Heap Construction—50 Years Later

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

[1]*Institute for Artificial Intelligence, University Bremen, Am Fallturm 1, 28359 Bremen, Germany*
[2]*Department of Computer Engineering and Systems, Alexandria University, 21544 Alexandria, Egypt*
[3]*Department of Computer Science, University of Copenhagen, Universitetsparken 5, 2100 Copenhagen East, Denmark*
Email: jyrki@di.ku.dk

We study the problem of constructing a binary heap in an array using only a small amount of additional space. Let N denote the size of the input, M the capacity of the cache, and B the width of the cache lines of the underlying computer, all measured as numbers of elements. We show that there exists an in-place heap-construction algorithm that runs in $\Theta(N)$ worst-case time and performs at most $1.625N + o(N)$ element comparisons, $1.5N + o(N)$ element moves, $N/B + O(N/M \cdot \lg N)$ cache misses, and $1.375N + o(N)$ branch mispredictions. The same bound for the number of element comparisons was derived and conjectured to be optimal by Gonnet and Munro; however, their algorithm requires $\Theta(N)$ pointers. For a tuning parameter S, the idea is to divide the input into a top tree of size $\Theta(N/S)$ such that each of its leaves root a bottom tree of size $\Theta(S)$. When $S = \Theta(\lg N/\lg\lg N)$, we can convert the bottom trees into heaps one by one by packing the extra space needed in a few words, and subsequently use Floyd's *sift-down* procedure to adjust the heap order at the upper levels. In addition to our theoretical findings, we also compare different heap-construction alternatives in practice.

Keywords: Data Structures; Binary Heaps; Heap Construction; Algorithm Engineering; Element Comparisons; Element Moves; Cache Misses; Branch Mispredictions

1. INTRODUCTION

The *binary heap*, introduced by Williams [26], is a binary tree in which each node stores one element. This tree is *almost complete* meaning 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 *complete* if it stores $2^\ell - 1$ elements, for a positive integer ℓ. The elements are in (min-)*heap order*, if 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 is conveniently stored in an array where the elements are kept in the breadth-first order of the tree starting by the root.

In this paper we consider the problem of constructing a binary heap of N elements given in an array. Our objective is to perform the construction *in-place*, i.e. using a constant amount of additional space. To be more specific, at any point of time, at most a constant number of elements can be stored outside the input array, and a constant number of variables are used to store pointers, counters, and indices. As usual, we assume that each variable (word) is capable of storing $O(\lg N)$ bits. When measuring time, we assume that instructions supported by contemporary computers, including element comparisons and element moves, have unit cost per instruction.

1.1. Earlier work

Williams' [26] original algorithm constructs a binary heap in-place in $\Theta(N \lg N)$ worst-case time. Floyd [11] improved the worst-case running time to $\Theta(N)$. These classical results are covered by most textbooks on algorithms and data structures (see, e.g. [5, Chapter 6]). Since our construction relies on Floyd's heap-construction algorithm (**F**), its complete description is given in Figure 1. Floyd's algorithm visits the branch nodes in reverse array-index order, and at each branch node it sinks the element at that node into the two binary heaps already built in its two subtrees to establish the heap order, in a procedure known as *sift-down*. The node of the element that is being sifted down is first vacated; let x refer to this element. On the way down, the elements at the two children (if any) of the vacant node are compared, then the smaller of the two is compared to x, and is promoted to the vacant place if it is smaller than x. If a promotion took place,

```
procedure root
output index
return 0

procedure left-child
input i: index
output index
return 2i + 1

procedure right-child
input i: index
output index
return 2i + 2

procedure parent
input i: index
output index
assert i ≠ 0
return ⌊(i − 1)/2⌋
```

```
procedure sift-down
input a: element[ ] as reference, i: index, N: size
assert i < N
x ← a[i]
while left-child(i) < N
    j ← left-child(i)
    if right-child(i) < N and a[right-child(i)] < a[j]
        j ← right-child(i)
    if not (a[j] < x)
        break
    a[i] ← a[j]
    i ← j
a[i] ← x

procedure make-heap
input a: element[ ] as reference, N: size
if N < 2
    return
for i ← parent(N − 1) down to root()
    sift-down(a, i, N)
```

FIGURE 1. Floyd's heap-construction algorithm (**F**) in pseudo-code. Throughout the paper, array indexing starts from 0, whereas in [11] it is started from 1. Array $\boldsymbol{a}$ is passed by reference; other parameters are passed by value.

the process continues from the vacant child and repeats until either a leaf is reached or until x is not larger than the smaller element at the two children. Finally, x is moved to the vacant node, which may be its original place if no promotions took place inside the loop.

Without loss of generality, we can restrict ourselves to considering inputs of size $2^\ell - 1$ elements, for a positive integer ℓ. As pointed out for example in [3], the nodes that do not root a complete subtree are located on the path from the last leaf to the root. Assuming that we have an algorithm **A** that can convert a complete subtree into a heap, a general algorithm that accepts inputs of arbitrary size is obtained by

1. traversing the path from the last leaf to the root—call this path p,
2. considering the siblings of each of the nodes on p and converting the subtrees rooted at them into heaps using algorithm **A**, and
3. calling Floyd's *sift-down* for each of the nodes on p bottom-up.

For a tree of size N, the number of nodes on p is $\lfloor \lg N \rfloor + 1$. Hence, after making the complete subtrees into heaps, combining them into a single heap can be done in $O(\lg^2 N)$ worst-case time, so this combination does not affect the constant factor of the leading term in the complexity expressions for the overall heap construction.

To get an indication of the runtime performance of a heap-construction algorithm when the corresponding program is run on a computer, there is a long tradition to analyse meticulously different quantities associated with this metric [18, Section 5.2.3]. The *performance indicators* that are relevant for this study are the following:

- **element comparison**: For variables x and y that store an element, any assignment "$b \leftarrow (x < y)$" is an element comparison, where b is a Boolean variable. Naturally, either x or y or both can be array locations.
- **element move**: For variables x and y that store an element, any assignment "$x \leftarrow y$" is an element move. Again, either x or y or both can be 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.
- **cache miss**: In a two-level memory, the data is transferred in blocks between a small, fast memory (*cache*) and a large, slow memory. A miss occurs when the desired data is not in the fast memory and the processor has to wait until the block transfer from the slow memory is completed.
- **branch misprediction**: A conditional branch "**if** (b) **goto** λ", where b is a Boolean variable and λ is a label, is problematic in a pipelined processor since the next instruction may not be known—will the jump materialize or not—when its execution is started. A misprediction occurs when a branch predictor, that is supposed to guess the next instruction, makes a wrong prediction and the processor has to wait until the correct instruction is fed into the pipeline.

Observe that an element comparison is not necessarily followed by a conditional branch. Therefore, the number of branch mispredictions can be smaller than that of element comparisons. The number of branch mispredictions can also be larger if the results of index comparisons are often mispredicted.

In particular, note that we have not tried to minimize the number of instructions executed since this type of optimization depends heavily on the instruction set used by the underlying computer. In his epic series of books on the meticulous analysis of programs, Knuth [17, Section 1.3] uses his mythical MIX computer and his plan is to use a more modern MMIX computer in the future. In [15] and its follow-up papers (see, e.g. [1, 9]), the primitives of the pure-C programming language were used for the same purpose. Our experiments show that, when optimizing the performance indicators mentioned above, it is often the case that the number of instructions executed will increase. However, since cache misses and branch mispredictions are expensive on contemporary computers, the actual running time can still be smaller. Also, in applications where elements are complicated objects, element comparisons and element moves can be costly.

Even though the running time of Floyd's heap-construction algorithm is asymptotically optimal, for some performance indicators the optimal complexity bounds for heap construction are still unknown. In Table 1, we summarize the known bounds on the number of element comparisons when constructing a binary heap of size N. Here the champions are the algorithm of Gonnet and Munro [13] and the algorithm of McDiarmid and Reed [20] that perform at most $1.625N + o(N)$ element comparisons in the worst case and approximately $1.521N + o(N)$ element comparisons in the average case, respectively. Unfortunately, both algorithms require a linear amount of extra space to achieve these bounds. Both algorithms are conjectured to provide the best possible upper bounds in their respective categories, but no one has been able to prove or disprove these conjectures. The best lower bound of $1.37N$ element comparisons comes from Li and Reed [19]; the proof relies on computer calculations, the correctness of which we have not tried to verify. An information-theoretical lower bound of $1.364N$ element comparisons was proved in [13].

Already in the original description, Williams [26] tried to minimize the number of element moves performed in *sift-down* in two ways:

1. Dig a hole at the root and keep the element being sifted down in a temporary location, and then repeatedly fill this hole with an element taken from one of the children and create a new hole at the level below. This avoids the use of swaps that are more expensive.

2. Ensure that no element moves are done if the element being sifted down is the smallest of all elements under consideration (see Figure A.1 in the Appendix).

As shown in Figure 1, Floyd's algorithm also employed the hole technique. The number of element moves performed by Floyd's algorithm is at most $2N$ in the worst case and approximately $1.744N$ on the average. Since *sift-down* is called for many small trees, with the second optimization the number of element moves reduces to approximately $1.531N$ on the average. Since we could not find these average-case results from the literature as simple, both of them are proved analytically in the Appendix. The algorithm of McDiarmid and Reed [20] can be programmed to perform exactly the same number of element moves as the move-optimized version of Floyd's algorithm, i.e. it performs at most $2N$ element moves in the worst case and approximately $1.531N$ element moves on the average.

