A Catalogue of Algorithms for Building Weak Heaps*

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

[1] Faculty 3—Mathematics and Computer Science, University of Bremen, PO Box 330 440, 28334 Bremen, Germany
`edelkamp@tzi.de`
[2] Department of Computer Science, University of Copenhagen, Universitetsparken 1, 2100 Copenhagen East, Denmark
`{elmasry,jyrki}@diku.dk`
[3] Computer and Systems Engineering Department, Alexandria University, Alexandria 21544, Egypt

Abstract. An array-based weak heap is an efficient data structure for realizing an elementary priority queue. In this paper we focus on the construction of a weak heap. Starting from a straightforward algorithm, we end up with a catalogue of algorithms that optimize the standard algorithm in different ways. As the optimization criteria, we consider how to reduce the number of instructions, branch mispredictions, cache misses, and element moves. We also consider other approaches for building a weak heap: one based on repeated insertions and another relying on a non-standard memory layout. For most of the algorithms considered, we also study their effectiveness in practice.

1 Introduction

In its elementary form, a priority queue is a data structure that stores a collection of elements and supports the operations *construct*, *find-min*, *insert*, and *extract-min*. In applications, for which this set of operations is sufficient, the basic selection that the users have to make is to choose between binary heaps [9] and weak heaps [3]. Both of these data structures are known to perform well, and the difference in performance is quite small in typical cases.

Most library implementations are based on binary heaps. However, one reason why a user might select 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 we summarize the results known for these two data structures.

In [4] we showed that, for weak heaps, the cost of *insert* can be improved to an amortized constant. The idea is to use a buffer that supports constant-time insertion. A new element is inserted into the buffer as long as the number of its elements is below the threshold. Once the threshold is reached, a bulk insertion is performed by moving all elements of the buffer to the weak heap. This modification increases the number of element comparisons per *extract-min* by one.

Table 1. The worst-case number of element comparisons performed by two elementary priority queues; n denotes the number of elements stored in the data structure prior to the operation in question.

data structure	*construct*	*find-min*	*insert*	*extract-min*
binary heap [6, 9]	$2n$	0	$\lceil \lg n \rceil$	$2\lceil \lg n \rceil$
weak heap [3]	$n-1$	0	$\lceil \lg n \rceil$	$\lceil \lg n \rceil$

In this paper we study the construction of a weak heap in more detail. The standard algorithm for building a weak heap 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 similar fundamental algorithms. For some applications the proposed optimizations may be significant although we do not consider any concrete applications per se.

Our catalogue of algorithms for building a weak heap include the following:

Instruction optimization: We utilize bit-manipulation operations on the word level to find the position of the least-significant 1-bit fast. 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 optimization that reduces the number of branch mispredictions from $O(n)$ to $O(1)$. On modern computers having long pipelines, a branch misprediction 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 performed from $3n$ to $2n$. In the meantime, we keep the number of element comparisons performed unchanged.

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

New memory layout: We investigate an alternative weak-heap array embedding that does not need any complex ancestor computations.

The experiments discussed in the paper were carried out on a laptop (model Intel® Core$^{\text{TM}}$2 CPU P8700 @ 2.53GHz) running under Ubuntu 11.10 (Linux kernel 3.0.0-16-generic) using `g++` compiler (`gcc` version 4.6.1) with optimization level `-O3`. The size of L2 cache was about 3 MB and that of the main memory 3.8 GB. Input elements were 4-byte integers and reverse bits occupied one byte each. All execution times were measured using the function `gettimeofday` in `sys/time.h`. Other measurements were done using the tools available in `valgrind` (version 3.6.1). For a problem of size n, each experiment was repeated $2^{26}/n$ times and the average was reported.

