Near-Optimal Working-Set Heaps and Dijkstra on Pointer Machines

Ivor van der Hoog
IT University of Copenhagen, Denmark

John Iacono
Université libre de Bruxelles, Belgium

Eva Rotenberg
IT University of Copenhagen, Denmark

Daniel Rutschmann
Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria

Abstract

A heap is a dynamic data structure that stores a set of labeled values under the following operations: `pop` returns the minimum value of the heap, `Push`(x_i) pushes a new value x_i onto the heap, and `DecreaseKey`$(i,\ v)$ decreases the value x_i to v. A working-set heap is a heap that supports the $x_i \leftarrow$ `pop`() operation in $O(\log \Gamma(x_i))$ time where $\Gamma(x_i)$ is the size of the *working set*: the number of elements that were pushed onto the heap while x_i was in the heap. The goal of working set heap design is to maintain the working set property while minimizing the overhead of the `Push` and `DecreaseKey` operations. On a word RAM, there exist working set heaps that support `Push` and `DecreaseKey` in amortized constant time. In this paper, we show via a simple construction that pointer machines, one of the most general and least-assuming computational models, support working set heaps that support `Push` in amortized constant time and `DecreaseKey` in inverse-Ackermann time. A by-product of this analysis is that Dijkstra's shortest path algorithm can be near-universally optimal on a pointer machine – incurring only an additive $O(m\,\alpha(m))$ overhead compared to the optimal running time for distance ordering, where m denotes the number of edges in the graph.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Data structures, graph algorithms, amortized analysis

Digital Object Identifier 10.4230/LIPIcs.ESA.2026.45

Related Version *Full Version*: `https://arxiv.org/abs/2604.24134`

Funding *Ivor van der Hoog, Eva Rotenberg, and Daniel Rutschmann* thank the VILLUM Foundation grant (VIL37507) "Efficient Recomputations for Changeful Problems" for supporting this work. *Daniel Rutschmann* is supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (No. 101019564). *John Iacono* is supported by the Fonds de la Recherche Scientifique – FNRS.

Acknowledgements This work started at Dagstuhl 25191: Adaptive and Scalable Data Structures.

1 Introduction

Heap data structures are one of the fundamental data structures in theoretical computer science, and there exist many heap implementations in the literature. Most of the heaps in the literature are based on heap-ordered trees, i.e. tree structures where the data stored in a node has a value that is not smaller than the value of the data stored in its parent.

At a high level, a heap is a data structure used to repeatedly extract the minimum of a dynamic set. Formally, the `Push` operation adds a new data node onto the heap, and `pop` extracts the minimum of the current heap. Heaps typically also support various other

operations: `MakeHeap` creates a new heap and `Peek` yields the minimum value in the heap without removing it. Another common and useful operation is `DecreaseKey` which selects a current data node in the heap and replaces the data with a smaller value. The combination of `DecreaseKey` and `Pop` then allows for arbitrary deletions from the heap.

We are interested in implementing heaps on a pointer machine – one of the most general models of computation, and one of the least-assuming generalizations of the RAM model used to analyze data structures. The classical pointer machine solution is a heap-ordered tree, which achieves logarithmic time for all aforementioned heap operations. Early examples are the implicit binary heaps of Williams [37], the leftist heaps of Crane [6] and Knuth [26], and the binomial heaps of Vuillemin [36]. Fredman and Tarjan introduced Fibonacci heaps [11] which achieved $O(1)$ amortized time for all the above operations except for `Pop`, which requires $O(\log n)$ amortized time where n is the number of data nodes in the heap. Fibonacci heaps had an immediate and famous application in Dijkstra's algorithm for single-source shortest paths [7]. If Dijkstra's algorithm is run on a graph with n nodes and m edges then, using a Fibonacci heap, the algorithm takes $\Theta(m + n \log n)$ time, which is worst-case optimal.