Consider a computer that has a two-level memory, where the size of the cache is M and the data is transferred in blocks of size B between the cache and main memory; both M and B are measured in elements. An algorithm is said to be *cache oblivious* if it does not know M and B. We assume that the cache is *ideal*, so that the optimal off-line algorithm is used to eject a block from the cache when it is full. The ideal cache model is standard in the analysis of cache-oblivious algorithms [12]. When constructing a binary heap of size N, an optimal cache-oblivious algorithm performs N/B cache misses, since the whole input has to be read at least once. Bojesen et al. [1] showed how Floyd's algorithm can be made cache oblivious by performing about N/B cache misses under reasonable assumptions. For the algorithms involving linear extra space, this kind of behaviour cannot be achieved due to the cache misses incurred when accessing the additional memory.

By decoupling comparisons from conditional branches, Elmasry and Katajainen [9] showed that any program can be transformed to a form that has at most one conditional branch, the outcome of which is easy to predict. However, the runtime penalty induced by this transformation could be high; the running time of the transformed program depends on the length of the original program. In their case study, they considered heap construction: It was effective to remove hard-to-predict branches (those related to element comparisons), but it did not give any, or gave only a little, performance gain to remove easy-to-predict branches (those related to index comparisons). That is, instead of aiming at the absolute minimum number of branch misprections, the golden mean turned out to be the best course of action in practice.

1.2. Contributions

In the theoretical part of this study, our main contribution is an algorithm template that can be used to make an existing algorithm **A** for heap construction

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. As a shorthand, $\sim f(N)$ represents any quantity that approaches $f(N)$ as N grows.

inventor	name	worst case	average case	extra space
Floyd [11, 21]	**F**	$2N$	$\sim 1.881N$	$\Theta(1)$ words
Gonnet & Munro [13]	**GM**	$\sim 1.625N$	$\sim 1.625N$	$\Theta(N)$ words
McDiarmid & Reed [20]	**MR**	$2N$	$\sim 1.521N$	$\Theta(N)$ bits
Li & Reed [19]	**lower bound**	$\sim 1.37N$	$\sim 1.37N$	$\Omega(1)$ words

to run in-place. First, in order to solve a subproblem of size S, we reduce the amount of extra space used by algorithm **A** to $O(\lg N + S \lg S)$ bits. Second, we use this modified algorithm to build heaps of size $\Theta(S)$ at the bottom of the input tree. For $S = \Theta(\lg N / \lg \lg N)$, we keep the needed bits in a constant number of words. Third, we combine these bottom heaps by exploiting Floyd's *sift-down* procedure at the upper levels of the tree. The key observation is that the work done by all *sift-down* calls at the upper levels is now sublinear.

We apply this algorithm template for both the algorithm of Gonnet and Munro [13] and that of McDiarmid and Reed [20]. This leads to in-place algorithms that achieve the best known bounds for the number of element comparisons performed in the worst case and in the average case, respectively. In addition to the number of element comparisons, we optimize the number of element moves. As far as we know, prior to our work, no in-place algorithm was known to achieve the bound of $1.5N + o(N)$ element moves in the worst case. By using extra space for S elements, the number of element moves can be reduced to $N + O(S + N \lg S/S)$. Without affecting the number of element comparisons and that of element moves, we show how the derived algorithms can be made cache oblivious such that they incur about N/B cache misses under reasonable assumptions. Here we rely on the work of Bojesen et al. [1]. Finally, we do branch optimization in a way that does not make the other performance indicators worse. Here we rely on the work of [9] and [10].

In the experimental part of this study, we examine when different optimizations are effective on contemporary computers. Branch optimization turns out to be effective for small problem instances, and cache optimization for large problem instances. Since the number of instructions executed is increased by a factor of four to eight as a consequence, comparison and move optimizations are first effective when these primitives require a large number of instructions.

Compared to the conference version [4], in this journal version of the paper we have

- made the algorithm template more generic so that it can be instantiated with the algorithm of Gonnet and Munro [13] and that of McDiarmid and Reed [20] without the use of any advanced data structures;
- reduced the upper bound on the number of element moves from $2.125N + o(N)$ to $1.5N + o(N)$;
- incorporated the effect of branch mispredictions; and
- made the experiments more extensive by measuring the number of primitives executed, instead of just considering the running time, the number of element comparisons, and the number of element moves for average-case inputs.

2. OPTIMIZING HEAP CONSTRUCTION FOR DIFFERENT PERFORMANCE INDICATORS

Assume we are given a heap-construction algorithm **A** that requires some extra space at run-time. In this section we describe a transformation to convert algorithm **A** to an in-place algorithm, the effectiveness of which is almost the same as the original. The basic requirement set for algorithm **A** is that it can be modified to use $O(\lg N + S \lg S)$ bits of additional space when constructing a subheap of size S. Later on, we show how the algorithm of Gonnet and Munro [13] can be modified to satisfy this requirement; the algorithm of McDiarmid and Reed [20] can process a subtree of size S with $O(\lg N + S)$ bits so the transformation applies to it as such.

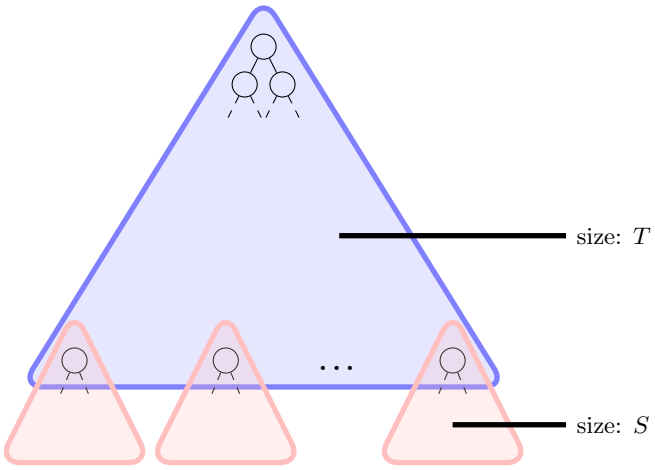
2.1. Algorithm template

Let us consider the task of building a binary heap in an array of size N, where $N = 2^\ell - 1$ for some positive integer ℓ. The basic idea is to use a stratification technique where the input, when seen as a tree, is divided into a *top tree* of size $2^k - 1$ and a collection of *bottom trees*, each leaf of the top tree rooting a bottom tree of size $S = 2^h - 1$, for some positive integers k and h. This two-layers partitioning is visualized in Figure 2.

Now we make the input array into a heap as follows:

1. Improve algorithm **A** such that it can process a subheap of size S using at most $O(\lg N + S \lg S)$ bits of additional space.
2. Apply this improved algorithm for all bottom trees. Since these subproblems are independent, the space used in one construction can be released and reused in the next.

FIGURE 2. Division of the input into two layers: top tree of size T and bottom trees of size S; $T = 2^k - 1$, $S = 2^h - 1$, and $N = 2^{k+h-1} - 1$, for some positive integers k and h.



3. Use the *sift-down* procedure of Floyd's algorithm to establish heap order at the top tree.

The crucial observation is that, by carefully choosing the parameter S, the work done by the *sift-down* at the upper levels becomes sublinear. Next, we explain why.

Provided that the modification specified in Item 1 is possible, we choose S to be a power of two minus one from the half-open interval $[\lg N/\lg\lg N \,..\, 2\lg N/\lg\lg N)$, although depending on the space efficiency of algorithm **A** other choices would also work. With this choice, the template provides an algorithm that is fully in-place, since in Item 2 the bits needed can be stored in a constant number of words and in Item 3 the *sift-down* procedure of Floyd's algorithm operates in-place.

Let us now analyse the performance of this template under the assumption that algorithm **A** can process a subheap of size S at $T_{\mathbf{A}}(S)$ cost and that algorithm **F** can handle one call of *sift-down* at $\alpha \cdot j$ cost, where α is a constant and j is the height of the starting node of *sift-down*. The number of bottom trees is at most N/S. Hence, the cost of Item 2 is bounded by $N/S \cdot T_{\mathbf{A}}(S) + O(1)$. In particular, the constant factor of the leading term of the cost expression is determined by $T_{\mathbf{A}}(S)$, i.e. it will not increase by this construction. Since there are $(N+1)/2^{j+1}$ nodes at height j of the input tree, and as we process the nodes at height $\lfloor \lg S \rfloor + 1$ upwards, the total work done in Item 3 is proportional to

$$\begin{aligned} \sum_{j=\lfloor \lg S \rfloor + 1}^{\lfloor \lg N \rfloor} \alpha \cdot j \cdot (N+1)/2^{j+1} &= O\left(N \cdot \tfrac{\lg S}{S}\right) \\ &= O\left(N \cdot \tfrac{(\lg\lg N)^2}{\lg N}\right) \\ &= o(N)\,. \end{aligned}$$

2.2. Comparison optimization

In our construction we should repeatedly convert a subtree into a heap. Assume that the elements are in an array $\boldsymbol{a}$ and let J be the index of the root of the subtree under consideration. We assume that the subtree is complete. For a positive integer ℓ, let $2^\ell - 1$ be the number of its elements.

As a minor complication, the algorithm of Gonnet and Munro [13] (**GM**) needs 2^ℓ elements to build a binary heap of size $2^\ell - 1$; one *excess* element is simply discarded. When we process a subtree of size $2^\ell - 1$, we can use any element outside it as an additional element. For the sake of simplicity, we assume first that the output is to be produced to a separate array that is of the same size as the array $\boldsymbol{a}$. Hence, when a subtree rooted at index J is processed, the output is to be produced in the corresponding place in the output array. After processing one subtree, the excess element produced by the computation must be put back in place of the additional element taken from $\boldsymbol{a}$.