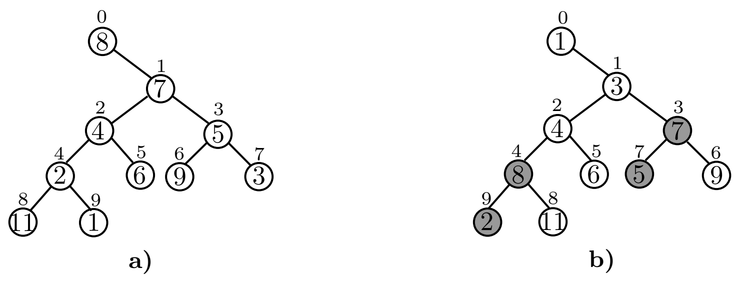

Fig. 2. a) An input of 10 integers and **b)** a weak heap constructed by the standard algorithm. The reverse bits are set for grey nodes.

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 also support *delete* and *decrease* operations (see [2, 4]).

2 The Standard Weak-Heap Construction Procedure

A *weak heap* (see Fig. 2) 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 the right subtree of this node.

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 the *weak-heap ordering* or half-tree ordering. In particular, this property enforces no relation between an element in a node and those stored in the left subtree of this node.

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 $r_i \leftarrow 1 - r_i$.

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 use *d-ancestor*(j) to denote the index of such ancestor. By 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 bottom-up construction of a weak heap (see Fig. 3) the nodes are visited one by one in reverse order, and the two weak heaps rooted at a node and its distinguished ancestor are joined. It has been shown, for example in [5], 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.

```
procedure: d-ancestor($j$: index)
while ($j$ bitand 1) $= r_{\lfloor j/2 \rfloor}$
    $j \leftarrow \lfloor j/2 \rfloor$
return $\lfloor j/2 \rfloor$

procedure: join($i, j$: indices)
if $a_j < a_i$
    swap($a_i, a_j$)
    $r_j \leftarrow 1 - r_j$

procedure: construct($a$: array of $n$ elements, $r$: array of $n$ bits)
for $i \in \{0, 1, \ldots, n-1\}$
    $r_i \leftarrow 0$
for $j = n-1$ to 1 step $-1$
    $i \leftarrow$ d-ancestor($j$)
    join($i, j$)
```

Fig. 3. Standard construction of a weak heap.

3 Instruction Optimization: Accessing Ancestors Faster

The number of instructions executed by the standard algorithm can be reduced by observing that the reverse bits are initialized to 0 and set bottom-up while scanning the nodes. Therefore, the distinguished ancestor can be computed from the array index by considering the position of the least-significant 1-bit. On most computers this position can be computed by using the native primitive operation that counts the number of trailing 0-bits in a word.

Assuming the availability of the needed hardware support, this refinement makes the analysis of the algorithm straightforward: Each distinguished ancestor is accessed in constant worst-case time, and for each node (except the root) one element comparison and at most one element swap are performed.

```
**procedure**: $d\text{-}ancestor(j\text{: index})$
$z \leftarrow trailing\_zero\_count(j)$;
**return** $j$ >> $(z+1)$
```

Fig. 4. A faster way of finding the distinguished ancestor of a node.

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 both the built-in functions *__builtin_ctz* and *__builtin_popcount* provided by our compiler (g++); in out test environment the former led to a superior performance. As seen from the numbers in Table 5, the instruction optimization made the program faster, and the numbers in Table 6 verify that the number of instructions executed actually reduced.

Table 5. Standard vs. instruction-optimized constructions; execution time divided by n in nanoseconds; the elements were given in random order.

n	**standard**	**instruction optimized**
		__builtin_ctz
2^{10}	10.49	7.86
2^{15}	10.26	7.49
2^{20}	10.61	7.83
2^{25}	10.96	8.16

Table 6. Standard vs. instruction-optimized constructions; number of instructions executed divided by n; the elements were given in random order.

n	**standard**	**instruction optimized**	
		__builtin_ctz	*__builtin_popcount*
2^{10} 2^{15} 2^{20} 2^{25}	22.5	12.5	16.5

4 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 weak-heap construction algorithm has few conditional branches. By our instruction optimization we already removed the loop used for computing the distinguished ancestors. 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 is incurred. Hence, the main issue to guarantee $O(1)$ branch mispredictions is to remove the **if** statement in the *join* procedure. To do that, we replace the conditional branch with arithmetic operations.

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

Fig. 7. Joining two weak heaps without a conditional branch.

