In-Place Heap Construction with Optimized Comparisons, Moves, and Cache Misses*

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

[1] Department of Computer Science, Electrical and Space Engineering, Luleå University of Technology, 971 87 Luleå, Sweden
[2] Faculty 3—Mathematics and Computer Science, University of Bremen, PO Box 330 440, 28334 Bremen, Germany
[3] Department of Computer Science, University of Copenhagen, Universitetsparken 1, 2100 Copenhagen East, Denmark

Abstract. We show how to build a binary heap in-place in linear time by performing $\sim 1.625n$ element comparisons, at most $\sim 2.125n$ element moves, and $\sim n/B$ cache misses, where n is the size of the input array, B the capacity of the cache line, and $\sim f(n)$ approaches $f(n)$ as n grows. The same bound for element comparisons was derived and conjectured to be optimal by Gonnet and Munro; however, their procedure requires $\Theta(n)$ pointers and does not have optimal cache behaviour. Our main idea is to mimic the Gonnet-Munro algorithm by converting a navigation pile into a binary heap. To construct a binary heap in-place, we use this algorithm to build bottom heaps of size $\Theta(\lg n)$ and adjust the heap order at the upper levels using Floyd's *sift-down* procedure. On another frontier, we compare different heap-construction alternatives in practice.

1 Introduction

The *binary heap*, introduced by Williams [16], is a binary tree in which each node stores one element. This tree is almost complete in the sense that all the levels are full, except perhaps the last level where elements are stored at the leftmost nodes. A binary heap is said to be *perfect* if it stores $2^k - 1$ elements, for a positive integer k. The elements are maintained in (min-)*heap order*, i.e. for each node the element stored at that node is not larger than the elements stored at its (at most) two children. A binary heap can be conveniently represented in an array where the elements are stored left-to-right in the level order of the tree.

In this paper we consider the problem of constructing a binary heap of n elements given in an array. Our objective is to do the construction in-place, i.e. using $O(1)$ words of additional memory. We assume that each word stores $O(\lg n)$ bits. The original algorithm of Williams constructs a binary heap in-place in $\Theta(n \lg n)$ time. Soon after, Floyd [6] improved the construction time to $\Theta(n)$ with at most $2n$ element comparisons. These classical results are covered by most textbooks on algorithms and data structures (see, e.g. [4, Chapter 6]).

* © 2012 Springer-Verlag. This is the authors' version of the work. The original publication is available at `www.springerlink.com`.

Table 1. The number of element comparisons required by different heap-construction algorithms. The input is of size n; the average-case results assume that the input is a random permutation of n distinct elements.

Inventor	**Abbreviation**	**Worst case**	**Average case**	**Extra space**
Floyd [6]	**Alg. F**	$2n$	$\sim 1.88n$	$\Theta(1)$ words
Gonnet & Munro [8]	**Alg. GM**	$1.625n$	$1.625n$	$\Theta(n)$ words
McDiarmid & Reed [12]	**Alg. MR**	$2n$	$\sim 1.52n$	$\Theta(n)$ bits
Li & Reed [11]	**Lower bound**	$\sim 1.37n$	$\sim 1.37n$	$\Omega(1)$ words

In the literature, several performance indicators have been considered: the number of element comparisons, the number of element moves, and the number of cache misses. Even though Floyd's heap-construction algorithm is asymptotically optimal, for most performance indicators the exact optimal complexity bounds are still unknown. In Table 1 we summarize the best known bounds on the number of element comparisons when constructing a heap. By an element move we mean any assignment of elements to variables and array locations. Hence, a swap of two array elements is counted as three element moves and a cyclic rotation of k array elements is counted as $k + 1$ element moves.

Without loss of generality, we assume that our heaps are perfect. As pointed out for example in [3], the nodes that do not root a perfect heap are located on the path from the last leaf to the root. This means that a heap of size n can be partitioned into at most $\lfloor \lg n \rfloor$ perfect subheaps. After building these subheaps, combining them can be done bottom-up in $O(\lg^2 n)$ time, and hence this does not affect the constant of the leading term in the complexity expressions.