The original description of Gonnet and Munro [13] relied on a heap-ordered binomial tree [23]. This is a compact representation of a tournament tree (called a selection tree in [18, Section 5.2.3]), but the two data structures are equivalent. This equivalence is discussed in length in [14, Section 3]. Several other data structures like a weak heap [6] and a navigation pile [16] could be used equally well. In the conference version of this paper [4] we used a navigation pile, but, since a tournament tree leads to a simpler implementation, we use it here.

For a visualization of the involved data structures, see Figure 3. For a set of 2^ℓ elements (players), a *tournament tree* is a complete binary tree of $2^{\ell+1} - 1$ nodes built above this set. We store the tree implicitly in another array with the same layout as a binary heap, so the child-parent relationships are calculated in the same way. In a tournament tree, the ith leaf stores a cursor to the ith element. Each branch node stores a cursor to the smaller of the elements pointed to by its two children. We call the subtree rooted at the child containing the winner of a match the *winner subtree* and the other the *loser subtree*. Obviously, the root refers to the overall champion (**Andre** in Figure 3).

Given a subtree specified by an index J and an additional element specified by an index E, the algorithm creates a binary heap and an excess element as follows:

1. Populate the leaves of the tournament with the indices of the players in the input E, J, *left-child*(J), *right-child*(J),..., either in breadth-first or depth-first order.
2. Run the tournament by performing all the matches in a bottom-up knock-out manner until the overall champion is known.
3. Convert the tournament tree into a binary heap by

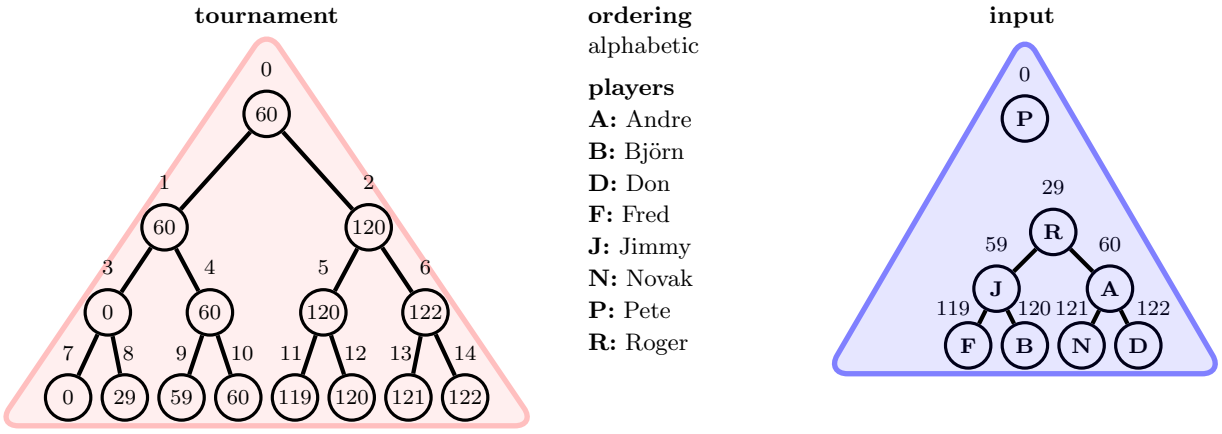


FIGURE 3. A tournament tree for 8 players in pink and the input tree in blue; the indices are displayed above the nodes; ordering is alphabetic; the root of the input tree is used as an additional element and the root of the subheap being processed has index 29.

outputting it to a separate output array plus an excess element by returning its index to the caller.

To populate a tournament tree, we set the cursors for the leaf nodes. Thereafter we run the tournament by traversing the tournament tree bottom-up level by level and setting the cursor at each branch node after comparing the elements referred to by its children. Such an initialization requires $N-1$ element comparisons. Of the three steps, the conversion step is more involved. For this purpose, we need a subroutine that can be used to update the tournament tree when the old champion retires. When the champion is replaced by another player, all the matches on the path from the corresponding leaf to the root have to be replayed. For a tournament tree that has 2^ℓ elements, such an update involves ℓ element comparisons.

The conversion from a tournament tree for 2^ℓ elements into a binary heap of size $2^\ell - 1$ plus an excess element can be done recursively; the key optimization is to stop the recursion for a subproblem of size 8, when one element comparison is enough to complete the conversion [13]:

1. If $N = 2^3$, with one additional element comparison, create a binary heap of size 7 and return by forwarding an excess element to the caller.

2. Put the element referred to by the root of the tournament tree at the root of the output heap.

3. Convert the loser subtree recursively into a binary heap and an excess element.

4. In the winner subtree, replace the smallest element with the excess element received from Item 3 and update the tree in accordance.

5. Convert the updated winner subtree into a binary heap and return by forwarding the excess element to the caller.

Let us analyse the number of element comparisons performed by this conversion procedure. To convert a tournament tree of size 2^ℓ into a binary heap, we perform two recursive calls for tournament trees of size $2^{\ell-1}$ each. In addition, we need to execute one update to refresh the cursors in the winner subtree. Let $C(2^\ell)$ be the number of element comparisons needed to convert a tournament tree of size 2^ℓ into a binary heap plus an excess element. The number of element comparisons performed for the minimum update is $\ell-1$. From this, the next recurrence relation follows:

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

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

The algorithm of Gonnet and Munro can now be used to make the subtrees hanging on the path from the last leaf to the root into heaps. For each subtree being processed, the parent of its root can be used as an additional element. In post-processing, the nodes on the path from the last leaf to the root are processed using Floyd's *sift-down* procedure. Finally, the handles to the input and the output arrays are swapped to give an illusion that the construction was in the same array.

Using this algorithm as a subroutine to process the bottom trees in our template, the next theorem follows.

Theorem 2.1. *To arrange an array of size N in heap order, the algorithm of Gonnet and Munro modified to use a tournament tree requires working space for N elements plus $o(N)$ additional space, runs in $O(N)$ time, and performs at most $1.625N + o(N)$ element comparisons and at most $N + o(N)$ element moves.*

2.3. Space optimization

In our algorithm template, when all bottom trees are processed using the **GM** algorithm, any element from the top tree can be used as the additional element. If the algorithm was used as such to create a subheap of size S, the tournament tree would require $O(S \lg N)$ bits of space, which is more than what we are planning to use. Instead of storing an array index at each node of a tournament tree, it would be more efficient to recall the index of the root of the bottom tree under consideration, and to store at each node an *offset* that specifies the distance of the intended cursor from that recalled index. Since each offset is a number between 0 and S, the whole tournament tree can be stored in $O(\lg N + S \lg S)$ bits. When $S = \Theta(\lg N / \lg \lg N)$, the tournament tree can be stored in a constant number of words.

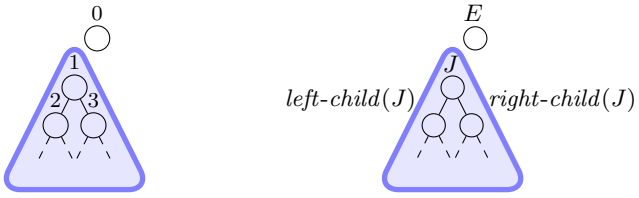


FIGURE 4. Offsets used by the algorithm (left) and the corresponding indices used in array $\boldsymbol{a}$ (right); J refers to the root of the bottom tree in question and E to the node containing an additional element.

For an illustration of the correspondence between offsets and indices, see Figure 4. The key is that the algorithm operates with the offsets, but these are converted to full-length indices on-the-fly whenever needed. The conversion routine is described in Figure 5. To assure that this conversion is a constant-time operation, we have to assume that the whole-number logarithm of a positive integer can be computed at unit cost. If wanted, we could avoid this assumption by storing the relative height of each node together with the corresponding offset.

procedure *offset-to-index*
input Δ: offset, J: index, E: index (**default** 0)
output index
if $\Delta \neq 0$
  $h \leftarrow \lfloor \lg \Delta \rfloor$
  return $2^h * J + \Delta - 1$;
return E

FIGURE 5. Converting an offset to an index; an offset is drawn from the range $\{0, 1, \ldots, S\}$ and an index from the range $\{0, 1, \ldots, N-1\}$.

Our next objective is to get rid of the output array, and arrange the elements in heap order within the input array. While permuting the elements we need to also be careful to optimize the number of element moves.

2.4. Move optimization

Let us consider the aforementioned version of **GM** in greater detail to understand how elements are moved when one of the bottom trees of size S is being processed. In particular, the goal is to do things without a separate output array. There are four arrays involved: the array $\boldsymbol{a}$ of elements, the array $\boldsymbol{t}$ of offsets storing the tournament tree, an array $\boldsymbol{\sigma}$ of offsets emulating the heap, and an array $\boldsymbol{b}$ of bits telling which elements are already in their final destinations.

Basically, the heap construction is done as in address-table sorting [18, Section 5.2]; during the process, the elements are not moved at all, but the moves are emulated by updating the offsets in the array $\boldsymbol{\sigma}$. After the computation, when the array $\boldsymbol{\sigma}$ is read in breadth-first order, it specifies a permutation of the S elements. It is well known [17, Section 1.3.3] that, given the specification of a permutation and using an additional array of S bits, the task of permuting S elements in an array in-place can be accomplished in $O(S)$ worst-case time by following the cycle structure of the permutation. In the worst case, the number of element moves performed is S plus the number of cycles in the given permutation. Since the length of each cycle can be two, $\lfloor (3/2)S \rfloor$ is an upper bound for the number of element moves.