Since Fibonacci heaps, many other heaps with fast `DecreaseKey` have been introduced (see [4] for a survey): The Pairing heap [14] is a self-adjusting structure whose asymptotic amortized `DecreaseKey` running time is unknown. Only an upper bound of $O(2^{\sqrt{\log \log n}})$ [28] and a lower bound of $\Omega(\log \log n)$ [13] are known. The strict Fibonacci heap has worst-case rather than amortized running times [3]. Chan's quake heaps were invented to be a simpler alternative to Fibonacci heaps with $O(1)$ amortized time `DecreaseKey` [4]. Rank-pairing heaps [17] are another alternative to Fibonacci heaps with $O(1)$ amortized time `DecreaseKey` but their structure is closer to the pairing heap. Violation heaps [9], Hollow Heaps [18], and Fibonacci Heaps revisited [25] are other variants with $O(1)$ amortized time `DecreaseKey`. Slim and smooth heaps [19] have $O(\log \log n)$ `DecreaseKey`, but achieve constant indegree.

Adaptive bounds. An adaptive heap supports the `Pop` operation in $o(\log n)$ amortized time under specific circumstances, while the `Push` operation takes $O(1)$ amortized time. Most existing heaps with adaptive bounds work without supporting the `DecreaseKey` operation: Iacono [20] shows that, on a pairing heap, $x \leftarrow$`Pop`() takes $O(1 + \log \Gamma(x))$ time where $\Gamma(x)$ is the number of data nodes that were pushed while x was on the heap. Elmasry [8] designs a heap on which $x \leftarrow$`Pop`() takes $O(1 + \log \Gamma'(x))$ time where $\Gamma'(x)$ is the number of data nodes that were pushed while x was on the heap and remain in the heap until x is popped. Elmasry, Farzan and Iacono [10] improve this to $O(\min(\log \Gamma'(x), \log \ell))$ time where $\Gamma'(x)$ is defined in the same way and ℓ is the number of data nodes that were pushed *before* x was pushed and that remain in the heap until x is popped. Rutschmann [29] designs a heap with the unified bound: $x \leftarrow$`Pop`() takes $O(\min_y(1 + \log(a(x, y)) + \log(b(x, y))))$ time where y can be any data node that was already popped, $a(x, y)$ is the number of push operations between `Push`(x) and `Push`(y), and $b(x, y)$ is the number of `Pop` operations between $x \leftarrow$`Pop`() and $y \leftarrow$`Pop`(). Kozma and Saranurak [27] introduce the Smooth heap, which is conjectured to be instance optimal – if true, this implies that the smooth heap has all the above bounds.

We are aware of two adaptive heaps that support `DecreaseKey`. Both implement the *working set property*: `Push` and `Peek` take (amortized) constant time, and `Pop` takes $O(1 + \log \Gamma(x))$ time, where $\Gamma(x)$ is the number of data nodes that were pushed while x was on the heap. Haeupler, Hladík, Rozhoň, Tarjan, and Tětek [16] present a word RAM implementation of a working set heap that moreover supports `DecreaseKey` in amortized constant time. They show that using this heap, Dijkstra's algorithm is universally optimal on the word RAM (see Corollary 4). Van der Hoog, Rotenberg, and Rutschmann [35] simplify their result.

RAM Model. The two adaptive heaps that support `DecreaseKey` are implemented not on a pointer machine, but on a word RAM. The word RAM model is very powerful, and in this model it is frequently possible to improve upon the best pointer-machine model results, typically by double-log factors. Fundamentally, from any given node in a pointer machine, one can only access a constant number of nodes. In contrast, in the RAM model one can access an arbitrary element of an array in constant time. Furthermore, if on a RAM the data is assumed to have a certain representation, such as word-sized integers rather than simply comparison-based objects, the world of what is commonly known as *bit-tricks* becomes available for data structure implementations. These were introduced by Fredman and Willard with Fusion trees [12] and extended to the heap setting with Atomic Heaps [15]; further improvements were made by Thorup in relation to the circuit complexity [33].