Our main contribution is a simple technique for making existing algorithms for heap construction to run in-place. First, we reduce the amount of extra space used by the algorithm to a linear number of bits. Second, we use such an algorithm to build heaps of size $\Theta(\lg n)$ at the bottom of the input tree; we keep the needed bits in a constant number of words. Third, we combine these bottom heaps by exploiting the *sift-down* procedure of Floyd's algorithm at the top nodes of the tree. The key observation is that the work done by all *sift-down* calls is sublinear. We apply this approach for both the algorithm of Gonnet and Munro [8] and that of McDiarmid and Reed [12]. The inventors believe that their algorithms are optimal with respect to the number of element comparisons; the first in the worst-case sense and the second in the average-case sense.

In our in-place variant of the **GM** algorithm, we also optimize the number of element moves. For each bottom tree, we start by building a navigation pile at the end of the element array [9]; this technique is used by Kronrod [10]. To optimize the number of element moves when converting a navigation pile to a binary heap, we employ the hole technique that is also used by Floyd [6].

Another consequence of our in-place construction is that both algorithms can be modified—without affecting the number of element comparisons and element moves—such that the cache behaviour of the algorithms is almost optimal under reasonable assumptions. That is, without any knowledge about the size of cache blocks (B) and the size of fast memory (M), the algorithms incur about n/B

cache misses. Here we rely on an improvement proposed by Bojesen et al. [1] showing how Floyd's algorithm can be made cache oblivious. For the algorithms involving linear extra space, this kind of optimality cannot be achieved due to the cache misses incurred when accessing the additional memory.

2 Heap Construction with a Navigation Pile

The **GM** algorithm [8] builds a binary heap on $2^k - 1$ elements in two phases: First a heap-ordered binomial tree [13] of size 2^k is constructed, and then this binomial tree is converted into a binary heap plus one *excess element* that is discarded. Since a binomial tree is a pointer-based data structure, we considered some related data structures that are more space-economical like a weak heap [5] and a navigation pile [9]. We decided to use the latter as our basic building block since it involves a fewer number of element moves for the overall construction.

2.1 Building a navigation pile

Consider an array of elements $a_0, \ldots, a_{n-1}$, where $n = 2^k$ for a positive integer k. A *navigation pile* [9] is a compact representation of a tournament tree built above this array. This tournament tree is a complete binary tree of size $2^k - 1$. Extending this tree by the element array, we get a complete binary tree of size $2^{k+1} - 1$. We say that the elements have height 0, all the bottommost nodes of the tournament tree have height 1, and so on; the root has height k.

Let us number the nodes at each level starting from 0. For a node at height h with index i at that level, its parent is at level $h+1$ (if any) and has index $\lfloor i/2 \rfloor$, and its children are at level $h-1$ (if any) and have indices $2i$ and $2i+1$. These definitions are illustrated in Fig. 1.

A node of the tournament tree is said to *span* the leaves of the subtree rooted at that node. For each node we store a reference to the minimum element within the range spanned. If a node has height h, it spans 2^h elements. More precisely, if the index of the node is i at its level, it spans the elements in the range

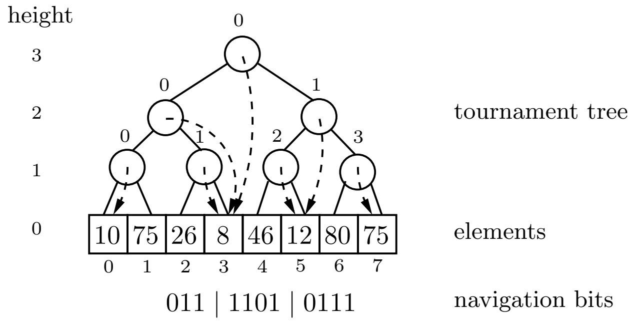

Fig. 1. A navigation pile of size 8; navigation information is visualized with pointers.