The procedure *permute-in-place* of Figure 6 is a straightforward implementation of the permutation algorithm that processes the cycles one at a time, by walking through the array $\boldsymbol{b}$ to repeatedly advance to the next unvisited cycle. If $\boldsymbol{\sigma}[i] = j$, it means that the element at the position specified by the offset j should be moved to the position specified by the offset i. To move the elements to their final destinations with one extra element move per cycle, we use a temporary location to store an element that will be *ejected*. For each cycle, the array entry for the first element of that cycle is ejected and the other elements are moved in sequence to their destinations. At the end, the ejected element is moved to its destination. To avoid unnecessary element moves, we handle the trivial cycles of length 1, if any, in the first loop that initializes the bit array $\boldsymbol{b}$ telling which elements have been processed so far. The second loop handles all non-trivial cycles. The fact that there are at most $(S+1)/2$ non-trivial cycles ensures the upper bound of $1.5N + o(N)$ on the number of element moves.

When we apply this move-optimization procedure within our in-place implementation of the algorithm of Gonnet and Munro, the next theorem follows.

THEOREM 2.2. *To arrange an array of size N in heap order, when our algorithm template is instantiated with the move-optimized version of the algorithm of Gonnet and Munro, we get a heap-construction algorithm that operates in-place, runs in $O(N)$ worst-case time, performs at most $1.625N + o(N)$ element comparisons and at most $1.5N + o(N)$ element moves.*

```
procedure permute-in-place
input σ: offset[ ] as reference, n: size
      a: element[ ] as reference, J: index
      b: bit[ ] as reference
i ← 0
while i < n
    b[i] ← false
    if σ[i] = i
        b[i] ← true
    i ← i + 1
i ← 0
while i < n
    if not b[i]
        b[i] ← true
        H ← offset-to-index(i, J)
        temporary ← a[H]
        k ← σ[i]
        while k ≠ i
            b[k] ← true
            K ← offset-to-index(k, J)
            a[H] ← a[K]
            H ← K
            k ← σ[k]
        a[H] ← temporary
    i ← i + 1
```

FIGURE 6. Performing an in-place permutation while optimizing the number of element moves.

Assume for a moment that, for an integer parameter S, we were allowed to use $O(S)$ extra space for pointers, counters, indices, and *elements*. Then by the strategy used in the basic version of the **GM** algorithm, with bottom trees of size S each, we could organize the computation as follows:

1. Allocate space for S elements and move one bottom tree aside into it.
2. Make the next bottom tree into a heap while moving it to the evacuated area. When the elements are moved, a new empty bottom tree is created.
3. Continue this process of building heaps until all bottom trees are processed. At the end, make the bottom tree in the temporary storage into a heap using the last empty space as output area.
4. Release the temporary storage allocated.

That is, we use the hole technique for subtrees, not for single elements as in [11]. The nodes at the top tree are to be processed using Floyd's *sift-down* as before.

When processing the bottom trees, each element is moved once, except those put aside. The performance of this non-in-place variant can be summarized as follows.

Theorem 2.3. *For an integer parameter S, assume that $O(S)$ extra space is available to store pointers, counters, indices, and elements. To arrange an array of size N in heap order, when the above-mentioned variant of the algorithm of Gonnet and Munro is used, we get a heap-construction algorithm that uses $O(S)$ extra space, runs in $O(N)$ worst-case time, performs at most $1.625N+O(N \lg S/S)$ element comparisons and at most $N + O(S + N \lg S/S)$ element moves.*

2.5. Cache optimization

Consider the construction of a binary heap of size N on a computer that has a two-level memory. Assume that the capacity of the cache is M and that the data is transferred in blocks of size B between the two memory levels. We use the typical assumption that $M \gg B \lg M$. Under this assumption, a big portion of the input can be simultaneously stored inside the fast memory. When a subtree is inside the fast memory, among the blocks containing the elements of this subtree there may be at most two blocks per level that also contain elements from outside this subtree. By the assumption, a subtree of size cM together with these $2\lceil \lg(cM) \rceil$ blocks, constituting $cM + 2B\lceil \lg(cM) \rceil$ elements, can be simultaneously inside the fast memory, provided that $c < 1$ is a small enough positive constant.

Consider an algorithm **A** (either **MR** or our modification of **GM**) that is used to construct all bottom heaps of size S, for $S = \Theta(\lg N / \lg \lg N)$. When $N > 2S$, the parent of every subtree being processed must exist and can be used as the additional element needed by the construction. 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 Figure 7, the nodes at level $\lfloor \lg S \rfloor$ are visited from right to left. After making a subtree rooted at such a node into a binary heap using algorithm **A**, the ancestors of that node are visited one by one until an ancestor is met that is a right child (its index is even); for each of the visited ancestors the *sift-down* routine is called.

When processing a subtree of size cM during the depth-first traversal, each block is read into fast memory only once. When such a subtree has been processed, the blocks of the fast memory can be replaced arbitrarily, except that the blocks containing elements from outside this part are kept inside fast memory until their elements are processed. For the topmost $\lceil N/(cM) \rceil$ elements, we can assume that each *sift-down* incurs at most $\lg N$ cache misses. Thus, the total number of cache misses incurred is at most $N/B +$

```
procedure ancestor
input i: index, h: height
output index
return ⌊(i + 1)/2^h⌋ − 1

procedure make-heap
input a: element[ ] as reference, N: size
assert N ≠ 0 and N = 2^⌈lg N⌉ − 1
S ← 2^⌊lg(lg N/ lg lg N)⌋ − 1
if N ≤ 2S
    F::make-heap(a, N)
    return
h ← ⌊lg S⌋ // height of the bottom trees
j ← ancestor(N − 1, h) // index of the root of a bottom tree
i ← parent(j) // index specifying the stop condition
while j > i
    A::make-heap(a, j, N) // make a subheap
    z ← j
    while (z bitand 1) = 1
        z ← parent(z)
        F::sift-down(a, z, N)
    j ← j − 1
```

FIGURE 7. Given an algorithm **A** that can make a subheap, its cache-optimized version traverses the nodes above the bottom trees in depth-first order.

$O(N/M \cdot \lg N)$. If we further assume that $M \gg B \lg N$, the first term in this formula dominates. The discussion so far can be summarized as follows:

THEOREM 2.4. *To arrange an array of size N in heap order, on a computer that has two-level memory where the cache is of size M and the cache blocks of size B, when the cache-optimized algorithm template is instantiated with our modification of the algorithm of Gonnet and Munro, we get a heap-construction algorithm that operates in-place, runs in $O(N)$ worst-case time, performs at most $1.625N + o(N)$ element comparisons, $1.5N + o(N)$ element moves, and $N/B + O(N/M \cdot \lg N)$ cache misses.*

2.6. Branch optimization

Consider the construction of a binary heap of size N using the cache-optimized version of the **GM** algorithm. At the upper levels, the number of operations performed is $o(N)$ so the same bound holds for the number of branch mispredictions. In order to analyse the number of branch mispredictions done at the lower levels, we have to provide a more detailed description of the modified version of the **GM** algorithm used for constructing a subheap of size S. For the purpose of our analysis, we assume that the branch predictor used by the underlying hardware is static. Typically, such a predictor predicts that the body of the **then** part of an **if** statement is to be executed first. Hence, the programmer should place the probable case in the **then** part. For a loop condition, the prediction is correct except for the last iteration when stepping out of the loop.

In the modified version of the **GM** algorithm described in Figure 8 we have used the branch optimization where we interpreted the result of a comparison as an integer and then used it to add 0 or 1 to an index. The same optimization was used in [22] to optimize the behaviour of samplesort and in [9] to optimize the behaviour of Floyd's heap-construction algorithm.

The procedures *populate-tournament* and *run-tournament* initialize a tournament tree: *populate-tournament* sets the offsets at the leaf nodes in one loop and *run-tournament* sets the offsets at the branch nodes in another loop. According to our assumption, these loops incur one branch misprediction each. In particular, the conditional branch in *offset-to-index* only incurs a branch misprediction when it is called with the offset of the additional element (0). The procedure *convert-tournament* creates a permutation that specifies the order of the elements in the subheap being constructed. To unravel the branch behaviour of this procedure, we programmed it iteratively by using a stack. For a subheap of size S, the maximum size of this recursion stack is only $O(\lg^2 S)$ bits. The conditional branch in the **if** statement, testing whether we are in the general case or in the base case, will be mispredicted $(1/8)(S + 1)$ times, i.e. once for every base case. The conditional branch in the second **if** statement testing the emptiness of the stack is only mispredicted once, since after this misprediction the procedure terminates. Inside the procedure *update-tournament* there are two loops that both incur a branch misprediction when stepping out of them. This procedure is called $(1/8)(S + 1)$ times, which results in $(1/4)(S + 1)$ additional branch mispredictions. The base case must handle one hard-to-predict branch, but these branch mispredictions were avoided by using conditional moves. Thus, *convert-tournament* incurs at most $(3/8)(S + 1)$ branch mispredictions. Note that the number of branch mispredictions can be further reduced by increasing the size of the base case to be larger than 8, and using a straight-line code for it with conditional moves replacing conditional branches. For example, if the base case is of size 16, we can reach a bound of at most $(3/16)(S + 1)$ mispredictions.

