Weak heaps and friends

Edelkamp, Stefan; Elmasry, Amr; Katajainen, Jyrki; Weiß, Armin

Published in:
Combinatorial Algorithms

DOI:
10.1007/978-3-642-45278-9_1

Publication date:
2013

Document Version
Peer reviewed version

Citation for published version (APA):
Edelkamp, S., Elmasry, A., Katajainen, J., & Weiß, A. (2013). Weak heaps and friends: recent developments. In T. Lecroq, & L. Mouchard (Eds.), Combinatorial Algorithms: 24th International Workshop, IWOCA 2013, Rouen, France, July 10-12, 2013, Revised Selected Papers (pp. 1-6). Springer. Lecture notes in computer science, Vol.. 8288 https://doi.org/10.1007/978-3-642-45278-9_1

Weak Heaps and Friends: Recent Developments

Stefan Edelkamp[1], Amr Elmasry[2], Jyrki Katajainen[3], and Armin Weiß[4]

[1] Faculty 3—Mathematics and Computer Science, University of Bremen
PO Box 330 440, 28334 Bremen, Germany
[2] Department of Computer Engineering and Systems, Alexandria University
Alexandria 21544, Egypt
[3] Department of Computer Science, University of Copenhagen
Universitetsparken 5, 2100 Copenhagen East, Denmark
[4] Institute for Formal Methods in Computer Science, University of Stuttgart
Universitätstraße 38, 70569 Stuttgart, Germany

Abstract. A weak heap is a variant of a binary heap where, for each node, the heap ordering is enforced only for one of its two children. In 1993, Dutton showed that this data structure yields a simple worst-case-efficient sorting algorithm. In this paper we review the refinements proposed to the basic data structure that improve the efficiency even further. Ultimately, *minimum* and *insert* operations are supported in $O(1)$ worst-case time and *extract-min* operation in $O(\lg n)$ worst-case time involving at most $\lg n + O(1)$ element comparisons. In addition, we look at several applications of weak heaps. This encompasses the creation of a sorting index and the use of a weak heap as a tournament tree leading to a sorting algorithm that is close to optimal in terms of the number of element comparisons performed. By supporting *insert* operation in $O(1)$ amortized time, the weak-heap data structure becomes a valuable tool in adaptive sorting leading to an algorithm that is constant-factor optimal with respect to several measures of disorder. Also, a weak heap can be used as an intermediate step in an efficient construction of binary heaps. For graph search and network optimization, a weak-heap variant, which allows some of the nodes to violate the weak-heap ordering, is known to be provably better than a Fibonacci heap.

1 Weak Heaps

In its elementary form, a priority queue is a data structure that stores a collection of elements and supports the operations *construct*, *minimum*, *insert*, and *extract-min* [4]. In applications where this set of operations is sufficient, the priority queue that the users would select is a binary heap [29] or a weak heap [7]. Both of these data structures are known to perform well, and in typical cases the difference in performance is marginal. Most library implementations are based on a binary heap. However, one reason why a user might vote for a weak heap over a binary heap is that weak heaps are known to perform less element comparisons in the worst case: Comparing binary heaps vs. weak heaps for *construct* we have $2n$ vs. $n - 1$ and for *extract-min* we have $2\lceil \lg n \rceil$ vs. $\lceil \lg n \rceil$

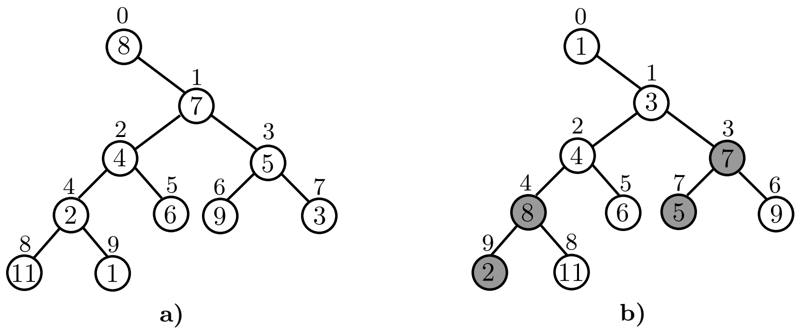

Fig. 1. a) An input of 10 integers and **b)** a weak heap constructed by the standard algorithm (reverse bits are set for grey nodes, small numbers above the nodes are the actual array indices.) Source: [11, 12]

element comparisons, where n denotes the number of elements stored in the data structure prior to the operation. Moreover, *minimum* and *insert* have matching complexities 0 and $\lceil \lg n \rceil$ element comparisons, respectively.

More formally, a *weak heap* (see Fig. 1) is a binary tree that has the following properties:

1. The root of the entire tree has no left child.
2. Except for the root, the nodes that have at most one child are at the last two levels only. Leaves at the last level can be scattered, i.e. the last level is not necessarily filled from left to right.
3. Each node stores an element that is smaller than or equal to every element stored in its right subtree.

