Weak heaps engineered[☆]

Stefan Edelkamp[a], Amr Elmasry[b], Jyrki Katajainen[c,*]

[a] *Faculty 3—Mathematics and Computer Science, University of Bremen*
PO Box 330 440, 28334 Bremen, Germany
[b] *Department of Computer Engineering and Systems, Alexandria University*
Alexandria 21544, Egypt
[c] *Department of Computer Science, University of Copenhagen*
Universitetsparken 5, 2100 Copenhagen East, Denmark

Abstract

A weak heap is a priority queue that supports the operations *construct*, *minimum*, *insert*, and *extract-min*. To store n elements, it uses an array of n elements and an array of n bits. In this paper we study different possibilities for optimizing *construct* and *insert* such that *minimum* and *extract-min* are not made slower. We provide a catalogue of algorithms that optimize the standard algorithms in various ways. As the optimization criteria, we consider the worst-case running time, the number of instructions, branch mispredictions, cache misses, element comparisons, and element moves. Our contributions are summarized as follows:

1. The standard algorithm for *construct* runs in $O(n)$ worst-case time and performs $n-1$ element comparisons. Our improved algorithms reduce the number of instructions, the number of branch mispredictions, the number of element moves, and the number of cache misses.

2(a) Even though the worst-case running time of the standard *insert* algorithm is logarithmic, we show that—in contrast to binary heaps—n repeated *insert* operations require at most $3.5n + O(\lg^2 n)$ element comparisons.

2(b) We improve a recent result of ours, in which we achieve $O(1)$ amortized time per insertion, to guarantee $O(1)$ worst-case time per insertion. After the deamortization, *minimum* still takes $O(1)$ worst-case time and involves no element comparisons, and *extract-min* takes $O(\lg n)$ worst-case time and involves at most $\lg n + O(1)$ element comparisons. This constant-factor optimality concerning the number of element comparisons has previously been achieved only by pointer-based multipartite priority queues.

We have implemented most of the proposed algorithms and tested their practical behaviour. Interestingly, for integer data, reducing the number of branch mispredictions turned out to be an effective optimization in our experiments.

[☆] A preliminary version of this paper was presented at the 23rd International Workshop on Combinatorial Algorithms held in Tamil Nadu, India, in July 2012.

[*] Corresponding author

Keywords: data structures, priority queues, weak heaps, constant factors

1. Introduction

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* [5]. In applications for which this set of operations is sufficient, the basic priority queue that the users would select is a binary heap [15] or a weak heap [6]. Both of these data structures are known to perform well, and the difference in performance is quite marginal in typical cases. Most library implementations are based on binary heaps. 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. In Table 1 (top section) we summarize the performance of these two data structures.

A *weak heap* (see Figure 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 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 enforces no relation between an element in a node and those stored in its left subtree.

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 related research articles (see, for example, [4]) it has been reported that a compact representation of a bit array, where $\lceil n/w \rceil$ words are used for representing n bits in a computer with word size w, can be slower than a less compact representation where one byte is reserved for each bit. We could verify this to be the case in our computing environment. Therefore, for most of our implementations reverse bits are maintained in a byte array.

In Section 2 of this paper we study the problem of constructing a weak heap for a collection of n elements given in an array. The standard algorithm for building a weak heap [6] is asymptotically optimal, involving small constant factors. Nevertheless, this algorithm can be improved in several ways. The reason why we consider these different optimizations is that it may be possible to apply the same type of optimizations for other fundamental algorithms. For

some applications the proposed optimizations may be significant, still we do not consider any concrete applications per se.

We provide algorithms that perform the following optimizations:

Instruction optimization: We utilize bit-manipulation operations on the word level to find the position of the least-significant 1-bit. Since the used bit-manipulation operations are native on modern computers, a loop can be replaced by a couple of instructions that are executed in constant time.

Branch optimization: We describe a simple twist that reduces the number of branch mispredictions of *construct* from $\Theta(n)$ to $O(1)$. On modern computers having long pipelines, branch mispredictions can be expensive.

Cache optimization: We improve the cache behaviour of an alternative weak-heap construction algorithm by implementing it in a depth-first manner. The resulting algorithm is cache oblivious. We give a recursive implementation and point out how to convert it to an iterative implementation.

Move optimization: We reduce the bound on the number of element moves from $3n$ to $2n$. Still, the number of element comparisons is unchanged.

In section 3 we study the problem of inserting elements into a weak heap. The standard insertion algorithm [6] runs in $O(\lg n)$ worst-case time involving at most $\lceil \lg n \rceil$ element comparisons. There exists a general data-structural transformation [1] that improves the running time to a constant; but this transformation is complicated (involving techniques like global rebuilding), increases the memory overhead due to the additional pointers introduced, and transfers the cost of *insert* to *extract-min* making the latter slower.

In [8] we showed that—for array-based weak heaps—the cost of *insert* can be improved to an amortized constant while preserving $O(\lg n)$ worst-case running time and at most $\lg n + O(1)$ element comparisons for *extract-min*. The idea is to use a buffer that supports constant-time insertion. A new element is inserted into the buffer as long as the buffer size is below a threshold. Once the buffer is full, all the elements of the buffer are moved to the weak heap.

We complement these results in two ways:

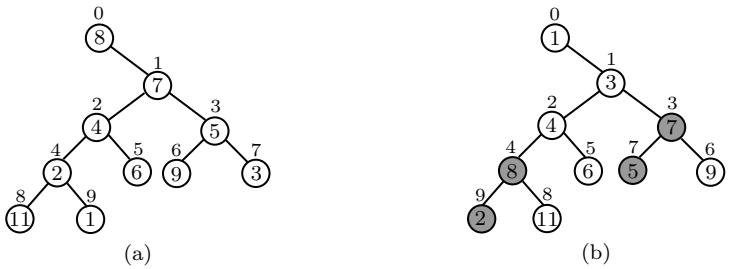

Figure 1: (a) An input of 10 integers and (b) a weak heap constructed by the standard algorithm. The reverse bits are set to 1's for the grey nodes. The small numbers above the nodes are the actual array indices.

Table 1: The worst-case number of element comparisons performed by some elementary priority queues—the data structures in the top and bottom sections are array-based and those in the middle section are pointer-based; n denotes the number of elements stored in the data structure prior to the operation in question. All structures are modified to support *minimum* in $O(1)$ worst-case time with no element comparisons (by maintaining a pointer to the minimum and updating it whenever necessary).

Data structure	*construct*	*insert*	*extract-min*
binary heap [12, 15]	$2n$	$\lceil \lg n \rceil$	$2\lceil \lg n \rceil$
weak heap [6]	$n-1$	$\lceil \lg n \rceil$	$\lceil \lg n \rceil$
binomial queue [11, 13]	$n-1$	2	$2\lg n + O(1)$
run-relaxed weak heap [8]	$n-1$	3	$2\lg n + O(1)$
multipartite priority queue [11]	$O(n)$	$O(1)$	$\lg n + O(1)$
one-buffer weak heap [8]	$n-1$	$O(\lg n)$	$\lceil \lg n \rceil + 1$
two-buffer weak heap [this paper]	$n-1$	$O(1)$	$\lg n + O(1)$

Repeated insertions: We show that starting from an empty weak heap and inserting n elements repeatedly into it one by one, the overall construction involves at most $3.5n + O(\lg^2 n)$ element comparisons.