Consider the procedure *permute-in-place* in Figure 6. In the first loop, the single assignment can be executed as a conditional move so this loop only incurs one branch misprediction. In the second loop, the elements are visited once. If the first check whether the element has been moved or not incurs a branch misprediction, the loop index is advanced and no other mispredictions occur. Inside the **if** statement the conditional branch of the **while** statement incurs one branch misprediction when the processing of a cycle

```
procedure populate-tournament
input t: offset[] as reference, n: size
i ← 0
for j ← parent(n) to n − 1
    t[j] ← i
    i ← i + 1

procedure run-tournament
input t: offset[] as reference, n: size
      a: element[] as reference, J: index
for j ← parent(n − 1) down to root()
    L ← offset-to-index(t[left-child(j)], J)
    R ← offset-to-index(t[right-child(j)], J)
    k ← left-child(j) + (a[R] < a[L])
    t[j] ← t[k]

procedure update-tournament
input t: offset[] as reference, i: node, n: size
      excess: offset, a: element[] as reference
      J: index
j ← i // root of the subtree considered
m ← t[i] // offset of the old champion
while left-child(j) < n
    j ← left-child(j) + (t[right-child(j)] = m)
t[j] ← excess
while j ≠ i
    j ← parent(j)
    L ← offset-to-index(t[left-child(j)], J)
    R ← offset-to-index(t[right-child(j)], J)
    k ← left-child(j) + (a[R] < a[L])
    t[j] ← t[k]

procedure general-case
input j: node, n: size
output Boolean
return left-child(left-child(left-child(j))) < parent(n)

procedure convert-tournament
input t: offset[] as reference, n: size
      σ: offset[] as reference
      a: element[] as reference, J: index
stack ← ∅
j ← root()
while true
    σ[j + 1] ← t[j]
    if general-case(j, n)
        k ← left-child(j)
        winner ← k + (t[j] ≠ t[k])
        stack.push(winner)
        loser ← k + (t[j] = t[k])
        j ← loser
    else
        excess ← handle-base-case(t, j, σ, a, J)
        if stack ≠ ∅
            j ← stack.pop()
            update-tournament(t, j, n, excess, a, J)
        else
            σ[0] ← excess
            return

procedure make-heap
input a: element[] as reference
      J: index, S: size
assert S ≠ 0 and S = 2^⌈lg S⌉ − 1
allocate space for t, σ, and b
populate-tournament(t, 2 ∗ S + 1)
run-tournament(t, 2 ∗ S + 1, a, J)
convert-tournament(t, 2 ∗ S + 1, σ, a, J)
permute-in-place(σ, S + 1, a, J, b)
free space allocated for t, σ, and b
```

FIGURE 8. Algorithm of Gonnet and Munro modified to build a subheap of size S rooted at J; the procedure *handle-base-case*—that is not shown—is a straight-line program that handles the base case of size 8 with one element comparison.

ends. The procedure *permute-in-place* then incurs at most one branch misprediction per element, i.e. $S+O(1)$ branch mispredictions in total.

Our actual implementation follows the algorithmic details described. The aim was to make the number of branch mispredictions reasonably small without increasing the number of element comparisons or the number of element moves performed. Using the fact that the procedures *populate-tournament*, *run-tournament*, *convert-tournament*, and *permute-in-place* are called at most N/S times and that S is $\Theta(\lg N/\lg\lg N)$, the performance of our algorithm can be summarized as follows:

Theorem 2.5. *To arrange an array of size N in heap order, on a computer that has a two-level memory where the cache is of size M and the cache blocks of size B, when the cache-optimized template is instantiated with our branch optimization for the algorithm of Gonnet and Munro, we get a heap-construction algorithm that operates in-place, runs in $O(N)$ worst-case time, performs at most $1.625N + o(N)$ element comparisons, $1.5N+o(N)$ element moves, $N/B+O(N/M\cdot\lg N)$ cache misses, and $1.375N + o(N)$ branch mispredictions.*

3. EXPERIMENTS

The heap-construction algorithm of Gonnet and Munro [13] 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. In addition to the in-place methods discussed, we implemented relaxed variants of the proposed algorithms that work *in-situ*, i.e. using $O(\lg N)$ variables to store pointers, counters, and indices. We compare the performance of our algorithms to several existing alternatives, and report the results of our experiments in this section.

3.1. Methods considered

In an early stage of this study, we collected programs from public repositories and wrote a number of competitors for heap construction. In total, we looked at over 20 heap-construction methods including: Williams' [26] algorithm of repeated insertions; Floyd's [11] algorithm of repeated merging with top-down, bottom-up [25] and binary-search [2] *sift-down* policies, as well as depth-first and layered versions of it [1]; and McDiarmid and Reed's [20] variant 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* 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 checked other engineered variants with many 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 a weak heap [6] and a navigation pile [16] were faster for integer data, but performed more element moves than the versions using a tournament tree, and hence these were also excluded from this study.

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

- **stl**: The *make-heap* function that came with our `g++` compiler. On closer inspection, it was found to rely on the bottom-up *sift-down* policy [18, Exercise 5.2.3–18] (see also [25]). Two of the underlying subroutines passed elements by value, so this resulted in some unnecessary element moves.

- **F**: Floyd's [11] Algol program converted into C++. In *sift-down*, this program employed the hole technique, so element swaps were not used.

- **GM**: Our implementation of the algorithm of Gonnet and Munro [13] using a tournament tree. For an input of size N, the program used a tournament tree that had space for $(4/3)N$ indices and a temporary output area that had space for N elements.

- **space-efficient GM**: The space-efficient variation on **GM** described in this paper. Both cache and branch optimizations were applied. This program could be configured to operate in-place or in-situ. Both versions used $O(1)$ extra space for elements. The in-place variant used a packed array that could store a sequence of integers of equal length compactly. The in-situ variant stored the offsets in an integer array. The program accepted a tuning parameter γ (32 by default) and set the size of the bottom trees to the closest power of two larger than, or equal to, $\gamma \lg N / \lg \lg N$.

- **MR**: The bottom trees were processed using the algorithm of McDiarmid and Reed [20]. As in the previous program, both cache and branch optimizations were applied. Also this program could be configured to operate in-place (with a packed array of bit pairs) or in-situ (with an array of bytes). The program accepted a tuning parameter μ (16 by default) and set the size of the bottom trees to the closest power of two larger than, or equal to, $\mu \lg N$. As proposed in [24], all the elements on the *sift-down* path were moved cyclically first after the final position of the new element was known. When converting a bottom tree into a heap, the indices of the element array from the interval $[0\,..\,N)$ and those of the packed array from the interval $[0\,..\,S)$ were updated in tandem, which doubled the instruction count for index operations.

We considered the following optimization options for Floyd's program:

- $\mathbf{opt_1}$ [9]: We made sure that *sift-down* was always called with an odd N. This way, inside the inner loop, one easy-to-predict branch could be removed.

- $\mathbf{opt_2}$ [9]: We interpreted the result of an element comparison as an integer and used this value in normal index arithmetic. This way, inside the inner loop, the hard-to-predict branch in "**if** (condition) $j \leftarrow j + 1$" could be replaced with an assignment "$j \leftarrow j +$ (condition)".

- $\mathbf{opt_3}$ [9]: As in Figure A.1 in the Appendix, we did not make any element moves when the element at the root stayed in its original location.

- $\mathbf{opt_4}$ [1]: We visited the nodes in reverse depth-first order instead of reverse breadth-first order.

- $\mathbf{opt_5}$ [9]: We made the construction in a single loop by fusioning the two loops in *make-heap* and *sift-down*. Inside this loop, conditional moves

were used so that the number of element moves would not increase to $5N$, but two of the element moves were left behind conditional branches. The outcome of these two conditional branches was predicted reasonably well so it was not worth avoiding these branches.

For subscript i, we use $\mathbf{F_i}$ to denote the version of **F** that used $\mathbf{opt}_i$. When several subscripts are in use, all these optimizations were applied at the same time.

3.2. Test environment

We performed the experiments in three different computers, all of which run Linux and offered `g++` compiler. Since the results were similar in all of these computers, we only report the results obtained in one of them. During experimentation, all unnecessary system services were shut down. The hardware and software specifications of the test computer were as follows.

- **processor**: Intel® Core™ i5-2520M CPU @ 2.50GHz × 4
- **word size**: 64 bits
- **L_1 instruction cache**: 32 KB, 64 B per line, 8-way associative
- **L_1 data cache**: 32 KB, 64 B per line, 8-way associative
- **L_2 cache**: 256 KB, 64 B per line, 8-way associative
- **L_3 cache**: 3.1 MB, 64 B per line, 12-way associative
- **main memory**: 3.8 GB, 8 KB per page
- **operating system**: Ubuntu 14.04 LTS
- **Linux kernel**: 3.13.0-79-generic
- **compiler**: `g++` version 4.8.4
- **compiler options**: `-O3 -std=c++11 -x c++ -Wall -DNDEBUG`
- **profiler**: `valgrind` version 3.10.1
- **profiler options**: `--tool=cachegrind --cache-sim=yes --branch-sim=yes`

A simple driver was written to measure the number of element comparisons, the number of element moves, and the running time. The profiler was used to measure the number of instructions, the number of L_3-level cache misses, and the number of branch mispredictions. The numbers provided by the profiler were based on simulations.

When describing the algorithms, we specialized them for inputs of size $2^\ell - 1$ or 2^ℓ, for an integer ℓ, but all our programs were generalized to handle inputs of arbitrary sizes. The input for the driver could be selected to be

- an increasing sequence of integers $\langle 0, 1, \ldots, N-1\rangle$,
- a decreasing sequence of integers $\langle N-1, N-2, \ldots, 0\rangle$, or
- a random permutation of integers $\{0, 1, \ldots, N-1\}$.