From the first two properties we can deduce that the height of a weak heap that has n elements is $\lceil \lg n \rceil + 1$. The third property is called *weak-heap ordering* or *half-tree ordering*. In particular, this property does not enforce any relation between an element in a node and those stored in the left subtree of that node. If perfectly balanced, weak heaps resemble heap-ordered binomial trees [27]. Binomial-tree parents are distinguished ancestors in the weak-heap setting.

In an array-based implementation, besides the element array a, an array r of *reverse bits* is used, i.e. $r_i \in \{0, 1\}$ for $i \in \{0, \ldots, n-1\}$. The array index of the left child of a_i is $2i + r_i$, the array index of the right child is $2i + 1 - r_i$, and the array index of the parent is $\lfloor i/2 \rfloor$ (assuming that $i \neq 0$). Using the fact that the indices of the left and right children of a_i are reversed when flipping r_i, subtrees can be swapped in constant time by setting $r_i \leftarrow 1 - r_i$. In a compact representation of a bit array on a w-bit computer $\lceil n/w \rceil$ words are used.

In a pointer-based implementation, the bits are no more needed, but the children and the parent of a node are accessed by following pointers, and the children are reversed by swapping pointers. Pointer-based weak heaps can be used to implement addressable priority queues, which also support *delete* and *decrease* operations [1, 9, 10].

2 Constant-Factor-Optimal Sorting

Dutton [7] showed that to sort n elements WeakHeapSort, a Heapsort variant that uses a weak heap, requires at most $n\lceil \lg n\rceil - 2^{\lceil \lg n\rceil} + n - 1 \leq n \lg n + 0.089n$ element comparisons in the worst case. Algorithms, for which the constant in the leading term in the bound expressing the number of element comparisons performed is the best possible, are called *constant-factor optimal*. Since the early attempts [23], many people have tried to close the gap to the lower bound and to derive constant-factor-optimal algorithms for which the number of primitive operations performed is in $O(n \lg n)$. Other members in the exclusive group of constant-factor-optimal Heapsort algorithms include UltimateHeapsort [21] and MDRHeapsort [25] (analysed by Wegener [28]); the former is fully in-place whereas the latter needs $2n$ extra bits, but for the in-place algorithm the constant α in the bound $n \lg n + \alpha n$ is larger. Knuth [23] showed that MergeInsertion is a sorting algorithm which performs at most $n \lg n - (3 - \lg 3)n + n(\phi + 1 - 2^{\phi}) + O(\lg n)$ element comparisons, where $3 - \lg 3 \approx 1.41$ and $0 \leq \phi \leq 1$. However, when implemented with an array, a quadratic number of element moves may be needed to accomplish the task.

Edelkamp and Wegener [15] gave the worst-case and best-case examples for WeakHeapSort, which match the proved upper bounds. Experimentally they showed that in the average case the number of element comparisons performed is about $n \lg n + \beta n$ with $\beta \in [-0.46, -0.42]$. Edelkamp and Stiegeler [14] showed that for sorting indices (as required in many database applications) WeakHeapSort can be implemented so that it performs at most $n \lg n - 0.91n$ element comparisons, which is only off by about $0.53n$ from the lower bound [23].

Recently, in [16], the idea of QuickHeapsort [2, 5] was generalized to the notion of QuickXsort: Given some black-box sorting algorithm X, QuickXsort can be used to speed X up provided that X satisfies certain natural conditions. QuickWeakHeapsort and QuickMergesort were described as two examples of this construction. QuickMergesort performs $n \lg n - 1.26n + o(n)$ element comparisons on the average and the worst case of $n \lg n + O(n)$ element comparisons can be achieved without affecting the average case. Furthermore, a weak-heap tournament tree yields an efficient implementation of MergeInsertion for small values of n. Taking it as a base case for QuickMergesort, a worst-case-efficient constant-factor-optimal sorting algorithm can be established, which performs $n \lg n - 1.3999n + o(n)$ element comparisons on an average. QuickMergesort with constant-size base cases showed the best performance [16]: When sorting integers it was only 15% slower than Introsort [26] taken from a C++ standard-library implementation.

In [8], two variations of weak heaps were described: The first one uses an array-based weak heap and the other, a *weak queue*, is a collection of pointer-based perfect weak heaps. For both, *insert* requires $O(1)$ amortized time and *extract-min* $O(\lg n)$ worst-case time including at most $\lg n + O(1)$ element comparisons, n being the number of elements stored. In both, the main idea is to temporarily store the inserted elements in a buffer and, once it becomes full, to move the buffer elements to the main queue using an efficient bulk-insertion procedure. By employing the new priority queues in AdaptiveHeapsort [24], the resulting algorithm was shown to be constant-factor optimal with respect to several measures of disorder. Unlike some previous constant-factor-optimal adaptive sorting algorithms [17–19], AdaptiveHeapsort relying on the developed priority queues is practically workable.