Worst-case constant-time insertion: We improve the one-buffer amortized solution such that *insert* takes $O(1)$ time in the worst case. The modified structure preserves the cost of *minimum* and that of *extract-min*. The key idea is to use two buffers instead of one and to merge the buffers into the weak heap incrementally.

Previously, the constant-factor optimality with respect to the number of element comparisons ($O(1)$-time *insert* and *minimum*, and $O(\lg n)$-time *extract-min* involving at most $\lg n + O(1)$ element comparisons) has only been achieved with a more elaborate pointer-based structure called a multipartite priority queue [11].

In Table 1 (middle section) we summarize the performance of pointer-based priority queues: binomial queues [11, 13], run-relaxed weak heaps [8], and multipartite priority queues [11]. In Table 1 (bottom section) we highlight the difference between one-buffer weak heaps and two-buffer weak heaps. It should be pointed out that the buffers are maintained at the end of the element array.

When describing our algorithms we have taken an approach where we keep the description, analysis, implementation, and experimentation related to one algorithm together, instead of having a separate section on experiments and their outcomes. We describe the algorithms in the form of pseudo-code that is detailed enough for explanatory purposes. The actual programs used in the experiments are provided in an electronic appendix [7].

Since the effect of some of our optimizations is sensitive to the environment where the programs are run, we repeated the experiments on several computers. In this paper we present the results for two computers:

32-bit architecture: Intel® Celeron™ Pentium 4 CPU @ 2.66 GHz—the size of the L2 cache is 256 MB, this cache is 4-way associative, the length of the cache lines is 64 B, and the size of the main memory is 1 GB. Input elements are 4-byte integers.

64-bit architecture: Intel® Core™ i5-2520M CPU @ 2.50 GHz × 4—the size of the L3 cache is about 3 MB, this cache is 12-way associative, the length of the cache lines is 64 B, and the size of the main memory is 3.8 GB. Input elements are 8-byte integers.

In both computers the underlying operating system was Ubuntu 12.04 (Linux kernel 3.2.0-(23/41)-generic), the compiler used was `g++` (`gcc` version 4.7.3), and the compiler options used were `-O3`, `-Wall`, `-std=c++11`, `-msse4.2`, `-mabm`, and `--param case-values-threshold=50`. All execution times were measured using the function `gettimeofday` in `sys/time.h`. Other measurements were done using the profiling tools available in `valgrind` (version 3.7.0). For a problem of size n, each experiment was repeated $2^{26}/n$ times and the mean was reported.

Even though we only consider array-based weak heaps, it deserves to be mentioned that pointer-based weak heaps can be used to implement addressable priority queues, which in addition support *delete* and *decrease* operations requiring a direct access to the elements stored in the data structure; see [3, 8].

2. Construction

2.1. The standard construction procedure for weak heaps

The *distinguished ancestor* of a_j, $j \neq 0$, is the parent of a_j if a_j is a right child, and the distinguished ancestor of the parent of a_j if a_j is a left child. We assume that, for a given entry, the procedure *distinguished-ancestor* returns the index of its distinguished ancestor. Due to the weak-heap ordering, no element is smaller than the element at its distinguished ancestor.

The *join* operation combines two weak heaps into one, conditioned on the following settings. Let a_i and a_j be two nodes in a weak heap such that the element at a_i is smaller than or equal to every element in the left subtree of a_j. Conceptually, a_j and its right subtree form a weak heap, while a_i and the left subtree of a_j form another weak heap. (Note that a_i cannot be a descendant of a_j.) If the element at a_j is smaller than that at a_i, the two elements are swapped and r_j is flipped. As a result, the element at a_j will be smaller than or equal to every element in its right subtree, and the element at a_i will be smaller than or equal to every element in the subtree rooted at a_j. Thus, *join* requires constant time and involves one element comparison and possibly an element swap.

In the standard construction of a weak heap (see Figure 2) the nodes are visited one by one in reverse order (from the end to the beginning of the array), and the two weak heaps rooted at a visited node and its distinguished ancestor are joined. It has been shown, for example in [10], that the amortized cost to get from a node to its distinguished ancestor is $O(1)$. Hence, the overall construction requires $O(n)$ time in the worst case. Moreover, $n - 1$ element comparisons and at most $n - 1$ element swaps are performed.

```
class weak-heap

a: element[]
r: bit[]
n: index

method distinguished-ancestor(j: index)
assert j ≠ 0
while (j bitand 1) = r_⌊j/2⌋
|  j ← ⌊j/2⌋
return ⌊j/2⌋

method join(i: index, j: index)
if a_j < a_i
|  swap(a_i, a_j)
|  r_j ← 1 − r_j

method construct()
for i ∈ {0, 1, ..., n − 1}
|  r_i ← 0
for j = n − 1 downto 1
|  i ← distinguished-ancestor(j)
|  join(i, j)
```

Figure 2: Standard construction of a weak heap. Here, as in other figures, we assume that there are n elements in the array a.

2.2. Instruction optimization: Accessing ancestors faster

The number of instructions executed by the standard construction algorithm can be reduced by observing that the reverse bits are initialized to 0 and set bottom-up while scanning the nodes. Therefore, in the *join* operation the assignment $r_j \leftarrow 1 - r_j$ can be replaced with $r_j \leftarrow 1$, and in the *distinguished-ancestor* operation the output can be computed from the array index by considering the position of the least-significant 1-bit in j. Edelkamp and Stiegeler [9] implemented this optimization by replacing the test $(j \ \textbf{\textit{bitand}}\ 1) = r_{\lfloor j/2 \rfloor}$ with $(j \ \textbf{\textit{bitand}}\ 1) = 0$. Our observation is that the loop is unnecessary on computers where the position of the least-significant 1-bit can be located using the native

```
method distinguished-ancestor(j: index);
assert j ≠ 0;
z ← trailing-zero-count(j);
return j >> (z + 1)
```

Figure 3: A faster way of finding the distinguished ancestor of a node.

primitive operation that counts the number of trailing 0-bits in a word (see Figure 3). Assuming the availability of the needed hardware support, the loop removal makes the analysis of the algorithm straightforward: The distinguished ancestor can be accessed in $O(1)$ worst-case time, and for each node (except the root) one element comparison and at most one element swap are performed.