$[i \times 2^h \cdots i \times 2^h + 2^h - 1]$. To address any of these elements, we need to know the height of the node, its index, and an offset inside the range (these are h bits called the *navigation bits*). The navigation information for the root uses k bits, that for the children of the root $k-1$ bits, and so on. The total number of navigation bits for all the nodes in the tournament tree is then:

$$2^k \sum_{h=1}^{k} h/2^h < 2n\,.$$

Hence, these bits can be stored in a bit-array of size at most $2n$ bits.

We can traverse the navigation pile starting from the root or from a leaf. During such a traversal we recall the height of the current node, its index at its present level, and the start position of the navigation bits of the present level in the bit array. When this information is available, we can access the parent or the children of the current node such that the same information is available for the accessed node. This can be accomplished by relying on simple arithmetic operations. For more details of implementing navigation piles and the related primitive operations, see [9].

To initialize a navigation pile, we traverse the tree bottom-up level by level and update the navigation information at each node by comparing the elements referred to by its children. Such a construction requires $n-1$ element comparisons and no element moves. In addition, a navigation pile supports the priority-queue operations *insert* and *delete* easily and efficiently [9].

2.2 Converting a navigation pile into a binary heap

Later we shall need a factory that can produce binary heaps of some particular size 2^k-1, where k is a positive integer. For the time being, let us assume that the factory already has a navigation pile on 2^k elements in its *input area* occupying the last locations of the element array. The task is to move the elements one by one from the input area to an *output area* that will, at the end, contain a binary heap of size $2^k - 1$. As another outcome of the procedure, we provide a reference to the location of the excess element in the input area.

For a tournament-tree node in a navigation pile, the first bit of its navigation information tells whether the minimum element among the elements it spans is in the left or the right subtree of that node. We call the subtree in which the minimum element is the *winner subtree*, and the other subtree the *loser subtree.*

As for the **GM** algorithm, our procedure is recursive. The nodes of the navigation pile are visited starting from the root, which is initially the *current input node.* Correspondingly, in the output area, the root of the binary heap to-be-built is initially the *current output node.* If the current input node is visited for the first time, we immediately proceed to its loser subtree and recursively convert it into a binary heap. The right child of the current output node will root the created binary heap. After processing the loser subtree, we move the minimum element w among those spanned by the current input node to the current output node, and then we move the excess element of the loser subtree to the old position of w. Hereafter, we have to update the navigation information on the path from w's old place to the root of the winner subtree. (It is not necessary to update the navigation information at the current input node.) It takes one element comparison per internal node to fix this navigation information. If the current input node is at height k of the navigation pile, we perform $k-1$ element comparisons. (This is similar to an *insert* operation in a navigation pile.) After this fix, the winner subtree can be converted into a binary heap recursively. The output is placed at the left subtree of the current output node.

As the base case, we use $k = 3$ (a navigation pile with 8 elements). The minimum element w of the structure is referred to by the root of the navigation pile and is readily known. Notice also that the loser subtree of the root forms a heap of size 3 plus one excess element. In other words, the minimum element among the elements of the loser subtree is readily known as well. We are only left with determining the minimum element within the winner subtree. In addition to w, there are three other elements in the winner subtree, and the smaller of two of them is already known. It follows that we only need to compare the third element with the smaller of these two elements. Hence, only one element comparison is needed for the base case.

Let us now analyse the performance of this conversion procedure. To convert a navigation pile of size 2^k into a binary heap, we perform two recursive calls for navigation piles of size 2^{k-1}. In addition, we need to call *insert* once to refresh the navigation information in the winner subtree. Let $C(2^k)$ be the number of element comparisons needed to convert a navigation pile of size 2^k into a binary heap plus an excess element. The number of element comparisons performed by *insert* is $k-1$. The next recursive relation follows:

$$\begin{cases} C(8) &= 1, \\ C(2^k) &= 2C(2^{k-1}) + k - 1\,. \end{cases}$$

For $n = 2^k \geq 8$, the solution of this relation is $C(n) = 5/8 \cdot n - \lg n - 1$. Adding the $n-1$ element comparisons needed for the construction of the navigation pile, the total number of element comparisons to build a binary heap on n elements is bounded by $1.625n$.