As shown in Table 8, combining this optimization with that described in the previous section, in our test environment the running times again improved. In fact, these are the best running times among those of the algorithms presented in this paper. As confirmed in Table 9, for the branch-optimized version the number of branch mispredictions incurred is indeed negligible.

Table 8. Instruction-optimized vs. branch-optimized constructions; execution time divided by n in nanoseconds; the elements were given in random order.

n	**instruction optimized**	**branch optimized**
2^{10}	7.86	6.28
2^{15}	7.49	6.39
2^{20}	7.83	6.72
2^{25}	8.16	7.08

5 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 fix the weak-heap ordering by considering each node and its right subtree. The idea comes from Floyd's algorithm for constructing binary heaps [6]. To mimic this algorithm we need the *sift-down* procedure, which is explained next.

Table 9. Standard vs. instruction-optimized vs. branch-optimized constructions; total number of mispredicted branches; the elements were given in random order.

n	**standard**	**instruction optimized**	**branch optimized**
2^{10}	1 061	512	1
2^{15}	34 171	16 385	1
2^{20}	1 093 963	524 110	2
2^{25}	35 005 433	16 776 271	34

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

With *sift-down* in hand, a weak heap can be constructed by calling the procedure on every node starting from the the lower levels upwards (see Fig. 10). Unfortunately, the running times achieved for this method are not satisfactory, as indicated in Table 11. Compared to the standard method, the slowdown is more significant for large values of n. We relate this behaviour to cache effects.

```
procedure: sift-down($j$: index)
$k \leftarrow 2j + 1 - r_j$
while $2k + r_k < n$
    $k \leftarrow 2k + r_k$
while $k \neq j$
    join($j, k$)
    $k \leftarrow \lfloor k/2 \rfloor$

procedure: construct($a$: array of $n$ elements, $r$: array of $n$ bits)
for $i \in \{0, 1, \ldots, n-1\}$
    $r_i \leftarrow 0$
for $j = \lfloor n/2 \rfloor - 1$ to $0$ step $-1$
    sift-down($j$)
```

Fig. 10. An alternative construction of a weak heap.

Table 11. Standard vs. alternative constructions; execution time divided by n in nanoseconds; the elements were given in random order.

n	**standard**	**alternative**
2^{10}	10.49	11.16
2^{15}	10.26	11.66
2^{20}	10.61	18.44
2^{25}	10.96	19.96

6 Cache Optimization: Depth-First Construction

To avoid the bad cache performance of the previous method, the nodes of the heap should be visited in depth-first rather than in breadth-first order. This idea, applied for binary heaps in [1], improves the locality of accesses as the traversal tends to stay longer at nearby memory locations. The number of element comparisons obviously remains unchanged. A recursive procedure is given in Fig. 12.

```
**procedure**: $df\text{-}construct(i, j\text{: indices})$
**if** $j < \lfloor n/2 \rfloor$
    $df\text{-}construct(j, 2j + 1)$
    $df\text{-}construct(i, 2j)$
$join(i, j)$

**procedure**: $construct(a\text{: array of } n \text{ elements}, r\text{: array of } n \text{ bits})$
**for** $i \in \{0, 1, \ldots, n - 1\}$
    $r_i \leftarrow 0$
**if** $n > 1$
    $df\text{-}construct(0, 1)$
```

Fig. 12. Recursive depth-first weak-heap construction.

Although we do not explicitly use the *sift-down* procedure here, we would like to point out that this method is a cache-optimized version of the method described in the previous section. Following the guidelines given in [1], it is not difficult to avoid recursion when implementing the procedure.

The running times given in Table 13 indicate that we are far from those of the standard method. On the other hand, the cache-miss rates given in Table 14 indicate that the depth-first construction has a better cache behaviour.