To test the effectiveness of this idea in practice, we programmed the standard algorithm and this instruction-optimized refinement. When implementing the function *trailing-zero-count*, we tried the loop variant as well as variants that applied the built-in functions *builtin-ctz* and *builtin-popcount* provided by our compiler (`g++`). On our 64-bit processor, *builtin-ctz* was translated to the native `bsfq` instruction (bit scan forward). As an alternative, we used *builtin-popcount*(**bitnot** (x **bitand** ($x - 1$))) to compute *trailing-zero-count*(x); its translation used the native `popcntq` instruction. As indicated in Table 2, the instruction optimization reduced the number of instructions executed. However, as the numbers in Table 3 indicate, this did not automatically mean that the programs became faster. In fact, the **while** version was efficient, even though it executed more instructions than the best instruction-optimized version.

Table 2: Standard vs. instruction-optimized constructions; number of instructions executed divided by n—the average was the same in all cases; the elements were given in random order.

32/64 bit n	**standard**	**while loop**	*builtin-ctz*	*builtin-popcount*
2^{10} 2^{15} 2^{20} 2^{25}	22.5	16.1	12.8	16.6[a]

[a] The built-in instruction was not supported by our 32-bit test computer.

*2.3. Branch optimization: No **if** statements*

Branch prediction is an important efficiency issue in pipelined processors, as upon a conditional branch being fetched the processor normally guesses the outcome of the condition and starts the execution of the instructions in one of the branches speculatively. If the prediction was wrong, the pipeline must be flushed, a new set of instructions must be fetched in, and the work done with the wrong branch of the code is simply wasted. To run programs efficiently in this kind of environment one may want to avoid conditional branches if possible.

The standard construction algorithm has only few conditional branches. Instruction optimization already removed the loop used to access the distinguished ancestor. In accordance, the main body of the algorithm has two unnested loops that both end with a conditional branch; but only when stepping out of a loop a misprediction incurs. Hence, the main issue to guarantee $O(1)$ branch mispredictions is to remove the **if** statement in the *join* operation. To do that, we remove the conditional branch by using arithmetic operations (see Figure 4).

As shown in Table 4, combining this optimization with that described in the previous section, in our test environment, the running times improved—at

Table 3: Standard vs. instruction-optimized constructions; execution time in nanoseconds divided by n; the elements were given in random order.

32 bit n	**standard**	**while loop**	*builtin-ctz*	*builtin-popcount*
2^{10}	26.5	24.7	21.3	–
2^{15}	23.4	21.8	18.5	–
2^{20}	23.4	21.9	18.6	–
2^{25}	24.1	22.1	18.6	–

64 bit n	**standard**	**while loop**	*builtin-ctz*	*builtin-popcount*
2^{10}	9.0	8.1	7.9	8.5
2^{15}	8.2	7.1	7.1	7.8
2^{20}	8.3	7.3	7.2	7.8
2^{25}	8.3	7.5	7.3	8.2

method *join*(i: index, j: index)
$smaller \leftarrow (a_j < a_i)$
$\Delta \leftarrow smaller * (j - i)$
$k \leftarrow i + \Delta$
$\ell \leftarrow j - \Delta$
$t \leftarrow a_\ell$
$a_i \leftarrow a_k$
$a_j \leftarrow t$
$r_j \leftarrow smaller$

Figure 4: Joining two weak heaps without a conditional branch.

least for small problem instances. On our 64-bit computer, these are the best running times among those of our construction algorithms. The test results given in Table 5 confirm that for the branch-optimized version the number of branch mispredictions incurred is indeed negligible.

2.4. An alternative construction: Don't look upwards

Another way of building a weak heap is to avoid climbing to the distinguished ancestors altogether. We still build the heap bottom-up level by level, but we restore weak-heap ordering by considering each node and its right subtree. The idea comes from Floyd's algorithm for constructing binary heaps [12]. To mimic this algorithm we need a *sift-down* operation, which is explained next.

Assume that the elements at the right subtree of a_j obey the weak-heap ordering. The *sift-down* operation is used to establish weak-heap ordering between the element at a_j and those in the right subtree of a_j. Starting from the

Table 4: Instruction-optimized vs. branch-optimized constructions; execution time in nanoseconds divided by n; the elements were given in random order.

32 bit n	*builtin-ctz*	**branch optimized**
2^{10}	21.3	20.1
2^{15}	18.5	16.9
2^{20}	18.6	19.2
2^{25}	18.6	19.6

64 bit n	*builtin-ctz*	**branch optimized**
2^{10}	7.9	5.3
2^{15}	7.1	4.7
2^{20}	7.2	5.1
2^{25}	7.3	5.4

Table 5: Standard vs. instruction-optimized vs. branch-optimized constructions; number of mispredicted branches divided by n; the elements were given in random order.

32/64 bit n	**standard**	*builtin-ctz*	**branch optimized**
2^{10} 2^{15} 2^{20} 2^{25}	1.04	0.50	0.00

right child of a_j, the last node on the left spine of the right subtree of a_j is identified; this is done by visiting left children until reaching a node that has no left child. The path from this node to the right child of a_j is traversed upwards, and *join* operations are repeatedly performed between a_j and the nodes along this path. The correctness of the *sift-down* operation follows from the fact that, after each *join*, the element at location j is less than or equal to every element in the left subtree of the node considered in the next *join*.

Table 6: Standard vs. alternative constructions; execution time in nanoseconds divided by n; the elements were given in random order.

32 bit n	**standard**	**alternative**
2^{10}	26.5	23.2
2^{15}	23.4	22.9
2^{20}	23.4	45.5
2^{25}	24.1	45.5

64 bit n	**standard**	**alternative**
2^{10}	9.0	8.5
2^{15}	8.2	8.7
2^{20}	8.3	11.6
2^{25}	8.3	12.6

With *sift-down* in hand, a weak heap can be constructed by calling the procedure on every node starting from the lower levels upwards (see Figure 5). As indicated in Table 6, the running times achieved for this method are not satisfactory—at least for large problem instances. We relate the slowdown for large values of n to cache effects (see Table 8).

2.5. Cache optimization: Depth-first construction

To avoid the bad cache behaviour, the nodes of the heap should be visited in depth-first order rather than in breadth-first order. This idea—applied to binary heaps in [2]—improves the locality of accesses as the traversal stays longer at nearby memory locations. The number of element comparisons obviously remains unchanged. We would like to point out that this method is a cache-optimized version of the method described in the previous section. A recursive procedure is given in Figure 6. Following the guidelines given in [2], it is not difficult to avoid recursion when implementing the procedure.

As the running times in Table 7 show, the cache-optimized version was slower than the standard version. On the other hand, the cache-miss rates given in

```
method sift-down(j: index, n: index)
k ← 2j + 1 − r_j
if k ≥ n
|  return;
while 2k + r_k < n
|  k ← 2k + r_k
while k ≠ j
|  join(j, k)
|  k ← ⌊k/2⌋

method construct()
for i ∈ {0, 1, ..., n − 1}
|  r_i ← 0
for j = ⌊n/2⌋ − 1 to 0 step −1
|  sift-down(j, n)
```

Figure 5: An alternative construction of a weak heap.

```
method depth-first-construct(i: index, j: index)
if j ≥ n
|  return
r_j ← 0
depth-first-construct(j, 2j + 1)
depth-first-construct(i, 2j)
join(i, j)

method construct()
if n = 0
|  return
r_0 ← 0
depth-first-construct(0, 1)
```

Figure 6: Recursive depth-first weak-heap construction.