Let $M(2^k)$ be the number of element moves performed when converting a navigation pile of size 2^k into a binary heap. In the base case, every element except the excess element is moved from the input area to the output area. In the general case, after each recursive call, the minimum element is moved to the output area and the excess element of the loser subtree is moved to the place of the minimum element in the winner subtree. The next recursive relation follows:

$$\begin{cases} M(8) &= 7, \\ M(2^k) &= 2M(2^{k-1}) + 2\,. \end{cases}$$

For $n = 2^k \geq 8$, the solution of this relation is $M(n) = 9/8 \cdot n - 2$, i.e. the number of element moves in this case is bounded by $1.125n$.

Before proceeding, we have to consider one important detail. Since we aim at using this conversion procedure as a subroutine in our in-place algorithm, we must ensure that the recursion is handled only using a constant amount of extra storage. In an iterative implementation of the procedure, we keep track of the current input node (plus all the normal information required when traversing a navigation pile, including the present height), the current output node, and the previous input node. When this information is available, we can deduce whether the current input node is visited for the first, the second, or the third time; and whether we are processing a loser subtree or a winner subtree. So, our iterative implementation does not need any recursion stack.

3 Building a Binary Heap In-Place

Assume that the task is to build a binary heap on n elements; here we need not make any assumptions about n. Let $m = 2^{\lfloor \lg \lg n \rfloor + 1} - 1$. We call all complete binary trees of size m *bottom trees*. The basic idea is to use the algorithm described in the previous section to convert all bottom trees to *bottom heaps* and then ensure the heap order at the upper levels by using the *sift-down* procedure of Floyd's algorithm or any of its variants. The crucial observation is that, by carefully choosing the sizes of the bottom trees, the work done by the *sift-down* calls at the upper levels becomes sublinear. The details are explained next.

To be able to convert the bottom trees into binary heaps, we use the last $m + 1$ locations of the element array as the input area for our factory. The main distinction from our earlier construction is that we do not have an empty output area, and hence we have to use the other bottom trees as the output area. Observe that there is a constant number of (at most three) bottom trees that overlap the input area. These *special bottom trees* will be handled differently.

In the first phase, we process all the bottom trees that are not special. Consider one such bottom tree T. We construct a navigation pile in the input area of our factory and convert it into a bottom heap as explained earlier. For that, the bottom tree T is used as the output area. Each time before an element is moved from the input area to T, the element at that particular location in T is moved to the input area. After this process, the input area is again full of elements and can be used for the construction of the next bottom heap.

In the second phase, we process the special bottom trees by using Floyd's heap-construction algorithm. Since the number of the special bottom trees is a constant, this phase only requires $O(\lg n)$ work.

In the third phase, we ensure the heap order for the nodes at the upper levels by using the *sift-down* procedure of Floyd's algorithm; this is done level by level starting at level $\lfloor \lg \lg n \rfloor + 1$ upwards. For each such node we compare the values of its two children, then compare its value x with the smaller of its two children, and move this child to the parent if the value at the child is smaller than x. If a move took place, we start from the moved child and repeat until we either reach a leaf or until x is not larger than both children. We finally move x to the vacant node if any moves were performed. Other variants of the *sift-down* procedure that require less element comparisons can be used, but this will not affect our overall bounds as the work performed in the third phase is sublinear.

Note that this construction is fully in-place. Since the size of the navigation pile is only logarithmic, the bits needed can be kept in a constant number of words. In addition to these bits, the first phase only requires a constant number of other information. Since Floyd's algorithm is in-place, the second and third phases also require a constant number of words.