Pointer machine data structures. In this paper, we find it imperative to establish what the best pointer machine model solution is for any problem. This is for a number of reasons:

- First, we note that pointer machines are the more general model of computation. RAM models often allow for faster running times than pointer machine solutions, as RAM tricks often squeeze out loglog or similar factors from the best pointer machine result. In the interest of generality and purity of the model of computation, we therefore want to see what overhead is obtained by relying on a pointer machine rather than a Word RAM.
- Secondly, improvements in the pointer machine model often lead to better RAM model algorithms as efficient solutions in this model require fundamental algorithmic efficiency.
- Thirdly, many of the techniques used to speed up RAM algorithms are invoked to shave off log-factors in manners that do not always translate to practice. In practice, such bit-trick techniques require careful packing of multiple items into machine words to exploit word-level parallelism which in itself introduces practical overhead. Pointer machine solutions, such as ours, are in comparison typically simple and implementable.
- Fourthly, the pointer machine model lends itself more naturally to lower bounds. For example, the union-find problem has a well-known inverse-Ackermann lower bound that is derived in the pointer machine model. Achieving strong upper bounds in the pointer machine model can indicate a potential for lower bounds that cannot exist on a RAM.

This makes the pointer machine model the natural model to analyze data structures.

Our results. Our goal is to find a pointer machine heap with the working set property that supports the `DecreaseKey` operation with small overhead. We aim to provide a general and simple, all-purpose pointer machine solution. Previous works with adaptive heap bounds either (1) disallow `DecreaseKey`, (2) support `DecreaseKey` in logarithmic or doubly logarithmic time, or (3) use the sledgehammer of RAM tricks to solve small-size problems in constant time and eliminate iterated-log terms from otherwise inefficient structures.

We present a pointer-model heap with constant-time `Push`, working-set `Pop`, and inverse-Ackermann time `DecreaseKey`. It is based on a simple, recursive, and elegant construction using Fibonacci heaps or other optimal pointer machine model non-adaptive heaps as building blocks. While our implementation is not necessarily simpler than the RAM implementations [16, 34], we consider our solution considerably simpler compared to other adaptive pointer machine heaps [8, 10, 20] and in addition, we support the `DecreaseKey` operation.

The inverse-Ackermann is a stubborn bottleneck in our approach, and all our attempts at removing this overhead have failed. This leaves open the question of whether the inverse-Ackermann is *necessary* in the pointer machine model. For example, is a reduction possible to other problems with inverse-Ackermann running times such as union-find?

2 Preliminaries

We implement working-set heaps on a pointer machine. Knuth [26] gave one of the earliest formal definitions of a pointer machine model, which on a high level is a directed graph where nodes can store data values and nodes have pointers to other nodes. Tarjan [32] refined the definition, formalizing the operations supported by a pointer machine. In this paper, we will use a more general version of the pointer machine by Tarjan [32]. In particular, our model of computation only requires a subset of the operations supported by Tarjan as we have no need to modify our data (only our data structure). For algorithmic convenience, we assume that each node in the pointer machine can store $O(1)$ pointers. In contrast, Tarjan [32] uses only one pointer, but that pointer can point to specific memory cells called *pointer fields* that record any additional pointers. These assumptions are asymptotically equivalent.

Formally defining a pointer machine data structure. A pointer machine consists of *nodes* and *pointers*. There are two types of nodes. A *data node* stores any type of data (integers, logical values, strings, real numbers, whatsoever). A *pointer node* stores a number of constantly many bits. Each node can have up to constantly many pointers, where a pointer is a directed edge from one node to another. The *dynamic input* is a set of data nodes, where the current input size n denotes the number of data nodes. An *update* adds, removes, or changes the data of a data node. A *dynamic data structure* maintains an arbitrarily-large set of pointer nodes and the pointer nodes of the data nodes. One pointer node is called the *root* and we maintain the invariant that any node can be reached by a directed path from the root. The *size* of the data structure is the total number of nodes it has.