Table 7: Standard vs. cache-optimized constructions; execution time in nanoseconds divided by n; the elements were given in random order.

32 bit n	**standard** byte array	**standard** bit array	**cache optimized**
2^{10}	26.5	36.4	30.7
2^{15}	23.4	32.2	27.8
2^{20}	23.4	32.3	27.7
2^{25}	24.1	32.9	27.8

64 bit n	**standard** byte array	**standard** bit array	**cache optimized**
2^{10}	9.0	14.7	11.5
2^{15}	8.2	13.5	10.2
2^{20}	8.3	13.7	10.4
2^{25}	8.3	13.3	10.2

Table 8: The number of cache misses incurred by different algorithms as a factor of n/B, where B is the block size in words; the elements were given in random order.

32 bit n	**standard** byte array	**standard** bit array	*builtin-ctz*	**alternative**	**cache optimized**
2^{10}	1.25	1.03	1.25	1.25	1.25
2^{15}	1.25	1.03	1.25	1.25	1.25
2^{20}	2.67	2.03	2.43	7.57	1.26
2^{25}	2.75	2.09	2.50	7.62	1.26

64 bit n	**standard** byte array	**standard** bit array	*builtin-ctz*	**alternative**	**cache optimized**
2^{10}	1.13	1.02	1.13	1.13	1.13
2^{15}	1.13	1.02	1.13	1.13	1.13
2^{20}	1.94	1.66	1.85	5.09	1.13
2^{25}	2.36	2.03	2.24	5.55	1.12

Table 8 indicate that the depth-first construction has a better cache behaviour.

2.6. Move optimization: Trading swaps for delayed moves

Now we implement the aforementioned depth-first construction in a different way (see Figure 7). The idea is to walk down the left spine of the child of the root and call the procedure recursively at every node we visit. After coming back from the recursive calls, the *sift-down* operation is applied to restore weak-heap

```
method depth-first-construct(i: index)
r_i ← 0
j ← 2i + 1
while j < ⌊n/2⌋
|   depth-first-construct(j)
|   j ← 2j
sift-down(i, n)

method construct()
if n = 0
|   return
depth-first-construct(0)
```

Figure 7: Another implementation for depth-first weak-heap construction.

ordering at the root. In the worst case the number of element moves performed by this algorithm is the same as that performed by the standard algorithm. For both algorithms, $n-1$ swaps may be performed. As each swap involves three element moves (assignments), the number of element moves is bounded by $3n$.

To reduce the bound on the number of element moves to $2n$, we postpone the swaps done during the *sift-down* operation. (A similar approach was used by Wegener [14] when implementing *sift-down* in a bottom-up manner for binary heaps.) The idea is to use a bit vector (of at most $\lg n$ bits) that indicates the winners of the comparisons along the left spine; a 1-bit indicates that the corresponding element on the spine was the smaller of the two elements involved in this comparison. Of course, we still compare the next element up the spine with the current winner of the executed comparisons. After the results of the comparisons are set in the bit vector, the actual element movements are performed. More precisely, the elements corresponding to 1-bits are rotated; this accounts for at most $\mu+2$ element moves if all the $\mu+1$ elements on the left spine plus the root are rotated (assuming that the number of nodes on the left spine of the child of the root is μ). To calculate the number of element moves involved, we note that the sum of the lengths of all the left spines is $n-1$. In addition, we note that the *sift-down* operation will be executed for at most $n/2$ nodes; these are the non-leaves. Summing the number of element moves for all the—at most $n/2$—*sift-down* operations, we get the bound $2n$. Tables 9 and 10 show the effect of move optimization in practice.

3. Insertion

3.1. Repeated insertions: Linear number of element comparisons

To insert an element e, we first add e to the next available array entry. To reestablish the weak-heap ordering, we use the *sift-up* operation. As long as e is smaller than the element at its distinguished ancestor, we swap the two using

Table 9: Standard vs. move-optimized constructions; execution time in nanoseconds divided by n; the elements were given in random order/decreasing order.

32 bit	**standard**		**move optimized**	
n	**random**	**decreasing**	**random**	**decreasing**
2^{10}	26.5	16.2	43.9	37.2
2^{15}	23.4	13.5	41.1	34.6
2^{20}	23.4	15.8	41.1	34.7
2^{25}	24.1	16.2	41.2	34.3

64 bit	**standard**		**move optimized**	
n	**random**	**decreasing**	**random**	**decreasing**
2^{10}	9.0	4.8	15.2	10.8
2^{15}	8.2	4.1	14.4	9.8
2^{20}	8.3	4.7	14.4	9.7
2^{25}	8.3	5.4	14.6	9.9

Table 10: Standard vs. move-optimized constructions; number of moves divided by n; the elements were given in random order/decreasing order.

32/64 bit	**standard**		**move optimized**	
n	**random**	**decreasing**	**random**	**decreasing**
2^{10}	1.49	2.99	1.16	1.99
2^{15}	1.49	2.99	1.16	1.99
2^{20}	1.49	3.00	1.16	2.00
2^{25}	1.50	3.00	1.16	2.00

the *join* operation and repeat for the new location of e. Thus, *sift-up* at location j requires $O(\lg j)$ time and involves at most $\lceil \lg(1+j) \rceil$ element comparisons.

When constructing a weak heap using repeated insertions (see Figure 8) we observed that, while the execution time increased with n, the number of element comparisons stayed constant per element even when n increased (see Tables 11 and 12). As the worst-case input for the experiments, we used the special sequence $\langle 0, n-1, n-2, \ldots, 1 \rangle$ as advised in [10]. Next we prove that the number of element comparisons performed is indeed linear in the worst case.

Theorem 1. *The total number of element comparisons performed while constructing a weak heap using n in-a-row insertions is at most $3.5n + O(\lg^2 n)$.*

Proof. We distinguish between two types of element comparisons done by the *sift-up* operations. An element comparison that involves the root or triggers the **break** statement to get out of the **while** loop is called a *terminal element comparison*. There is exactly one such comparison per insertion, except for the

```
class weak-heap

a: element[]
r: bit[]
n: index
n′: index

method sift-up(j: index)
while j ≠ 0
|  i ← distinguished-ancestor(j)
|  before ← r_j
|  join(i, j)
|  if before = r_j
|  |  break
|  j ← i

method insert(x: element)
assert a_0..a_{n′−1} and r_0..r_{n′−1} form a weak heap
a_{n′} ← x
r_{n′} ← 0
if (n′ bitand 1) = 1     (*)
|  r_{⌊n′/2⌋} ← 0
sift-up(n′)
n′ ← n′ + 1

method construct()
for i = 1 to n − 1
|  insert(a_i)
```

Figure 8: Constructing a weak heap by repeated insertions.

first insertion, resulting in $n-1$ terminal element comparisons for the whole construction. All other element comparisons are *non-terminal*.