To optimize the number of element moves, we make two modifications to the first phase. Let us call all complete binary trees of size 7 *micro trees*. In the base case, we have to move the corresponding elements from the navigation pile to a micro tree, and vice versa. An element swap would involve three element moves. To reduce this to two element moves, we put the elements of the first micro tree aside at the beginning of the procedure. Then we move the corresponding elements from the input area to this empty space once those elements are processed, and then fill these vacated positions of the input area with the elements from the next micro tree. These actions are repeated whenever a base case is to be processed. Higher up we can use the hole technique to reduce the number of element moves by one per element. In a non-optimized form, for every step of the algorithm, we should perform a cyclic rotation of three elements: the minimum element w referred to by the current input node of the navigation pile, the excess element of its loser subtree, and the element at the current output node; this would mean four element moves. However, the pattern how elements are moved to the output area is completely predictable. Therefore, we can move the parent of the root of the first micro tree (the first output node) aside at the beginning of the procedure. Thereafter, we can move the current minimum element w from the navigation pile to this hole, move the excess element to the position of w, and create a new hole by moving the element at the next output position to the place of the excess element. Repeating these actions, a cyclic rotation of three elements would involve three element moves. After processing all the bottom trees (except the special ones), the elements that were put aside are taken back to the current holes in the input area.

Let us now analyse the performance of this procedure. The number of elements involved in all the constructions of the bottom heaps is bounded by n. It follows that, if we use our version of the **GM** algorithm in the first phase, the number of element comparisons is bounded by $1.625n + o(n)$. Compared to our earlier construction, one more element move is needed for moving each element to the input area. Including the cost of putting elements aside, the number of element moves is bounded by $2.125n + o(n)$. In the second phase the amount of work done is $O(\lg n)$. In the third phase, the number of element comparisons and moves performed by the *sift-down* routine starting from a node at height h is at most $2h$. Since there are at most $n/2^{h+1}$ nodes at height h, and as we process the nodes at height $\lfloor \lg \lg n \rfloor + 1$ upwards, the total work (element comparisons and moves) performed in the third phase is proportional to at most

$$\sum_{h=\lfloor \lg \lg n \rfloor + 1}^{\lfloor \lg n \rfloor} 2h \cdot n/2^{h+1} = O\left(n \cdot \frac{\lg \lg n}{\lg n}\right) = o(n) \,.$$

We note that in the base case the element moves are handled more efficiently than at the upper levels of the tree. By making the base case larger one could get the number of element moves even closer to $2n$. With extra space for $\sim\lg n$ elements, the bound $\sim 2n$ element moves could actually be reached.

4 Improving the Cache Behaviour

Consider the construction of a heap of size n on a computer that has a two-level memory: a slow memory containing the elements of the input array and a fast memory, call it a *cache*, where the data must be present before it can be moved further to the register file of the computer. Assume that the size of the cache is M and that the data is transferred in blocks of size B between the two memory levels; both M and B are measured in elements. We assume that the cache is ideal, so the optimal off-line algorithm is the underlying block-replacement strategy when the cache is full. The ideal cache model is standard in the analysis of cache-oblivious algorithms [7].

For our analysis, we assume that $M \gg B\lg(\max\{M, n\})$. Under this assumption, several small heaps of size $\sim\lg n$ can simultaneously be inside the fast memory. When a heap is inside the fast memory, among the blocks containing the elements of the heap, there may be at most two blocks per heap level that also contain elements from outside the heap. By the assumption, a heap of size cM together with these $2\lceil\lg(cM)\rceil$ blocks, constituting $cM + 2B\lceil\lg(cM)\rceil$ elements, can simultaneously be inside the fast memory, provided that $c < 1$ is a small enough positive constant.