3 Relaxed Weak Heaps and Relaxed Weak Queues

In [1], experimental results on the practical efficiency of three addressable priority queues were reported. The data structures considered were a weak heap, a weak queue, and a *run-relaxed weak queue* that extends a weak queue by allowing some nodes to violate the half-heap ordering; a run-relaxed weak queue is a variant of a run-relaxed heap [6] that uses binary trees instead of multiary trees. All the studied data structures support *delete* and *extract-min* in logarithmic worst-case time. A weak queue reduces the worst-case running time of *insert* to $O(1)$, and a run-relaxed weak queue that of both *insert* and *decrease* to $O(1)$. As competitors to these structures, a binary heap, a Fibonacci heap, and a pairing heap were considered. Generic programming techniques were heavily used in the code development. For benchmarking purposes several component frameworks were developed that could be instantiated with different policies.

In [9, 10], two new relaxed priority-queue structures, a *run-relaxed weak heap* and a *rank-relaxed weak heap*, were introduced. The relaxation idea originates from [6], but here it is applied in a single-tree context. In contrast to run relaxation, rank relaxation provides good amortized performance. Since rank-relaxed weak heaps are simpler and faster, they are better suited for network-optimization algorithms. For a request sequence of n *insert*, m *decrease*, and n *extract-min* operations, it can be shown that a rank-relaxed weak heap performs at most $2m + 1.5n\lceil\lg n\rceil$ element comparisons [9, 10]. When considering the same sequence of operations, this bound improves over the best bounds known for different variants of a Fibonacci heap, which may require $2m + 2.89n\lceil\lg n\rceil$ element comparisons in the worst case.

4 Heap Construction and Optimal In-Place Heaps

In [11, 12], different options for constructing a weak heap were studied. Starting from a straightforward algorithm, the authors ended up with a catalogue of algorithms that optimize the standard algorithm in different ways. As the optimization criteria, it was considered how to reduce the number of instructions, branch mispredictions, cache misses, and element moves. An approach relying on a non-standard memory layout was fastest, but the outcome is a weak heap where the element order is shuffled.

A binary heap can be built on top of a previously constructed a navigation pile [22] with at most $0.625n$ element comparisons. In [3], it was shown how this transformation can be used to build binary heaps in-place by performing at most $1.625n + o(n)$ element comparisons. The construction of binary heaps via weak heaps is equally efficient, but this transformation requires a slightly higher number of element moves.

In contrast to binary heaps, n repeated *insert* operations (starting from an empty structure) can be shown to require at most $3.5n + O(\lg^2 n)$ element comparisons [11, 12] (but $\Theta(n \lg n)$ time in the worst case). In addition, with constant memory overhead, $O(1)$ amortized time per *insert* can be improved to $O(1)$ worst-case time [12], while preserving $O(1)$ worst-case time for *minimum* and $O(\lg n)$ worst-case time with at most $\lg n + O(1)$ element comparisons for *extract-min*. This result was previously achieved only for more complicated structures like multipartite priority queues [19]. Still, none of the known constant-factor-optimal worst-case solutions can be claimed to be practical.

As a culmination, in [13], an in-place priority queue was introduced that supports *insert* in $O(1)$ worst-case time and *extract-min* in $O(\lg n)$ worst-case time involving at most $\lg n + O(1)$ element comparisons, n being the current size of the data structure. These upper bounds surpass the lower bounds known for a binary heap [20]. The designed priority queue is similar to a binary heap with two significant exceptions:

– To bypass the lower bound for *extract-min*, at the bottom levels a stronger invariant is enforced: For any node, the element at its left child should never be larger than the element at its right child.
– To bypass the lower bound for *insert*, $O(\lg^2 n)$ nodes are allowed to violate the binary-heap ordering in relation to their parents.

It is necessary to execute several background processes incrementally in order to achieve the optimal worst-case bounds on the number of element comparisons.