Using these input sequences, we wanted to understand how much the performance could vary for different performance indicators. This way we aimed at finding the worst-case behaviour for the performance indicators studied. For program **P**, we use $\overline{\mathbf{P}}$ to denote its performance for the randomly permuted input. Every experiment was repeated $r = \lfloor R/N \rfloor$ times, for $R = 2^{26}$ or $R = 715\,827\,882$, each time with a different input. At the end, the measurement results were scaled by dividing them by $r \times N$, i.e. by reporting the cost per element.

All the implemented programs had the same interface as the C++ standard-library function *make-heap*. The source code of the heap-construction programs discussed is made available alongside this article [7]. Although the programs accepted any element, sequence, and comparator type, all tests used 4-byte `int` elements, a C array to store the elements, and `std::less` in element comparisons. If other types of input data or comparison functions are used in the application in hand, we encourage that the reader performs his or her own experiments before deciding which program to use.

3.3. Experimental results

In Figure 9, we display the average-case running times for the most significant competitors considered. For cache-sensitive programs, with a careful scrutiny, four plateaus are visible in the curves, each corresponding to one of the memory levels in the test computer (L_1 cache, L_2 cache, L_3 cache, and main memory). For other programs, there are no visible memory effects; variations in the running times are due to the value of N. When N is close to a power of two, a program can be a bit faster than for other values. Based on these observations, instead of using plots, we hereinafter report the test results for only four problem sizes (small, medium, large, and huge): $N = 2^{10} - 1$, $2^{15} - 1$, $2^{20} - 1$, and $2^{25} - 1$.

For different performance indicators, the obtained results are reported in tabular form in Table 2 (running time), Table 3 (element comparisons), Table 4 (element moves), Table 5 (instructions), Table 6 (cache misses), and Table 7 (branch mispredictions). The running time, the number of instructions, cache misses, and branch misprediction were measured for the random input. For all programs, except **stl**, the up sequence maximized the number of element comparisons and element moves; for **stl**, the maximum was reached for the down sequence.

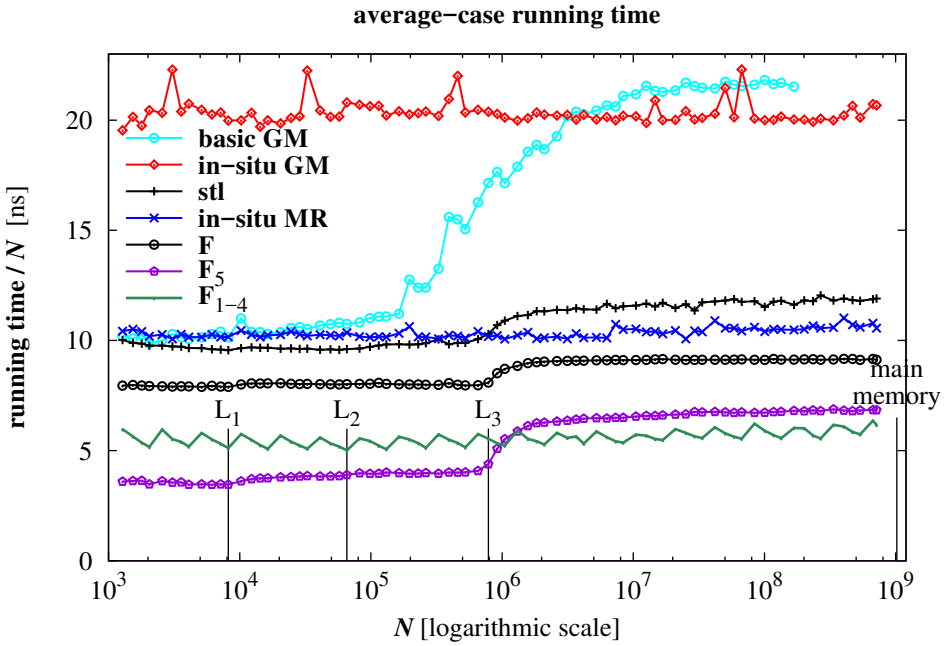


FIGURE 9. Running time [ns], divided by N, for some of the heap-construction programs considered; **input:** random permutation. The sizes of different memory levels are drawn as vertical lines. Since the virtual-memory support was switched off, none of the programs could handle much larger input instances than those shown.

TABLE 2. Running time [ns], divided by N, for ten heap-construction programs; **input:** random permutation.

N	stl	F	$\mathbf{F_1}$	$\mathbf{F_{12}}$	$\mathbf{F_{1\text{-}3}}$	$\mathbf{F_{1\text{-}4}}$	$\mathbf{F_5}$	GM	in-situ GM	in-situ MR
$2^{10}-1$	9.36	7.96	7.86	4.64	4.60	5.10	3.57	10.35	16.82	9.70
$2^{15}-1$	9.21	8.12	7.83	4.95	4.81	5.01	3.84	10.76	19.27	9.56
$2^{20}-1$	9.91	8.76	8.51	6.03	5.82	4.98	5.47	17.33	19.38	9.51
$2^{25}-1$	10.41	9.09	8.97	7.03	6.90	4.98	6.46	21.75	19.73	9.51

3.4. Discussion of the experimental results

Of the two starting points, **F** and **GM**, Floyd's program was simpler and faster. Moreover, due to the extensive consumption of memory, the running time of **GM** deteriorated when the size of the input became about one fourth of the size of the $\mathbf{L_3}$ cache. Both **in-situ GM** and **in-situ MR** were slower than **F**, but their behaviour with respect to the number of element comparisons and the number of element moves matched the theoretical bounds proved. In fact, on the average, the number of element moves performed by **in-situ GM** was close to N since, when permuting the elements into the final destinations, the cycles considered were typically slightly longer than two (which induced the term $1.5N$ as the upper bound on the number of element moves).

By varying the tuning parameters for **in-situ GM** and **in-situ MR**, it turned out that the results became stable and the theoretical bounds for the number of element comparisons and the number of element moves were reached when the parameters are set to the default values, or higher. Therefore, we used the default values in all the experiments reported.

From the different versions of **F**, the effects of branch optimization are clearly visible. However, when branch optimization was applied alone, the programs were still sensitive to caching effects. This is seen from the results for $\mathbf{F_1}$, $\mathbf{F_{12}}$, $\mathbf{F_{1\text{-}3}}$, $\mathbf{F_5}$; branch optimization was more effective for small values of N than for large values of N. Branch optimization and cache optimization applied in $\mathbf{F_{1\text{-}4}}$, however, improved the running time for large values of N, but an addition of a single branch made it a bit slower than the branch-optimized version $\mathbf{F_5}$ for small values of N.

For the branch-optimized version $\mathbf{F_5}$, the upper bound $2N$ for the number of element comparisons and element moves was reached for all types of inputs since the code makes very few choices. Hence, it is not adaptive for random inputs.

For integer data, the expected number of instructions executed by $\mathbf{F_{1\text{-}3}}$ was as low as 14.37. To optimize the number of element comparisons and the number of element moves, the price that we had to pay was a significant increase in the number of instructions executed; for **in-situ GM** it was about a factor of eight higher.

TABLE 3. Number of element comparisons performed, divided by N, for some heap-construction programs; **input:** decreasing sequence for **stl**; increasing sequence for **F**, **GM**, and **in-situ GM**; and random permutation for $\overline{\textbf{stl}}$, $\overline{\textbf{F}}$, $\overline{\textbf{F}_5}$, and $\overline{\textbf{in-situ MR}}$.

N	stl	$\overline{\text{stl}}$	F	$\overline{\text{F}}$	$\overline{\text{F}_5}$	GM	in-situ GM	$\overline{\text{in-situ MR}}$
$2^{10}-1$	1.98	1.64	1.98	1.86	1.98	1.65	1.72	1.52
$2^{15}-1$	2	1.65	2	1.88	2	1.63	1.65	1.54
$2^{20}-1$	2	1.66	2	1.88	2	1.63	1.64	1.53
$2^{25}-1$	2	1.65	2	1.88	2	1.63	1.64	1.53

TABLE 4. Number of element moves performed, divided by N, for some heap-construction programs; **input:** decreasing sequence for **stl**; increasing sequence for **F**, **GM**, and **in-situ GM**; and random permutation for $\overline{\textbf{stl}}$, $\overline{\textbf{F}}$, $\overline{\textbf{F}_3}$, $\overline{\textbf{F}_5}$, and $\overline{\textbf{in-situ MR}}$.

N	stl	$\overline{\text{stl}}$	F	$\overline{\text{F}}$	$\overline{\text{F}_3}$	$\overline{\text{F}_5}$	GM	in-situ GM	$\overline{\text{in-situ MR}}$
$2^{10}-1$	3.98	3.24	1.99	1.73	1.52	1.99	2.06	1.28	1.52
$2^{15}-1$	4	3.26	2	1.74	1.53	2	2	1.06	1.53
$2^{20}-1$	4	3.26	2	1.74	1.53	2	2	1.04	1.53
$2^{25}-1$	4	3.26	2	1.74	1.53	2	2	1.04	1.53

TABLE 5. Number of instructions executed, divided by N, for ten heap-construction programs; **input:** random permutation.

N	$\overline{\text{stl}}$	$\overline{\text{F}}$	$\overline{\text{F}_1}$	$\overline{\text{F}_{12}}$	$\overline{\text{F}_{1\text{-}3}}$	$\overline{\text{F}_{1\text{-}4}}$	$\overline{\text{F}_5}$	$\overline{\text{GM}}$	$\overline{\text{in-situ GM}}$	$\overline{\text{in-situ MR}}$
$2^{15}-1$ $2^{20}-1$ $2^{25}-1$	26.18	22.25	20.37	16.59	14.37	15.71	23	50.88	114 ± 1	43.44

TABLE 6. Number of block transfers performed | cache misses incurred, both divided by N/B, for some heap-construction programs; **input:** random permutation.