Next we calculate an upper bound on the number of non-terminal element comparisons performed. Fix a node x whose height is h, $h \in \{1, 2, \ldots, \lceil \lg n \rceil + 1\}$. Consider all the non-terminal element comparisons performed between the element at x and its distinguished ancestor throughout the process. For such a comparison to take place, an element should have been inserted in the right subtree of x; this would include the insertion of x itself. Consider the first element e that is inserted in the right subtree of x at distance d from x and results in a non-terminal element comparison between the element at x and its distinguished ancestor. That is, d can take the values 0, 1, ..., $h-1$.

For $d = 0$, the element at x and its distinguished ancestor are always compared unless x is the root. For $d = 1$, we have to consider the **if** statement marked with a star (*) in Figure 8. When the inserted node is the only child of

Table 11: Standard vs. repeated-insertion constructions; execution time in nanoseconds divided by n; the elements were given in random order/special sequence.

32 bit n	**standard**		**repeated insertions**	
	random	**special**	**random**	**special**
2^{10}	26.5	15.9	46.8	60.6
2^{15}	23.4	13.2	47.8	64.8
2^{20}	23.4	18.8	48.3	76.0
2^{25}	24.1	16.2	48.5	91.2

64 bit n	**standard**		**repeated insertions**	
	random	**special**	**random**	**special**
2^{10}	9.0	4.8	15.9	13.3
2^{15}	8.2	4.0	16.6	18.2
2^{20}	8.3	4.7	16.9	20.8
2^{25}	8.3	4.9	16.7	25.3

Table 12: Standard vs. repeated-insertion constructions; number of element comparisons divided by n; the elements were given in random order/special sequence.

32/64 bit n	**standard**		**repeated insertions**	
	random	**special**	**random**	**special**
2^{10}	0.99	0.99	1.76	3.38
2^{15}	0.99	0.99	1.77	3.49
2^{20}	0.99	0.99	1.77	3.49
2^{25}	1.00	1.00	1.77	3.49

x, it is made a left child by updating the reverse bit at x. So, this first insertion at distance one will never trigger a non-terminal element comparison; only the second insertion does that. Consider now the case where $d \geq 2$. Because of the non-terminal comparison between the element at x and its distinguished ancestor, the reverse bit at x is flipped and the right subtree of x becomes its left subtree. All the upcoming insertions that will land in this subtree at distance d from x will not involve x as a distinguished ancestor. It follows that, for this given subtree, the element at x will not be compared with that at its distinguished ancestor until an element is inserted below x at distance $d + 1$ from x. In conclusion, the node x can be charged with at most two element comparison for each level at distance $d \geq 2$ in the subtree of x, and at most one element comparison per level for $d = 0$ and $d = 1$. Summing the number of non-terminating element comparisons done for all values of d, we get that the element at x is compared against the element at its distinguished ancestor at most twice its height minus two when $h \geq 2$, and at most once when $h = 1$.

In a weak heap of size n, there are at most $\lceil n/2^h \rceil$ nodes of height h, $h \in \{1, 2, \ldots, \lceil \lg n \rceil + 1\}$. On the basis of the discussion in the preceding paragraph it follows that the number of non-terminal element comparisons is bounded by $\lceil n/2 \rceil + \sum_{h=2}^{\lceil \lg n \rceil + 1} (2h-2) \cdot \lceil n/2^h \rceil < 2.5n + O(\lg^2 n)$. Including the $n-1$ terminal element comparisons, we conclude that the total number of element comparisons performed by all the n insertions is at most $3.5n + O(\lg^2 n)$. □

Observe that the number of element comparisons and that of element moves go hand in hand, so the number of element moves performed is also linear. That is, the $O(n \lg n)$ running time is consumed in distinguished-ancestor calculations.

3.2. Amortized constant-time insertion: The power of a buffer

In an earlier paper [8] we described how to modify *insert* such that, for any intermixed sequence of operations, the amortized cost of *insert* is a constant and the worst-case cost of *extract-min* is $O(\lg n)$. Since the *insert* algorithm to be presented in the next section is based on this earlier solution and to keep the paper self-contained, we briefly recall this amortized solution here.

The key idea is to split the element array into two parts: the first subarray contains a weak heap and the second subarray a *buffer*. For both parts the minimum is at the front of the subarray. We keep track of which of the two contains the overall minimum by maintaining, what we call, a *minimum indicator*. As before, we let n denote the current number of elements in the element array. In addition, we keep a variable n' specifying the starting position of the buffer. We maintain an invariant that the buffer is never larger than $\lfloor \lg (1 + n') \rfloor$. A new element is inserted into the buffer and, if this causes the buffer to exceed its capacity, we perform a *bulk insertion* by embedding all the buffer elements into the weak heap. If no bulk insertion is required and the new element is smaller than the minimum of the buffer, these two elements are swapped and the minimum indicator is updated if necessary. The implementation details of this insertion algorithm are described in a pseudo-code form in Figure 9.

In a bulk insertion, weak-heap ordering is reestablished in two phases. In the first phase we process the nodes level by level as in the alternative weak-heap construction. We start with the parents of the buffer elements and perform a *sift-down* operation for each of them. This restores the weak-heap ordering between the element at the parent and those in its right subtree. We then consider the parents of the parents on the next upper level, restoring weak-heap ordering up to this level. We repeat this until the number of nodes that we need to deal with at a level is (at most) two. In the second phase we reestablish the weak-heap ordering on the two paths from the remaining two nodes upwards. To do that, we identify the distinguished ancestors of the two nodes and then perform a *sift-down* followed by a *sift-up* starting at each of these two distinguished ancestors. Note that, if the buffer encompasses more than half the elements, the second phase is trivial and the procedure falls back to the alternative weak-heap construction.