References

1. Bruun, A., Edelkamp, S., Katajainen, J., Rasmussen, J.: Policy-based benchmarking of weak heaps and their relatives. In: Festa, P. (ed.) SEA 2012. LNCS, vol. 6049, pp. 459–435. Springer, Heidelberg (2010)
2. Cantone, D., Cinotti, G.: QuickHeapsort, an efficient mix of classical sorting algorithms. Theoret. Comput. Sci. 285(1), 25–42 (2002)
3. Chen, J., Edelkamp, S., Elmasry, A., Katajainen, J.: In-place heap construction with optimized comparisons, moves, and cache misses. In: Rovan, B., Sassone, V., Widmayer, P. (eds.) MFCS 2012. LNCS, vol. 7464, pp. 259–270. Springer, Heidelberg (2012)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, 3th edn. (2009)
5. Diekert, V., Weiß, A.: QuickHeapsort: Modifications and improved analysis. In: Bulatov, A.A., Shur, A.M. (eds.) CSR 2013. LNCS, vol. 7913, pp. 24–35. Springer, Heidelberg (2013)
6. Driscoll, J.R., Gabow, H.N., Shrairman, R., Tarjan, R.E.: Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. Commun. ACM 31(11), 1343–1354 (1988)
7. Dutton, R.D.: Weak-heap sort. BIT 33(3), 372–381 (1993)

8. Edelkamp, S., Elmasry, A., Katajainen, J.: Two constant-factor-optimal realizations of adaptive heapsort. In: Iliopoulos, C.S., Smyth, W.F. (eds.) IWOCA 2011. LNCS, vol. 7056, pp. 195–208. Springer, Heidelberg (2011)
9. Edelkamp, S., Elmasry, A., Katajainen, J.: The weak-heap family of priority queues in theory and praxis. In: Mestre, J. (ed.) CATS 2012. Conferences in Research and Practice in Information Technology, vol. 128, pp. 103–112. Australian Computer Society, Inc., Adelaide (2012)
10. Edelkamp, S., Elmasry, A., Katajainen, J.: The weak-heap data structure: Variants and applications. J. Discrete Algorithms 16, 187–205 (2012)
11. Edelkamp, S., Elmasry, A., Katajainen, J.: A catalogue of algorithms for building weak heaps. In: Arumugam, S., Smyth, W.F. (eds.) IWOCA 2012. LNCS, vol. 7643, pp. 249–262. Springer, Heidelberg (2012)
12. Edelkamp, S., Elmasry, A., Katajainen, J.: Weak heaps engineered. J. Discrete Algorithms (to appear)
13. Edelkamp, S., Elmasry, A., Katajainen, J.: Optimal in-place heaps (submitted)
14. Edelkamp, S., Stiegeler, P.: Implementing Heapsort with $n \log n - 0.9n$ and Quicksort with $n \log n + 0.2n$ comparisons. ACM J. Exp. Algorithmics 7, Article 5 (2002)
15. Edelkamp, S., Wegener, I.: On the performance of Weak-Heapsort. In: Reichel, H., Tison, S. (eds.) STACS 2000. LNCS, vol. 1770, pp. 254–266. Springer, Heidelberg (2000)
16. Edelkamp, S., Weiß, A.: QuickXsort: Efficient sorting with $n \log n - 1.399n + o(n)$ comparisons on average. E-print arXiv:1307.3033, arXiv.org, Ithaca (2013)
17. Elmasry, A., Fredman, M.L.: Adaptive sorting: An information theoretic perspective. Acta Inform. 45(1), 33–42 (2008)
18. Elmasry, A., Hammad, A.: Inversion-sensitive sorting algorithms in practice. ACM J. Exp. Algorithmics 13, Article 1.11 (2009)
19. Elmasry, A., Jensen, C., Katajainen, J.: Multipartite priority queues. ACM Trans. Algorithms 5(1), Article 14 (2008)
20. Gonnet, G.H., Munro, J.I.: Heaps on heaps. SIAM J. Comput. 15(4), 964–971 (1986)
21. Katajainen, J.: The ultimate heapsort. In: Lin, X. (ed.) CATS 2012. Australian Computer Science Communications, vol. 20, pp. 87–96. Springer-Verlag Singapore Pte. Ltd., Singapore (1998)
22. Katajainen, J., Vitale, F.: Navigation piles with applications to sorting, priority queues, and priority deques. Nordic J. Comput. 10(3), 238–262 (2003)
23. Knuth, D.E.: Sorting and Searching, The Art of Computer Programming, vol. 3. Addison Wesley Longman, Reading, 2nd edn. (1998)
24. Levcopoulos, C., Petersson, O.: Adaptive heapsort. J. Algorithms 14(3), 395–413 (1993)
25. McDiarmid, C.J.H., Reed, B.A.: Building heaps fast. J. Algorithms 10(3), 352–365 (1989)
26. Musser, D.R.: Introspective sorting and selection algorithms. Software Pract. Exper. 27(8), 983–993 (1997)
27. Vuillemin, J.: A data structure for manipulating priority queues. Commun. ACM 21(4), 309–315 (1978)
28. Wegener, I.: Bottom-up-Heapsort, a new variant of Heapsort beating, on an average, Quicksort (if n is not very small). Theoret. Comput. Sci. 118(1), 81–98 (1993)
29. Williams, J.W.J.: Algorithm 232: Heapsort. Commun. ACM 7(6), 347–348 (1964)