Any update or query to the data structure runs a *program* with a *working memory*: an array of constant size where each cell points to a node. It executes a sequence of *instructions*:

- A *comparison* takes as input two nodes in working memory of the same type (pointer or data nodes) and adds to working memory a pointer node with the Boolean outcome.
- An *atomic operation* takes as input three nodes in working memory. It applies the operator to the latter two and replaces the first node with its output.
- A *copy* updates the working memory by replacing one node by the copy of another.
- A *traversal* updates the working memory by replacing one node by a node it points to.
- A *halt* instruction ends the computation and the output is the current working memory.

By this definition, input values can be compared but not manipulated. This makes the model very general. The *running time* is the total number of instructions until a *halt* instruction.

Worst-case running time bounds imply amortized bounds.

Defining heaps. Having defined our operations, we abstractly define a *heap* data structure.

▶ **Definition 1.** *A* heap *data structure stores a set of data nodes with data values in some linearly ordered universe. Let n denote the current number of data nodes. A heap supports the following operations:*

- `MakeHeap`*()* *creates a new data structure and returns a pointer r to its root.*
- `Push`*(r, x_j)* *Given a pointer r to a heap's root, and a data node x_j, add x_j to the heap.*
- `Peek`*(r)* *Given a pointer to the root r of a heap, output the minimum value on that heap.*
- `Pop`*(r)* *Given a pointer to the root r of a heap, output and remove its minimum value.*
- `DecreaseKey`*(r, x_j, v)* *Given a pointer r to the root of a heap, a data node x_j in that heap, and new value v smaller than the current value of x_j, replace the value of x_j with v.*
- *(optional)* `Meld`*(r, r')* *Given two pointers r and r' to two distinct heap data structures returns a pointer to one heap containing all elements.*

The interface of the `DecreaseKey` operation has a subtle complication: It requires not only a pointer to the data node, but also a pointer to the root of the data structure. Matching this interface is difficult when we maintain several heaps subject to the Meld operation [23]. Indeed, if our memory contains heaps $H_1, \ldots, H_k$ and we wish to perform `DecreaseKey` on some element $x_j \in H_i$ then the `DecreaseKey` interface demands a pointer to the root of H_i (in addition to the pointer to x_j). Explicitly storing a pointer from each data node x_j to the root of its heap H_i would solve this issue. However these pointers cannot be maintained efficiently under the Meld operation (as linearly many data nodes may need a new pointer).

Fibonacci heaps. Fredman and Tarjan [11] present a pointer machine implementation of a Fibonacci heap, which supports the `MakeHeap`, `Peek`, `Push`, and `DecreaseKey` operations in amortized constant time and the `Pop` operation in amortized logarithmic time. In addition, their heap supports the Meld operation in worst-case constant time. However, as noted earlier, to support Meld, the interface of the `DecreaseKey` operation requires an additional pointer. Brodal, Lagogiannis and Tarjan [3] achieve the same bounds with worst-case time instead of amortized time, with the same caveat on combining `Meld` and `DecreaseKey` operations.

Working set heaps. Iacono and Fredman [22] introduce several notions of *working set size*, some of which are asymptotically equivalent. Each notion assigns to every heap element $x_j \in H$ a value $\Gamma(x_j)$, called its *working set size*. The objective is to design heap data structures that support `Pop` in time $O(1 + \log \Gamma(x_j))$, where x_j is the element returned. In this paper, we focus on a specific definition from [22] that is used in recent work [16, 35]. To distinguish it from the variants in [22], we instead refer to this quantity as the element's *age*:

▶ **Definition 2.** *Let H be a heap of size n. For any data node $x_j \in H$ we define its* age $\Gamma(x_j)$ *as the number of elements that were pushed onto the heap after x_j (including x_j itself). A* working set *heap supports the `Push` and `Peek` in $O(1)$ time, `Decreasekey` in $O(f(n))$ time, and the `Pop` operation in $O(1 + \log \Gamma(x_j))$ time where x_j is the returned element.*