7 Move Optimization: Trading Swaps for Delayed Moves

Now we implement the aforementioned depth-first construction in a different way (see Fig. 15). The idea is to walk down the left spine of the child of the root

Table 13. Standard vs. cache-optimized constructions; execution time divided by n in nanoseconds; the elements were given in random order.

n	**standard**	**cache optimized**
2^{10}	10.49	15.05
2^{15}	10.26	15.08
2^{20}	10.61	15.21
2^{25}	10.96	15.20

Table 14. 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.

n	**standard**	**instruction optimized**	**alternative**	**cache optimized**
2^{10}	1.25	1.25	1.25	1.25
2^{15}	1.25	1.25	1.25	1.25
2^{20}	1.72	1.52	6.38	1.25
2^{25}	2.47	2.23	7.36	1.25

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 the weak-heap 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 (element 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

```
procedure: df-construct($i$: index)
$j \leftarrow 2i+1$
while $j < \lfloor n/2 \rfloor$
    df-construct($j$)
    $j \leftarrow 2j$
sift-down($i$)

procedure: construct($a$: array of $n$ elements, $r$: array of $n$ bits)
for $i \in \{0, 1, \ldots, n-1\}$
    $r_i \leftarrow 0$
if $n > 1$
    df-construct(0)
```

Fig. 15. Another implementation for depth-first weak-heap construction.

by Wegener [8] 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. The $2n$ bound follows. See Tables 16 and 17.

Table 16. Standard vs. move-optimized constructions; execution time divided by n in nanoseconds; the elements were given in random/decreasing orders.

n	**standard**		**move optimized**	
	random	**decreasing**	**random**	**decreasing**
2^{10}	10.49	7.86	22.60	19.91
2^{15}	10.26	7.60	22.02	18.43
2^{20}	10.61	7.95	22.14	18.52
2^{25}	10.96	8.30	22.11	18.53

Table 17. Standard vs. move-optimized constructions; number of moves per element; the elements were given in random/decreasing orders.

n	**standard**		**move optimized**	
	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

8 Repeated Insertions: Non-Linear Work, Yet a Linear Number of Element Comparisons

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

```
procedure: sift-up(j: index)
while $j \neq 0$
    $i \leftarrow$ d-ancestor($j$)
    before $\leftarrow r_j$
    join($i, j$)
    if before $= r_j$
        break
    $j \leftarrow i$

procedure: construct(a: array of n elements, r: array of n bits)
for $k = 1$ to $n - 1$
    $r_k \leftarrow 0$
    if ($k$ bitand 1)     (*)
        $r_{\lfloor k/2 \rfloor} \leftarrow 0$
    sift-up($k$)
```

Fig. 18. Constructing a weak heap by repeated insertions.

the two elements and repeat for the new location of e using the *join* procedure. 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 Fig. 18), we observed that, while the execution time increased with n, the number of element comparisons stayed constant per element for an increasing value of n (see Tables 19 and 20). As the worst-case input for the experiments we used the sequence of the form $\langle 0, n-1, n-2, \ldots, 1 \rangle$ as adviced in [5]. 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 first insertion, resulting in $n - 1$ terminal element comparisons during the whole construction. All other element comparisons are *non-terminal.*

Next we caclulate an upper bound for 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 elements at x and the distinguished ancestor of x throughout the process. For such a comparison to take place an element should have been inserted in the right subtree of x (may include 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 elements at x and its distinguished ancestor. That is, d can take values 0, 1, etc.

For $d = 0$, the elements 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 the program. When the inserted node is the only child of 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 elements 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 the distinguished ancestor of x until an element is inserted below x at distance $d + 1$ from x. In conclusion, the node x can be charged with at most one element comparison for each level of both its subtrees. 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; that is at most $2h - 2$ times.

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 preceeding paragraph it follows that the number of non-terminal element comparisons is bounded by $\lceil n/2 \rceil + \sum_{h=2}^{\lceil \lg n \rceil + 1} (2h - 2) \cdot \lceil n/2^h \rceil < 2.5n + O(\lg^2 n)$. Adding the $n - 1$ terminal element comparisons, the total number of element comparisons performed by all n insertions is at most $3.5n + O(\lg^2 n)$. $\square$

Observe that the number of element comparisons and that of element moves go hand in hand so that the number of element moves performed is also linear. That is, the running time is wasted in distinguished-ancestor calculations and, unfortunately, we cannot take any shortcuts since some of the reverse bits may be set above an insertion point.