Consider an algorithm **A** (either our version of the **GM** algorithm or the **MR** algorithm) that is used to construct all bottom heaps of size $\sim\lg n$. All the remaining nodes are made part of the final heap by calling Floyd's *sift-down* routine. To improve the cache behaviour of the algorithm, the enhancement proposed by Bojesen et al. [1] is to handle these nodes in reverse depth-first order instead of reverse breadth-first order. This algorithm can be coded using only a constant amount of extra memory by recalling the level where we are at, the current node, and the node visited just before the current node. When n is a power of two minus one, the procedure is pretty simple. In the iterative version given in Fig. 2, the nodes at level $\lfloor\lg\lg n\rfloor$ are visited from right to left. After constructing a binary heap below such a node using algorithm **A**, its ancestors are visited one by one until an ancestor is met that is a right child (its index is odd); for each of the visited ancestors the *sift-down* routine is called.

Here we ignored the detail that our version of algorithm **A** uses the last part of the element array as its input area, but it is easy to sift down the elements of the special bottom trees and their ancestors in a separate post-processing phase. Namely, the nodes on the right spine going down from the root of the heap to the rightmost leaf are easy to detect (their indices are powers of two minus one); so, in the other phases, visiting these nodes can be avoided.

When processing a heap of size cM during the depth-first traversal, each block is read into fast memory only once. When such a heap has been processed,

```
in-place-A::make-heap(a: array of n elements, less: comparison function)
1:  assert $n = 2^{\lceil \lg n \rceil} - 1$
2:  $h \leftarrow \lfloor \lg \lg n \rfloor$  // height of the bottom trees
3:  $j \leftarrow n/2^h$       // index of the root of the last bottom tree
4:  $i \leftarrow j/2$         // index of the parent of the node with index $j$
5:  while $j > i$
6:    A::make-heap($a$, $j$, $h$, less)
7:    $z \leftarrow j$
8:    while ($z$ bitand 1) = 0
9:      F::sift-down($a$, $z/2$, $n$, less)
10:     $z \leftarrow z/2$
11:   $j \leftarrow j - 1$
```

Fig. 2. In-place heap construction by traversing the nodes above the bottom trees in depth-first order. The root has index 1, and the leaves have height 0.

the blocks of the fast memory can be replaced arbitrarily, except that the blocks containing elements from outside this particular part are kept inside fast memory until their elements are processed. For the topmost $\sim n/(cM)$ elements, we can assume that each *sift-down* call incurs at most $\sim \lg n$ cache misses. Thus, the total number of cache misses incurred is at most $n/B + O(n \lg n/M)$. By our assumption, the first term in this formula is dominating.

5 Heap Construction in Practice

The heap-construction algorithm of Gonnet and Munro [8] is considered by many to be a theoretical achievement that has little practical significance. Out of curiosity, we wanted to investigate whether this belief is true or not; in particular, whether the improvements presented in this paper affect the state of affairs. Therefore, we implemented relaxed variants (with respect to the memory usage and the number of element moves performed) of the proposed algorithms and compared their performance to several existing algorithms. In this section we report the results of our experiments. All the implemented programs had the same interface as the C++ standard-library function `make_heap`.

The tests were performed on a 32-bit computer (model Intel® Core$^{\text{TM}}$2 CPU T5600 @ 1.83GHz) running under Ubuntu 11.10 (Linux kernel 3.0.0-13-generic) using `g++` compiler (`gcc` version 4.6.1) with optimization level `-O3`. The size of L2 cache of this computer was about 2 MB and that of the main memory 1 GB.[4]

[4] We also ran our experiments on two other 64-bit computers, one having an Intel i/7 CPU 2.67 GHz processor (Ubuntu 10.10 with Linux kernel 2.6.28-11-generic installed) and another having an AMD Phenom II X4 925 processor (4096 MB RAM and Linux Mint 11.0 installed).

In an early stage of this study, we collected programs from public software repositories and wrote a number of new competitors for heap construction. In total, we looked at over 20 heap-construction methods including: Williams' algorithm of repeated insertions [16]; Floyd's algorithm of repeated merging with top-down [6], bottom-up [14] and binary-search [2] *sift-down* policies, as well as depth-first and layered versions of it [1]; and McDiarmid and Reed's variant [12] that has the best known average-case performance.