The correctness of the bulk insertion can be demonstrated as follows. In the first phase of the procedure, after considering the ℓth level the value at each

```
class one-buffer-weak-heap

a: element[]
r: bit[]
n: index
n′: index
overall-minimum: index

method bulk-insert(n′: index, n: index)
assert a_0..a_{n′−1} and r_0..r_{n′−1} form a weak heap; a_{n′}..a_{n−1} is a buffer
right ← n − 1
left ← max{n′, ⌊right/2⌋ + 1}
while right > left + 1
    left ← ⌊left/2⌋
    right ← ⌊right/2⌋
    for j ∈ {right, right − 1, ..., left}
        sift-down(j, n)
j ← 0
if right ≠ 0
    j ← distinguished-ancestor(right)
    sift-down(j, n)
if left ≠ 0
    i ← distinguished-ancestor(left)
    sift-down(i, n)
    sift-up(i)
sift-up(j)

method insert(x: element)
assert a_0..a_{n′−1} and r_0..r_{n′−1} form a weak heap; a_{n′}..a_{n−1} is a buffer
a_n ← x
r_n ← 0
if n − n′ ≥ ⌊lg(1 + n′)⌋
    bulk-insert(n′, n + 1)
    n′ ← n + 1
    overall-minimum ← 0
else if a_n < a_{n′}
    swap(a_{n′}, a_n)
    if a_{n′} < a_0
        overall-minimum ← n′
    else
        overall-minimum ← 0
n ← n + 1
```

Figure 9: Amortized constant-time insertion.

node up to level ℓ is less than or equal to the value at every node in its right subtree. Once we reach a level where there are only two affected nodes, the distinguished ancestor of each of the two nodes is located. In the second phase of the procedure, the *sift-down* operation, when called for each of these two distinguished ancestors, ensures the weak-heap ordering up to both nodes. We then call the *sift-up* operation for each of the two nodes. (Note that one of these two distinguished ancestors could be the ancestor of the other, and hence the order of handling the two nodes is important.) Consider a $join(i, j)$ operation that is performed by the *sift-up* operation. At this point, the value of a_i is less than or equal to every element in the left subtree of a_j; this ensures the validity of the precondition for the *join* operations. After the *sift-up* operations, the weak-heap ordering must have been restored everywhere.

The intuition behind the amortized constant-time insertion is that the number of nodes that need to be considered almost halves from a level to the next higher level. In the meantime, the amount of work needed for a *sift-down* operation only increases linearly with the level number. The total work done when there are only two nodes to be considered is obviously logarithmic (two *distinguished-ancestor*, two *sift-down*, and two *sift-up* operations). Because the number of elements bulk-inserted is logarithmic, this last cost is a constant per *insert*. A more detailed analysis [8] shows that for n repeated insertions the number of element comparisons performed is at most $6n + o(n)$; $4n + o(n)$ are due to the bulk insertions and $2n$ due to insertions not requiring a bulk insertion.

Since the minimum indicator gives the location of the overall minimum, *minimum* involves no element comparisons and runs in $O(1)$ worst-case time.

In *extract-min* there are two cases to consider. If the overall minimum is in the buffer, the new minimum of the buffer is found by scanning the buffer and the found minimum is moved to the front of the buffer. Also, the minimum indicator is updated if necessary. To avoid holes in the buffer, the last element of the buffer is moved into the old place of the minimum if necessary. On the other hand, if the overall minimum is in the weak heap, a replacement element for the deleted element is moved from the end of the buffer and a *sift-down* operation for the replacement element at the root is applied. If the buffer is empty, the last element of the weak heap is used as a replacement element. Deleting the minimum of the buffer requires $O(\lg n)$ time and involves at most $\lceil \lg n \rceil$ element comparisons. This matches the bounds for deleting the minimum of the weak heap. Adding the extra comparison needed to update the minimum indicator, the number of element comparisons performed in the worst case is $\lceil \lg n \rceil + 1$.

Naturally, the new procedure could be used for constructing a weak heap by repeated insertions. Even though it is not recommended to do this in practice, we used it as a benchmark for measuring the efficiency of the *insert* procedure. In Table 13 we report the results of our experiments for the metrics we have considered earlier. The reader can compare these numbers to those obtained by the other weak-heap construction methods (see Tables 2–12). For example, the number of instructions executed is a factor of 10 higher than that for the instruction-optimized construction method. With some tuning we could improve the performance a little bit, e.g. by using our branch-optimization techniques

Table 13: Performance of the weak-heap construction based on bulk insertions; the elements were given in random order. All measurement results were divided by n, the last column by n/B; time is in nanoseconds.

32 bit n	**time**	**instructions**	**comparisons**	**moves**	**mispredictions**	**misses**
2^{10}	108.4	143	4.11	3.87	2.99	1.25
2^{15}	102.1	135	3.94	3.77	2.76	1.25
2^{20}	100.0	131	3.83	3.69	2.61	2.43
2^{25}	97.5	128	3.75	3.64	2.49	2.50

64 bit n	**time**	**instructions**	**comparisons**	**moves**	**mispredictions**	**misses**
2^{10}	32.1	135	4.11	3.87	2.95	1.13
2^{15}	29.9	129	3.94	3.77	2.74	1.13
2^{20}	28.7	125	3.83	3.69	2.58	1.85
2^{25}	27.6	123	3.75	3.64	2.46	2.24

the number of mispredictions can be reduced from $O(n)$ to $O(n/\lg n)$. However, these tunings did not have any significant effect on the running time.

3.3. Worst-case constant-time insertion: The power of two buffers

In this section we utilize the amortized solution to achieve the same bounds for all operations in the worst case. Our main goal is to show that it is possible to achieve constant-factor optimality with respect to the number of element comparisons performed. In the worst case, *minimum* and *insert* take $O(1)$ time, and *extract-min* takes $O(\lg n)$ time and involves at most $\lg n + O(1)$ element comparisons. As for multipartite priority queues [11], we can also support the operation *borrow*, which extracts an unspecified element from the data structure, in $O(1)$ worst-case time. The amount of space used in addition to the elements themselves is $n + O(w)$ bits, where w is the word size of the computer.

In principle, the idea behind this improvement is simple, but the technical details are a bit involved. We use two buffers—a *submersion buffer* and an *insertion buffer*—instead of one. That is, the element array is divided into three parts: a weak heap and the two buffers. We use n' and n'' to specify the indices of the starting points of the submersion buffer and the insertion buffer respectively, where $n' \leq n''$. As normal, we use n to denote the current number of elements stored in all three parts. We set the maximum size of the submersion buffer to $\lfloor \lg(1 + n') \rfloor$ and that of the insertion buffer to $\lfloor \lg(1 + n'') \rfloor$.

While new elements are inserted into the insertion buffer, the elements in the submersion buffer are at the same time incrementally embedded into the weak heap. An elegant solution for doing this would be to use a coroutine that executes a constant number of instructions per *insert*; after each such step it

pauses and in the next call it resumes its execution from the place where the computation was stopped in the previous call. One has to only make sure that each call of the coroutine executes enough instructions such that the submersion process is completed before the insertion buffer will overflow next time. When such an overflow happens there is no submersion buffer, so the insertion buffer can be made a submersion buffer and a new empty insertion buffer is created.

Since elements from the weak heap may be moved to the subarray reserved for the submersion buffer, we have very little information for where the elements originally in the submersion buffer are (since we do not want to allocate much memory for keeping track of them). However, the key point is to remember where in the weak heap the smallest elements that are still moving upwards are. Let us now consider in detail how the different operations are implemented.

The minimum of the weak heap is at its root, and as before we keep the minimum of the insertion buffer at its first location. For the elements taking part in the submersion process, we maintain an explicit pointer indicating where their minimum is. As before, we maintain a minimum indicator that keeps track of in which of the three parts the overall minimum resides. Because the minimum indicator is available, the *minimum* operation can be easily accomplished in $O(1)$ worst-case time without performing any element comparisons.