N	$\overline{\text{stl}}/\overline{\text{F}}$	$\overline{\text{F}_{1\text{-}4}}$	$\overline{\text{GM}}$	$\overline{\text{in-situ GM}}$	$\overline{\text{in-situ MR}}$
$2^{10}-1$	1 | 1	1 | 1	1 | 1	1 | 1	1 | 1
$2^{15}-1$	5.30 | 1	1.03 | 1	18.00 | 1.01	1.03 | 1	1.05 | 1
$2^{20}-1$	5.55 | 4.56	1.04 | 1	20.86 | 13.51	1.05 | 1	1.04 | 1
$2^{25}-1$	5.87 | 5.84	1.04 | 0.99	21.08 | 20.40	1.05 | 0.99	1.04 | 0.99

TABLE 7. Number of branches executed | branch mispredictions incurred, both divided by N, for some heap-construction programs; **input:** random permutation.

N	$\overline{\text{stl}}$	$\overline{\text{F}}$	$\overline{\text{F}_{1\text{-}3}}$	$\overline{\text{F}_{1\text{-}4}}$	$\overline{\text{F}_5}$	$\overline{\text{GM}}$	$\overline{\text{in-situ GM}}$	$\overline{\text{in-situ MR}}$
$2^{10}-1$	4.88 | 0.93	4.53 | 0.83	2.17 | 0.27	2.42 | 0.47	2.97 | 0.04	4.90 | 0.31	10.55 | 0.46	6.55 | 0.74
$2^{15}-1$	4.90 | 0.85	4.56 | 0.80	2.18 | 0.24	2.43 | 0.47	3 | 0.03	4.64 | 0.42	12.22 | 0.46	6.50 | 0.74
$2^{20}-1$	4.91 | 0.85	4.57 | 0.80	2.18 | 0.24	2.43 | 0.47	3 | 0.03	4.63 | 0.34	12.34 | 0.46	6.48 | 0.67
$2^{25}-1$	4.91 | 0.85	4.56 | 0.80	2.18 | 0.24	2.43 | 0.47	3 | 0.03	4.63 | 0.33	12.34 | 0.46	6.48 | 0.67

TABLE 8. Running time [ns], divided by N, for **in-situ** and **in-place** variants; **input:** random permutation.

N	$\overline{\text{in-situ GM}}$	$\overline{\text{in-place GM}}$	$\overline{\text{in-situ MR}}$	$\overline{\text{in-place MR}}$
$2^{10}-1$	16.82	38.90	9.70	14.93
$2^{15}-1$	19.27	48.86	9.56	14.79
$2^{20}-1$	19.38	49.62	9.51	14.80
$2^{25}-1$	19.73	49.25	9.51	14.78

TABLE 9. Number of instructions executed, divided by N, for **in-situ** and **in-place** variants; **input:** random permutation.

N	$\overline{\text{in-situ GM}}$	$\overline{\text{in-place GM}}$	$\overline{\text{in-situ MR}}$	$\overline{\text{in-place MR}}$
$2^{15}-1$ $2^{20}-1$ $2^{25}-1$	114 ± 1	295 ± 3	43.44	79.53

When measuring the cache misses, the experiments show that the cache behaviour of **stl**, **F** (and its branch-optimized variants), and **GM** was not optimal. For $\mathbf{F_{1\text{-}4}}$, **in-situ GM**, and **in-situ MR**, the cache performance was almost optimal, confirming the theoretical results.

The results show that the number of branches executed by **in-situ GM** and **in-situ MR** was higher than that for the other programs. This is an indication that the programs are more complicated. In a sense, for **in-situ MR**, the theory is a bit misleading since element comparisons are replaced with bit comparisons or index comparisons. So, in fact, the number of branches executed is increased, not reduced.

3.5. Cost of being in-place

Intuitively, an algorithm is considered to operate in-place if it does not need to make a copy of its data. In practical terms, any algorithm that has this property and uses sublinear additional space would be acceptable. This was the practical motivation for allowing $O(\lg N)$ extra space in our in-situ solutions. It is also well known that the requirement of operating fully in-place can make the programs unacceptably slow. How much in our case?

To understand how much overhead a fully in-place solution has in the case of the **GM** and **MR** algorithms, we implemented a separate packed-array structure that could store S small integers of at most $b \leq w$ bits each using a total of at most $O((Sb)/w)$ words, where w is the word size of the underlying computer. That is, under the assumption that $w \geq \lg N$, for $[S = O(\lg N/\lg\lg N)$ and $b = \lg S]$ or $[S = O(\lg N)$ and $b = 2]$, the amount of extra space used is only $O(1)$ words. Except for the packed array, only a few lines of code had to be changed to get from the in-situ solution (**in-situ GM** or **in-situ MR**) to an in-place solution (**in-place GM** or **in-place MR**).

For the in-place variants, the comparison, move, and cache behaviour were still at the same good level. We have to also admit that, for all variants of **GM** and **MR**, the branch behaviour is bad compared to the variants of **F**, so there is no reason to report that. This leaves us with two quantities: the running time and the number of instructions executed for a random input. We report these two quantities in Table 8 and Table 9.

The message is clear: One should take the in-place algorithms as theoretical achievements; we do not expect them to be competitive in practice.

3.6. Further tuning

Even though the in-place variants are not practically usable, there are still several options how the in-situ variants can be improved. Profiling showed that, for **in-situ GM**, the main reason for its inefficiency was the conversion of offsets to indices: it used about 30 % of its running time for *offset-to-index* calculations. On the other hand, the basic version avoided *offset-to-index* calculations by populating the tournament tree with indices, not with offsets. For small values of N, **GM** was a factor of two faster than **in-situ GM**.

This observation motivated us to implement yet another variant of **GM**:

- **tuned GM**: This version used $O(\lg N)$ extra space for pointers, counters, and indices, but it took the further step to use $O(\lg N)$ extra space for elements as well. The size of the bottom trees was fixed to a power of two minus one that was just larger than, or equal to, $\nu \lg N$. By default, $\nu = 12$. By Theorem 2.3, with this amount of extra space for elements, the number of element moves performed in the worst case reduced to $N+O(N \lg\lg N/\lg N)$.

The test results for **in-situ GM** and **tuned GM** are compared—for average-case inputs—in Table 10 (running time), Table 11 (instruction count), Table 12 (element moves), and Table 13 (branches and branch mispredictions). For integer data, the tuned version was never more than a factor of two to three slower than the best versions of Floyd's program. For the other performance indicators, except the number of branch mispredictions, it matched the best known bounds up to lower order terms, and even for branch mispredictions the results were not bad.

TABLE 10. Running time [ns], divided by N, for **tuned GM**; **input:** random permutation.

N	in-situ GM	tuned GM
$2^{10}-1$	16.82	9.47
$2^{15}-1$	19.27	9.62
$2^{20}-1$	19.38	9.47
$2^{25}-1$	19.73	9.62

TABLE 11. Number of instructions executed, divided by N, for **tuned GM**; **input:** random permutation.

N	in-situ GM	tuned GM
$2^{15}-1$		
$2^{20}-1$	114 ± 1	48 ± 0.25
$2^{25}-1$		

TABLE 12. Number of element moves, divided by N, for **tuned GM**; **input:** random permutation.

N	in-situ GM	tuned GM
$2^{10}-1$	1.22	1.46
$2^{15}-1$	1.04	1.05
$2^{20}-1$	1.03	1.02
$2^{25}-1$	1.03	1.01

TABLE 13. Number of branches executed | branch mispredictions incurred, both divided by N, for **tuned GM**; **input:** random permutation.

N	in-situ GM	tuned GM
$2^{10}-1$	12.22 | 0.46	4.01 | 0.24
$2^{15}-1$	12.22 | 0.46	3.48 | 0.19
$2^{20}-1$	12.34 | 0.46	3.45 | 0.19
$2^{25}-1$	12.34 | 0.46	3.42 | 0.19

4. CONCLUSIONS

For most practical purposes, Floyd [11] solved the problem of heap construction in 1964. We could readily use his Algol program and convert it into C++ with very few modifications. Bojesen et al. [1] showed how Floyd's algorithm can be made cache oblivious so that, under reasonable assumptions, its cache behaviour is almost optimal. Elmasry and Katajainen [9] showed how Floyd's algorithm can be modified to avoid branch mispredictions. For integer data, the cache-optimized version could outperform Floyd's original for large problem instances, and the branch-optimized version could outperform it for small problem instances. The other algorithms discussed in this paper can be used to implement programs that only outperform these champions when element comparisons and/or element moves are expensive.

Theoretically speaking, the heap-construction problem remains fascinating. We showed how the algorithms believed to be the best possible with respect to the number of element comparisons could be optimized with respect to the amount of space used and the number of element moves performed. In the worst case, our in-place variant of Gonnet and Munro's algorithm requires at most $1.625N + o(N)$ element comparisons and at most $1.5N + o(N)$ element moves. We also showed that the same technique can be used to run McDiarmid and Reed's algorithm in-place. Moreover, we proved that both algorithms can be modified to demonstrate almost optimal cache behaviour.

The algorithm template used to obtain these results is quite general and the same technique can be applied for other types of heaps as well. If a heap-building procedure exists that requires additional space for its operation, it can be converted to operate in-place or in a space-efficient manner by processing the bottom subtrees and the top subtree as described in this paper. Recently, we have shown [8] that a similar approach can be used successfully for the construction of strong heaps that are otherwise as binary heaps, but, for each node, the left child is known to store the smaller of the elements at the two children.