Table 19. Standard vs. repeated-insertion constructions; execution time divided by n in nanoseconds; the elements were given in random/special worst-case orders.

n	**standard**		**repeated insertions**	
	random	**special**	**random**	**special**
2^{10}	10.49	7.86	31.46	45.91
2^{15}	10.26	7.60	32.68	64.09
2^{20}	10.61	7.95	33.04	78.00
2^{25}	10.96	8.30	33.18	89.36

9 New Memory Layout: Less Work, Different Outcome

There are other possible array embeddings than the standard one (where the children of the node at location i are at locations $2i$ and $2i + 1$). Consider the

Table 20. Standard vs. repeated-insertion constructions; number of element comparisons divided by n; the elements were given in random/special worst-case orders.

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	1	1.77	3.49

following layout: For a node a_i whose depth in the tree is d, its right child is stored at location $i + 2^{d-r_i}$ and its left child at location $i + 2^{d-1+r_i}$. As with the standard layout, reverse bits are used to swap the two subtrees. To access the parent of a node a_i whose depth is d, we need an extra check. If $i \geq 2^{d-1} + 2^{d-2}$, the parent is at location $i - 2^{d-1}$; otherwise, the parent is at location $i - 2^{d-2}$. If the depth is not available, it can be computed in constant time using the number of leading zeros. The main advantage of this dual layout is that the construction algorithm does not need distinguished-ancestor calculations.

```
procedure: $dual\text{-}construct(a$: array of $n$ elements, $r$: array of $n$ bits)
$\ell \leftarrow n$
while $\ell > 1$
    for $i \in \{0, \ldots, \lfloor \ell/2 \rfloor\}$
        $join(i, \lfloor \ell/2 \rfloor + i)$
    $\ell \leftarrow \lfloor \ell/2 \rfloor$
```

Fig. 21. Constructing a weak heap for the new memory layout.

10 Conclusion

The weak heap is an amazingly simple and powerful structure. If perfectly balanced, weak heaps resemble heap-ordered binomial trees [7]. A weak heap is implemented in an array with extra bits that are used for subtree swapping.

Binomial-tree parents are distinguished ancestors in a weak-heap setting. We showed that, for Dutton's construction of weak heaps [3], a distinguished ancestor can be computed in constant wost-case time. We also provided depth-first weak-heap building procedures. The standard memory layout is fastest for accessing neighbouring nodes, but other layouts may lead to a faster construction.

Contrary to binary heaps, repeated insertions lead to a 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.

We would like end this paper with a warning: The experimental results reported depended on the 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 conditions will change, the overall picture may drastically change.

Source code

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

References

1. Bojesen, J., Katajainen, J., Spork, M.: Performance engineering case study: Heap construction. ACM J. Exp. Algorithmics 5, Article 15 (2000)
2. Bruun, A., Edelkamp, S., Katajainen, J., Rasmussen, J.: Policy-based benchmarking of weak heaps and their relatives. In: Proc. 9th Int. Symp. on Exp. Algorithms. Lecture Notes Comput. Sci., vol. 6049, pp. 424–435. Springer-Verlag, Berlin/Heidelberg (2010)
3. Dutton, R.D.: Weak-heap sort. BIT 33(3), 372–381 (1993)
4. Edelkamp, S., Elmasry, A., Katajainen, J.: The weak-heap data structure: Variants and applications. J. Discrete Algorithms 16, 187–205 (2012)
5. Edelkamp, S., Wegener, I.: On the performance of weak-heapsort. In: Proc. 17th Annual Symp. on Theoret. Aspects of Comput. Sci. Lecture Notes Comput. Sci., vol. 1770, pp. 254–266. Springer-Verlag, Berlin/Heidelberg (2000)
6. Floyd, R.W.: Algorithm 245: Treesort 3. Commun. ACM 7(12), 701 (1964)
7. Vuillemin, J.: A data structure for manipulating priority queues. Commun. ACM 21(4), 309–315 (1978)
8. 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)
9. Williams, J.W.J.: Algorithm 232: Heapsort. Commun. ACM 7(6), 347–348 (1964)