We found that sifting down with binary search and explicitly maintaining a search path were inferior, so we excluded them from later rounds. The layered construction [1] that iteratively finds medians to build a heap bottom-up was fast on large inputs, but the number of element comparisons performed was high (larger than $2n$), so we also excluded it. Our implementation of Floyd's algorithm with the bottom-up *sift-down* policy had the same number of element comparisons as the built-in function in the C++ standard library and its running time was about the same, so we also excluded it. We looked at other engineered variants that include many implementation refinements, e.g. the code-tuned refinements discussed in [1], but they were non-effective, so we relied on Floyd's original implementation. The **GM** versions using weak heaps instead of navigation piles were consistently slower and performed more element moves, and hence were also excluded from this report.

The preliminary study thus left us with the following noteworthy competitors:

std: The implementation of the `make_heap` function that came with our `g++` compiler relying on the bottom-up *sift-down* policy [14]. Two of the underlying subroutines passed elements by value. Although this may result in unnecessary element moves, we assumed that these function calls will not involve any element moves.

F: We converted Floyd's original Algol program into C++. This program rotates the elements on the *sift-down* path cyclically, so element swaps are not used.

BKS: This variant of Floyd's heap-construction program traverses the nodes in depth-first order as discussed in [1].

in-situ GM: We implemented the algorithm of Gonnet and Munro [8] using a navigation pile as described in this paper, except that the subroutine for converting a navigation pile into a binary heap was recursive. Instead of bit-compaction techniques we used full indices at the nodes, and we did not use the hole technique for optimizing the number of element moves. Also, we had to make the height of the bottom trees $2\lfloor \lg \lg n \rfloor$ (instead of $\lfloor \lg \lg n \rfloor$) before the performance characteristics of the **GM** algorithm became visible.

in-situ MR: As in the previous program, the nodes were visited in depth-first order, but now the bottom trees were processed using the algorithm of McDiarmid and Reed [12]. All the elements on the *sift-down* path were moved cyclically first after the final position of the new element was known as proposed in [15]. Again, no bit-compaction was in use and full bytes were used instead of bits. The height of the bottom trees was also set to $2\lfloor \lg \lg n \rfloor$.

We tested the programs for random permutations of n distinct integers for different (small, medium, large, and very large) problem sizes $n = 2^{10}-1$, $2^{15}-1$,

Program / n	std	F	BKS	in-situ GM	in-situ MR
$2^{10} - 1$	22.3	14.6	17.1	21.3	26.2
$2^{15} - 1$	22.2	14.6	17.4	23.0	24.4
$2^{20} - 1$	29.3	21.9	17.8	22.9	23.6
$2^{25} - 1$	29.8	21.7	17.5	22.9	23.6

Fig. 3. Execution time per element in nanoseconds.

Program / n	std	F	BKS	in-situ GM	in-situ MR
$2^{10} - 1$	1.64	1.86	1.86	1.74	1.52
$2^{15} - 1$	1.64	1.88	1.88	1.65	1.54
$2^{20} - 1$	1.64	1.88	1.88	1.63	1.53
$2^{25} - 1$	1.65	1.88	1.88	1.63	1.53

Fig. 4. Number of element comparisons performed per element.

Program / n	std	F	BKS	in-situ GM	in-situ MR
$2^{10} - 1$	2.25	1.73	1.73	2.06	1.52
$2^{15} - 1$	2.25	1.74	1.74	2.38	1.53
$2^{20} - 1$	2.25	1.74	1.74	2.38	1.53
$2^{25} - 1$	2.25	1.74	1.74	2.38	1.52

Fig. 5. Number of element moves performed per element.

$2^{20} - 1$, and $2^{25} - 1$. The elements were of type `int`. All our programs were tuned to construct binary heaps of size $2^k - 1$. The obtained results are given in Fig. 3 (execution time), Fig. 4 (element comparisons), and Fig. 5 (element moves).