In *insert*, the given element is appended to the insertion buffer and moved to the first location of the buffer if it is the new minimum of the buffer. The minimum indicator is updated if the inserted element is the overall minimum. Once the insertion buffer is full, a new submersion process is initiated. A constant number of coroutine calls are performed to continue embedding the submersion buffer into the weak heap. When no submersion work remains, the coroutine call reduces to a noop. A high-level description of *insert* is given in Figure 10.

Basically, a submersion process carries out a *bulk-insert* operation incrementally. Within the first phase of the *bulk-insert* procedure, we perform *sift-down* operations starting from the nodes of the submersion buffer bottom-up level by level. Two intervals that represent the nodes up to which the *sift-down* operation has been called are maintained. Each such interval is represented by two indices indicating the left and right nodes in the interval, call them (ℓ_1, r_1) and (ℓ_2, r_2). Note that these two intervals are at two consecutive levels of the heap, and that the parent of the right node of one interval has an index that is one less than the left node of the second interval, i.e. $\lfloor r_2/2 \rfloor = \ell_1 - 1$. We call these two intervals the *frontier*. As a special case, at some steps there may exist only one interval (ℓ, r) on the frontier. In addition to the frontier we also maintain other local variables that keep track of the nodes which *distinguished-ancestor*, *sift-down*, and *sift-up* in progress are currently handling. To sum up, we only need a constant number of machine words to record the state of the incremental submersion process. Once a *sift-down* operation is completed, the frontier is updated as follows. If there are two intervals, the upper interval is extended by $\ell_1 - 1$ and the lower interval shrinks by removing r_2 and possibly $r_2 - 1$ if both of them are siblings. If there is only one interval, it shrinks by removing ℓ and possibly $\ell + 1$, and a new interval that contains only one node (the index of which is $\lfloor \ell/2 \rfloor$) is added to the frontier. In the sequel, all the descendants of

```
class two-buffer-weak-heap

a: element[ ]
r: bit[ ]
n: index
n′: index
n″: index
submersion-minimum: index
overall-minimum: index
state: integer
κ: integer
frontier: two pairs of indices
other local variables: a couple of indices

method insert(x: element)
assert a_0..a_{n′−1} and r_0..r_{n′−1} form a weak heap; a_{n′}..a_{n″−1} is the
submersion buffer and a_{n″}..a_{n−1} the insertion buffer
a_n ← x
r_n ← 0
if a_n < a_{n″}
|  swap(a_{n″}, a_n)
|  if a_{n″} < a_{overall-minimum}
|  |  overall-minimum ← n″
if n − n″ ≥ ⌊lg(1 + n″)⌋
|  state ← 1
|  n′ ← n″
|  n″ ← n + 1
|  submersion-minimum ← n′
n ← n + 1
repeat κ times
|  state ← submersion-step(state)
```

Figure 10: Worst-case constant-time insertion.

the nodes at the frontier constitute the *submersion area*.

To optimize the number of element comparisons needed for the *extract-min* operation, we would like the minimum of the submersion area to be located on the frontier. For that, we substitute the *sift-down* procedure for weak heaps with the one introduced by Williams [15] for binary heaps. Although the work done by Williams' *sift-down* is almost twice as much, we still perform a constant amount of work incrementally with every *insert* operation.

For *borrow*, the last element of the array is extracted from the structure, unless n is greater than one and the last element is the overall minimum—a case that may happen if the insertion buffer contains one element or if the insertion buffer is empty and the submersion buffer contains one element. In

this case, we swap this element with the element at the root of the weak heap, update the minimum indicator, and extract the last element from the structure. Thus, *borrow* takes $O(1)$ worst-case time and involves no element comparisons.

In *extract-min* we handle three cases depending on in which of the three parts the overall minimum is. We emphasize that for all cases the running time is $O(\lg n)$ and the number of element comparisons is at most $\lg n + O(1)$:

1. If the overall minimum is at the root of the weak heap, it is removed, a replacement element is found using *borrow*, and the weak-heap ordering is reestablished by a *sift-down* operation. If this operation meets the frontier, the operation is stopped at the level above the frontier and the task to reestablish the weak-heap ordering is left for the submersion process. The minimum indicator is updated using two additional element comparisons.
2. If the overall minimum is in the insertion buffer, it is removed from there after being swapped with the last element. A new minimum is found by scanning the elements of the insertion buffer and then swapped with the element at the first position of the buffer. The minimum indicator is updated using two additional element comparisons.
3. The overall minimum is maintained by the submersion process:
   (a) Consider the case where the submersion process is in its first phase. In this case, the minimum is removed and a replacement element is found using *borrow*. After the replacement, the heap ordering below this location is enforced by performing Williams' *sift-down*. The new minimum of the submersion area is then found by scanning the nodes on the frontier. Assume that the highest node on the frontier has reached height h, where $h \in \{1, 2, \ldots, \lg \lg n + 1\}$. Williams' *sift-down* requires at most $2h + O(1)$ element comparisons. On the other hand, scanning the frontier for the new minimum requires at most $(\lg n)/2^{h-1} + O(1)$ element comparisons. Finally, as above, the minimum indicator is updated.
   (b) Consider the case where the submersion process is in its second phase, where the frontier comprises (at most) two nodes x and y. That is, at this point of time one of the two *distinguished-ancestor*, *sift-down*, or *sift-up* calls is being executed starting from either x or y. If a *distinguished-ancestor* call has not been finished, we finish it; this involves no element comparisons. If a *sift-down* operation is in progress, the minimum of the submersion area is at either x, y, or one of the distinguished ancestors of these two nodes. First, we delete the minimum and replace it with a borrowed element. If the minimum was at x or y, we perform Williams' *sift-down* starting from the replacement node. Otherwise, to remedy the weak-heap ordering, we perform a *sift-down* operation starting from the ancestor that had the minimum. The new minimum of the submersion area is then located among x, y, and the two distinguished ancestors. If a *sift-up* operation is in progress, we have reached an ancestor of x and an ancestor of y; call them x' and y'. Now the minimum of the

submersion area is at either x' or y'. We delete the minimum, replace it with a borrowed element, and apply a *sift-down* operation starting from the replacement node to remedy the weak-heap ordering. We then locate the new minimum of the submersion area among x' and y'. After the actual deletion, the minimum indicator is also updated.

The above discussion leads to the results that can be summarized as follows.

Theorem 2. *Let n be the number of elements in the data structure prior to each operation. A two-buffer weak heap supports minimum, borrow, and 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. The amount of extra (in addition to the element array) space used is $n + O(w)$ bits, where w is the word size.*

We have shown in [8] that, for the amortized *bulk-insert* solution, the amount of work done in the submersion process is proportional to the size of the submersion buffer when the process started, i.e. it is $O(\lg n)$. We have only showed that there exists a constant κ such that the worst-case number of instructions to be executed in connection with each *insert* is bounded by κ. However, we have not specified exactly how many instructions are executed in connection with each *insert* in the amortized sense. For our tests we determined the value of the constant κ experimentally.

