930*, pages 57–63, 1930. URL: https://dml.cz//handle/10338.dmlcz/500726.

42 Shunhua Jiang and Kasper Green Larsen. A faster external memory priority queue with DecreaseKeys. *arXiv*, 2018. doi:10.48550/ARXIV.1806.07598.

43 Donald B. Johnson. Priority queues with update and finding minimum spanning trees. *Information Processing Letters*, 4(3), December 1975. doi:10.1016/0020-0190(75)90001-0.

44 Poul-Henning Kamp. You're doing it wrong. *Communications of the ACM*, 53(7):55–59, July 2010. doi:10.1145/1785414.1785434.

45 Daniel H. Larkin, Siddhartha Sen, and Robert E. Tarjan. A back-to-basics empirical study of priority queues. In *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 61–72. Society for Industrial and Applied Mathematics, December 2013. doi:10.1137/1.9781611973198.7.

46 Christian Loeffeld. The logarithmic funnel heap: An efficient priority queue for extremely large sets, 2023. doi:10.48550/ARXIV.1705.10648.

47 Gonzalo Navarro and Rodrigo Paredes. On sorting, heaps, and minimum spanning trees. *Algorithmica. An International Journal in Computer Science*, 57(4):585–620, March 2010. doi:10.1007/s00453-010-9400-6.

48 Gonzalo Navarro, Rodrigo Paredes, Patricio V. Poblete, and Peter Sanders. Stronger quickheaps. *International Journal of Foundations of Computer Science*, 22(04):945–969, June 2011. doi:10.1142/s0129054111008507.

49 Rodrigo Paredes and Gonzalo Navarro. Optimal Incremental Sorting. In *2006 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, Proceedings, pages 171–182. Society for Industrial and Applied Mathematics, January 2006. doi:10.1137/1.9781611972863.16.

50 Mike Pedersen. Radix-heap: Fast monotone priority queues, 2016. URL: https://github.com/mpdn/radix-heap.

51 R. C. Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, 1957. doi:10.1002/j.1538-7305.1957.tb01515.x.

52 Peter Sanders. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics*, 5:7, December 2000. doi:10.1145/351827.384249.

53 Peter Sanders. Sequence heap, 2000. URL: https://github.com/raphinesse/SequenceHeap.

54 Peter Sanders and Sebastian Winkel. Super scalar sample sort. In *Algorithms – ESA 2004*, pages 784–796. Springer Berlin Heidelberg, 2004. doi:10.1007/978-3-540-30140-0_69.

55 Corwin Sinnamon and Robert E. Tarjan. A tight analysis of slim heaps and smooth heaps. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 549–567. Society for Industrial and Applied Mathematics, January 2023. doi:10.1137/1.9781611977554.ch24.

56 Corwin Sinnamon and Robert E. Tarjan. Efficiency of self-adjusting heaps. *ACM Transactions on Algorithms*, 21(4):1–39, September 2025. doi:10.1145/3708989.

57 Egor Starovoitov. Weak-heap, 2022. URL: https://github.com/starovoid/weakheap.

58 Mikkel Thorup. Equivalence between priority queues and sorting. *Journal of the ACM*, 54(6):28, December 2007. doi:10.1145/1314690.1314692.

59 P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *16th Annual Symposium on Foundations of Computer Science (Sfcs 1975)*. IEEE, October 1975. doi:10.1109/sfcs.1975.26.

60 Raphael von der Grün. Superscalar sample queue, 2021. URL: https://github.com/raphinesse/s3q.

61 Raphael von der Grün. Superscalar sample queue: Engineering a distribution-based priority queue. Master's thesis, Karlsruhe Institute of Technology, 2021. URL: https://ae.iti.kit.edu/english/4296.php.

62 Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, April 1978. doi:10.1145/359460.359478.

63 Jan Wassenberg, Mark Blacher, Joachim Giesen, and Peter Sanders. Vectorized and performance-portable quicksort. *Software: Practice and Experience*, 52(12):2684–2699, 2022. doi:10.1002/spe.3142.

64 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space Θ(N). *Information Processing Letters*, 17(2):81–84, August 1983. doi:10.1016/0020-0190(83)90075-3.

65 J.W.J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, June 1964. doi:10.1145/512274.3734138.

66 xvi-xv-xii-ix-xxii-ix-xiv. Fibonacci heap in rust, 2026. URL: https://github.com/xvi-xv-xii-ix-xxii-ix-xiv/fibonacci_heap.