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

ACKNOWLEDGEMENTS

We thank Jingsen Chen (Luleå University of Technology) for inspiring us to write the conference version of this paper [4].

APPENDIX A. AVERAGE NUMBER OF ELEMENT MOVES FOR FLOYD'S ALGORITHM

In this appendix, we analyse the number of element moves performed by Floyd's heap-construction algorithm and its move-optimized version. We recall the move-optimized version in Figure A.1 in the form proposed in [9]. In the analysis, we closely follow the guidelines given in [21]. Actually, the analysis given in [21, Theorem 2] turns out to be for the optimized version, not for the original version.

```
procedure sift-down
input a: element[ ] as reference, i: index, N: size
assert left-child(i) < N and N mod 2 = 1
j ← left-child(i) + (a[right-child(i)] < a[left-child(i)])
if not (a[j] < a[i])
    return
x ← a[i]
a[i] ← a[j]
i ← j
while left-child(i) < N
    j ← left-child(i) + (a[right-child(i)] < a[left-child(i)])
    if not (a[j] < x)
        break
    a[i] ← a[j]
    i ← j
a[i] ← x
```

FIGURE A.1. *sift-down* in the move-optimized and branch-optimized version of Floyd's heap-construction algorithm; otherwise, the construction is done as in Figure 1 except that, when $N \bmod 2 = 0$, the last element is inserted into the heap separately.

Theorem A.1. *On the average, Floyd's algorithm (Figure 1) performs approximately* $1.744N$ *element moves when constructing a heap of size* $N = 2^\ell - 1$*, for an integer* $\ell \geq 1$*.*

Proof. Let $S(i)$ denote the number of element moves performed by the *sift-down* procedure of Floyd's algorithm for a heap of size $n = 2^i - 1$ and let $P(i)$ denote the number of promotions done when sinking a random element down a heap of size $2^i - 1$, i.e. $P(i)$ is the number of element moves done inside the **while** loop. The key observation is that, with probability $1/(2^i - 1)$, the new element is the smallest of the $2^i - 1$ elements under consideration and the traversal down can be stopped. Hence, the following two recurrences are used to describe the number of element moves done:

$$P(i) = \begin{cases} 1 & \text{if } i = 1\,, \\ 1 + \frac{2^i-2}{2^i-1} \cdot P(i-1) & \text{otherwise}\,. \end{cases} \quad \text{(A.1)}$$

$$S(i) = \begin{cases} 0 & \text{if } i = 1\,, \\ 2 + \frac{2^i-2}{2^i-1} \cdot P(i-1) & \text{otherwise}\,. \end{cases} \quad \text{(A.2)}$$

By repeated substitutions of EquationA.1, we get for $i \geq 2$

$$\begin{aligned} \tfrac{2^i-2}{2^i-1} \cdot P(i-1) &= \tfrac{n-1}{n} + \tfrac{n-3}{n} + \cdots + \tfrac{n-(2^{i-1}-1)}{n} \\ &= i - 1 - \textstyle\sum_{i=1}^{i-1}(2^i-1)/n \\ &= i - 1 - \tfrac{2^i-2}{n} + \tfrac{i-1}{n} \\ &= i \cdot (n+1)/n - 2 \\ &= i \cdot 2^i/(2^i-1) - 2. \end{aligned}$$

In accordance, by using Equation A.2, we get for $i \geq 2$

$$S(i) = i \cdot 2^i/(2^i-1).$$

For $i = 1, 2, \ldots$, there are $2^\ell/2^i$ subheaps of size $2^i - 1$. Thus, the total number of element moves performed by *make-heap* is

$$\begin{aligned} & \textstyle\sum_{i=1}^{\ell} \frac{2^\ell}{2^i} \cdot S(i) \\ = \;& (N+1) \textstyle\sum_{i=2}^{\ell} \frac{i}{2^i-1} \\ < \;& (N+1) \textstyle\sum_{i=2}^{\infty} \frac{i}{2^i-1} \\ \approx \;& 1.744N. \end{aligned}$$

□

THEOREM A.2. *On the average, the move-optimized version of Floyd's algorithm (Figure A.1) performs approximately* $1.531N$ *element moves when constructing a heap of size* $N = 2^\ell - 1$*, for an integer* $\ell \geq 1$*.*

Proof. For the move-optimized version, instead of Equation A.1, we have the following recurrence

$$S(i) = \begin{cases} 0 & \text{if i} = 1\,, \\ \frac{2^i-2}{2^i-1} \cdot (2 + P(i-1)) & \text{otherwise}\,. \end{cases} \quad \text{(A.3)}$$

Using the formula for $\frac{2^i-2}{2^i-1} \cdot P(i-1)$ from Theorem A.1 and substituting in Equation A.3, we get for $i \geq 2$

$$S(i) = i \cdot 2^i/(2^i-1) - 2/(2^i-1).$$

Hence, the number of element moves performed by the move-optimized version is

$$\begin{aligned} & \textstyle\sum_{i=1}^{\ell} \frac{2^\ell}{2^i} \cdot S(i) \\ = \;& (N+1) \textstyle\sum_{i=2}^{\ell} (\frac{i}{2^i-1} - \frac{2}{2^i \cdot (2^i-1)}) \\ < \;& (N+1)(1.744 - \textstyle\sum_{i=2}^{5} \frac{2}{2^i \cdot (2^i-1)}) \\ < \;& (N+1)(1.744 - 0.1666 - 0.0357 - 0.0083 - 0.002) \\ \approx \;& 1.531N. \end{aligned}$$

□

REFERENCES

[1] J. Bojesen, J. Katajainen, and M. Spork, Performance engineering case study: Heap construction, *ACM J. Exp. Algorithmics* **5** (2000), 15.1–15.44.

[2] S. Carlsson, A variant of Heapsort with almost optimal number of comparisons, *Inform. Process. Lett.* **24**, 4 (1987), 247–250.

[3] J. Chen, A framework for constructing heap-like structures in-place, *Proceedings of the 4th International Symposium on Algorithms and Computation*, *Lecture Notes in Computer Science* **762**, Springer, Berlin/ Heidelberg (1993), 118–127.

[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, Berlin/Heidelberg (2012), 259–270.

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

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

[7] S. Edelkamp, A. Elmasry, and J. Katajainen, Heap-construction programs, CPH STL Report **2016-1**, Department of Computer Science, University of Copenhagen, Copenhagen (2016).

[8] S. Edelkamp, A. Elmasry, and J. Katajainen, Optimizing binary heaps, *Theory Comput. Syst.* (to appear).

[9] A. Elmasry and J. Katajainen, Lean programs, branch mispredictions, and sorting, *Proceedings of the 6th International Conference on Fun with Algorithms*, *Lecture Notes in Computer Science* **7288**, Springer, Berlin/Heidelberg (2012), 119–130.

[10] A. Elmasry, J. Katajainen, and M. Stenmark, Branch mispredictions don't affect mergesort, *Proceedings of the 11th International Symposium on Experimental Algorithms*, *Lecture Notes in Computer Science* **7276**, Springer, Berlin/Heidelberg (2012), 160–171.

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

[12] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandra, Cache-oblivious algorithms, *Proceedings of the 54th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society, Los Alamitos (1999), 285–297.

[13] G. H. Gonnet and J. I. Munro, Heaps on heaps, *SIAM J. Comput.* **15**, 4 (1986), 964–971.

[14] B. Haeupler, S. Sen, and R. E. Tarjan, Rank-pairing heaps, *SIAM J. Comput.* **40**, 6 (2011), 1463–1485.

[15] J. Katajainen and J. L. Träff, A meticulous analysis of mergesort programs, *Proceedings of the 3rd Italian Conference on Algorithms and Complexity*, *Lecture Notes in Computer Science* **1203**, Springer, Berlin/ Heidelberg (1997), 217–228.

[16] J. Katajainen and F. Vitale, Navigation piles with applications to sorting, priority queues, and priority deques, *Nordic J. Comput.* **10**, 3 (2003), 238–262.

[17] D. E. Knuth, *Fundamental Algorithms*, *The Art of Computer Programming* **1**, 3rd Edition, Addison Wesley Longman, Reading (1997).

[18] D. E. Knuth, *Sorting and Searching*, *The Art of Computer Programming* **3**, 2nd Edition, Addison Wesley Longman, Reading (1998).

[19] Z. Li and B. A. Reed, Heap building bounds, *Proceedings of the 9th International Workshop on Algorithms and Data Structures*, *Lecture Notes in Computer Science* **3608**, Springer, Berlin/Heidelberg (2005), 14–23.

[20] C. J. H. McDiarmid and B. A. Reed, Building heaps fast, *J. Algorithms* **10**, 3 (1989), 352–365.

[21] T. Pasanen, Elementary average case analysis of Floyd's algorithms to construct heaps, TUCS Technical Report **No. 64**, Turku Centre for Computer Science, Turku (1996).

[22] P. Sanders and S. Winkel, Super scalar sample sort, *Proceedings of the 12th Annual European Symposium on Algorithms*, *Lecture Notes in Computer Science* **3221**, Springer, Berlin/Heidelberg (2004), 784–796.

[23] J. Vuillemin, A data structure for manipulating priority queues, *Commun. ACM* **21**, 4 (1978), 309–315.

[24] I. Wegener, 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 (1992), 86–96.

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

[26] J. W. J. Williams, Algorithm 232: Heapsort, *Commun. ACM* **7**, 6 (1964), 347–348.