Since the programming language we used (C++) did not provide coroutines, we had to simulate them by hand. We did this by refactoring the existing code for the *bulk-insert* operation. First, we removed all function calls by inlining the functions. Second, we divided the program into basic blocks; a *basic block* being a piece of code that contains at most one branch or branch target. A branch target starts a block if exists, and a branch ends a block if exists. One can see the basic blocks as *states* of a finite state automaton. Third, we represented each state as a small integer (state 1 being the initial state) and maintained the current state in a variable. Fourth, we implemented the submersion step as a function that has one big **switch** statement where the current state is used for branching to get to the beginning of a particular basic block. In the program, **if** statements were used as guards so that the return value of the submersion step indicates the next state. That is, in each *insert* this function is called a constant number of times to execute one basic block per call.

Again, we used weak-heap construction to benchmark the performance of the *insert* algorithm. As the numbers in Table 14 indicate, the worst-case constant-time insertion algorithm is significantly more complicated than the other algorithms considered in this paper. The obtained heap-construction program was more than a factor of three slower than the corresponding program relying on the amortized constant-time insertion algorithm. The program is even slower than the standard algorithm for constructing a weak heap in a worst-case scenario. So be warned: In practice $O(1)$ is not always better than $O(\lg n)$!

Table 14: Performance of weak-heap construction employing the worst-case constant-time insertion algorithm; the elements were given in random order. All measurement results were divided by n, the last column by n/B; time is in nanoseconds.

32 bit n	**time**	**instructions**	**comparisons**	**moves**	**mispredictions**	**misses**
2^{10}	417.3	836	4.14	3.92	23.38	1.25
2^{15}	373.4	791	4.12	3.82	21.14	1.25
2^{20}	247.4	741	3.92	3.73	19.06	2.43
2^{25}	333.8	718	3.83	3.66	17.78	2.50

64 bit n	**time**	**instructions**	**comparisons**	**moves**	**mispredictions**	**misses**
2^{10}	118.1	609	4.14	3.92	23.29	1.13
2^{15}	106.4	577	4.12	3.82	20.94	1.13
2^{20}	101.5	541	3.92	3.73	18.90	1.85
2^{25}	99.6	525	3.83	3.66	17.66	2.24

4. Conclusion

The weak heap is an amazingly simple and powerful structure. If perfectly balanced, weak heaps resemble heap-ordered binomial trees [13]. A weak heap is implemented in an array with extra bits that are used for subtree swapping. Binomial-tree parents are distinguished ancestors in the weak-heap setting.

We showed that, for Dutton's weak-heap construction [6], the distinguished ancestor of any node in a weak heap can be located in constant worst-case time. We also provided branch-optimized, cache-optimized, and move-optimized procedures for building a weak heap. Contrary to binary heaps, repeated insertions in a weak heap require an amortized constant number of element comparisons per inserted element; we proved that a sequence of n insertions in an initially empty weak heap requires at most $3.5n + O(\lg^2 n)$ element comparisons. By attaching a submersion buffer and an insertion buffer and by processing weak-heap-order violating nodes in the submersion buffer incrementally, *insert* was made to run in constant worst-case time while retaining at most $\lg n + O(1)$ element comparisons and $O(\lg n)$ time for *extract-min*.

As far as we know, no implementation of the multipartite priority queues [11] exists. In contrast, we implemented *insert* for the two-buffer weak-heap structure, but we have to admit that it is slow. An obvious open question is how to reduce the (constant) number of element comparisons performed by *insert* and the additive constant in the number of element comparisons performed by *extract-min*. Compared to the multipartite priority queues, we reduced the amount of extra space needed from $O(n)$ words to $n + O(w)$ bits.

We would like to end with a warning: The experimental results reported here depend on the computing environment where the experiments were performed

and on the type of data used as input. In our experimental setup, element comparisons and element moves were cheap, and branch mispredictions and cache misses were expensive. When some of these settings change, the overall picture may drastically change.

References

[1] S. Alstrup, T. Husfeldt, T. Rauhe, and M. Thorup, Black box for constant-time insertion in priority queues, *ACM Transactions on Algorithms* **1**, 1 (2005), 102–106.

[2] J. Bojesen, J. Katajainen, and M. Spork, Performance engineering case study: Heap construction, *ACM Journal of Experimental Algorithmics* **5** (2000), Article 15.

[3] A. Bruun, S. Edelkamp, J. Katajainen, and J. Rasmussen, Policy-based benchmarking of weak heaps and their relatives, *Proceedings of the 9th International Symposium on Experimental Algorithms*, *Lecture Notes in Computer Science* **6049**, Springer-Verlag (2010), 424–435.

[4] J. Chen, S. Edelkamp, A. Elmasry, and J. Katajainen, In-place heap construction with optimized comparisons, moves, and cache misses, *Proceedings of the 37th International Symposium on Mathematical Foundations of Computer Science*, *Lecture Notes in Computer Science* **7464**, Springer-Verlag (2012), 259–270.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd Edition, The MIT Press (2009).

[6] R. D. Dutton, Weak-heap sort, *BIT* **33**, 3 (1993), 372–381.

[7] S. Edelkamp, A. Elmasry, and J. Katajainen, A catalogue of weak-heap programs, CPH STL Report **2012-2**, Department of Computer Science, University of Copenhagen (2012).

[8] S. Edelkamp, A. Elmasry, and J. Katajainen, The weak-heap data structure: Variants and applications, *Journal of Discrete Algorithms* **16** (2012), 187–205.

[9] S. Edelkamp and P. Stiegeler, Implementing Heapsort with $n \log n - 0.9n$ and Quicksort with $n \log n + 0.2n$ comparisons, *ACM Journal of Experimental Algorithmics* **7** (2002), Article 5.

[10] S. Edelkamp and I. Wegener, On the performance of Weak-Heapsort, *Proceedings of the 17th Annual Symposium on Theoretical Aspects of Computer Science*, *Lecture Notes in Computer Science* **1770**, Springer-Verlag (2000), 254–266.

[11] A. Elmasry, C. Jensen, and J. Katajainen, Multipartite priority queues, *ACM Transactions on Algorithms* **5**, 1 (2008), Article 14.

[12] R. W. Floyd, Algorithm 245: Treesort 3, *Communications of the ACM* **7**, 12 (1964), 701.

[13] J. Vuillemin, A data structure for manipulating priority queues, *Communications of the ACM* **21**, 4 (1978), 309–315.

[14] I. Wegener, Bottom-Up-Heapsort, a new variant of Heapsort beating, on an average, Quicksort (if n is not very small), *Theoretical Computer Science* **118**, 1 (1993), 81–98.

[15] J. W. J. Williams, Algorithm 232: Heapsort, *Communications of the ACM* **7**, 6 (1964), 347–348.