In our opinion—while not being overly tuned—our in-situ treatment for the **GM** algorithm showed acceptable practical performance. As expected, the number of element comparisons and element moves for the **in-situ GM** algorithm were beaten by the **in-situ MR** algorithm. Concerning the running time, the **in-situ GM** algorithm could beat the **in-situ MR** algorithm, but it was beaten by Floyd's algorithm **F** and the depth-first variant **BKS**.

6 Concluding Remarks

In practical terms, Floyd [6] solved the heap-construction problem in 1964. For our experiments we took his Algol program and converted it into C++ with very few modifications. For integer data, the cache-optimized version of Floyd's program described by Bojesen et al. [1] was the only program that could outperform Floyd's original program, and this happened only for large problem instances. The algorithms discussed in this paper can only outperform Floyd's program when element comparisons are expensive. In the worst case, Floyd's algorithm performs at most $2n$ element comparisons and $2n$ element moves. Furthermore, as shown in [1], by a simple modification, Floyd's algorithm can be made cache oblivious so that its cache behaviour is almost optimal.

In theoretical terms, the heap-construction problem remains fascinating. We showed how the two algorithms believed to be the best possible with respect to the number of element comparisons can be optimized with respect to the amount of space used and the number of cache misses incurred. In the worst case, our in-place variant of Gonnet and Munro's algorithm [8] requires $\sim 1.625n$ element comparisons and $\sim 2.125n$ element moves. We also showed that the same technique can be used to make McDiarmid and Reed's algorithm [12] in-place. Moreover, we proved that both of these in-place algorithms can be modified to demonstrate almost optimal cache behaviour.

The main question that is still not answered is: Can the bounds for heap construction be improved for any of the performance indicators considered?

References

1. Bojesen, J., Katajainen, J., Spork, M.: Performance engineering case study: Heap construction. ACM J. Exp. Algorithmics 5, Article 15 (2000)
2. Carlsson, S.: A variant of heapsort with almost optimal number of comparisons. Inform. Process. Lett. 24(4), 247–250 (1987)
3. Chen, J.: A framework for constructing heap-like structures in-place. In: 4th International Symposium on Algorithms and Computation. LNCS, vol. 762, pp. 118–127. Springer, Heidelberg (1993)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, Cambridge, 3nd edn. (2009)
5. Dutton, R.D.: Weak-heap sort. BIT 33(3), 372–381 (1993)
6. Floyd, R.W.: Algorithm 245: Treesort 3. Commun. ACM 7(12), 701 (1964)
7. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandra, S.: Cache-oblivious algorithms. In: 40th Annual Symposium on Foundations of Computer Science. pp. 285–297. IEEE Computer Society, Los Alamitos (1999)
8. Gonnet, G.H., Munro, J.I.: Heaps on heaps. SIAM J. Comput. 15(4), 964–971 (1986)
9. Katajainen, J., Vitale, F.: Navigation piles with applications to sorting, priority queues, and priority deques. Nordic J. Comput. 10(3), 238–262 (2003)
10. Kronrod, M.A.: Optimal ordering algorithm without operational field. Soviet Math. Dokl. 10, 744–746 (1969)
11. Li, Z., Reed, B.A.: Heap building bounds. In: 9th Workshop on Algorithms and Data Structures. LNCS, vol. 3608, pp. 14–23. Springer, Heidelberg (2005)
12. McDiarmid, C.J.H., Reed, B.A.: Building heaps fast. J. Algorithms 10(3), 352–365 (1989)
13. Vuillemin, J.: A data structure for manipulating priority queues. Commun. ACM 21(4), 309–315 (1978)
14. 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)
15. Wegener, I.: The worst case complexity of McDiarmid and Reed's variant of bottom-up heapsort is less than $n \log n + 1.1n$. Inform. and Comput. 97(1), 86–96 (1992)
16. Williams, J.W.J.: Algorithm 232: Heapsort. Commun. ACM 7(6), 347–348 (1964)