The goal of designing an efficient working set heap is to provide an implementation where the "overhead" $f(n)$ is as low as possible. Iacono [20] shows that a double pairing heap can support the `Push`, `Peek` and `Meld` operations in amortized constant time and the `Pop` operation in amortized $O(1 + \log \Gamma(x_j))$ time. However, this analysis does not support `DecreaseKey`. Haeupler, Hladík, Rozhoň, Tarjan, and Tětek [16] present a word RAM implementation of a working set heap with no overhead (i.e. $f(n) = 1$) where all running times are amortized. They show that using this heap, Dijkstra's algorithm is universally optimal on the word RAM (we forward reference Corollary 4 for the formal statement).

Van der Hoog, Rotenberg, and Rutschmann [35] simplify their analysis and show a simpler word RAM implementation that achieves amortized bounds. Both solutions crucially rely on bit-tricks. We argue later that replacing the bit-tricks with pointer machine operations naively incurs an overhead of $f(n) = \log \log \log n$. We show that it is possible to achieve $f(n) = \alpha(n)$ where $\alpha(n)$ is the inverse-Ackermann function. Note that we do not support `Meld` and that it is even unclear how the ages of elements would be defined under `Meld`.

▶ **Theorem 3.** *A heap can be implemented on a pointer machine that supports the following operations, where n denotes the heap size:*

- `MakeHeap` *in worst-case $O(1)$ time,*
- `DecreaseKey` *in amortized $O(\alpha(n))$ (inverse-Ackermann) time,*
- `Push` *in amortized $O(1)$ time,*
- `Peek` *in worst-case $O(1)$ time, and*
- `Pop` *in worst-case $O(1 + \log \Gamma(x_j))$ time.*

All worst-case operations build no amortized potential. I.e., invoking them does not affect the amortized analysis of the other operations.

▶ **Corollary 4** (Direct consequence of [16, 35]). *Fix any (directed) graph G with n vertices and m edges, where one vertex s is denoted as the source. Denote for a fixed single-source shortest path (SSSP) algorithm $\mathcal{A}$ by $U(\mathcal{A}, G)$ the maximum running time required by $\mathcal{A}$ to sort all vertices by their distance from s (the maximum is taken over all positive edge-weightings of G). Let $\mathcal{U}$ denote the minimum over all SSSP algorithms $\mathcal{A}$ of $U(\mathcal{A}, G)$. Then Dijkstra's algorithm equipped with the heap from Theorem 3 achieves a running time of $O(\alpha(n) \cdot m + \mathcal{U})$.*

3 A working set heap

The basis of a working set heap, as shown in [21], is an exponential bucketing scheme based on the *age* of data nodes in the heap. Consider all elements $x_j \in H$ ordered by their age. This bucketing scheme creates an ordered doubly linked list of pointer nodes (called *buckets*) H_i. One may choose whether these grow exponentially or doubly exponentially in size. Each bucket H_i stores a Fibonacci heap, such that the buckets together partition H. Moreover, for all i, the data in H_i has a lower age than the data in H_{i+1}. The goal is to ensure that for any $x_j \in H_i$, the logarithm of its age ($\Theta(\log \Gamma(x_j))$) lies in $\Omega(\log |H_i|)$. Crucially, for each bucket H_i the `Pop` operation (returning x_j) then runs in $O(\log \Gamma(x_j))$ time as Fibonacci heaps support `Pop` in time $O(\log N)$ where N denotes the size of the Fibonacci heap.

Maintaining the buckets and pointer machines. On a pointer machine these buckets $H_1, \ldots, H_k$ are pointer nodes that form a doubly linked list where for each bucket H_i, its third pointer points to the root of the Fibonacci heap $T(H_i)$. The high-level data structure can easily be maintained in amortized constant time via a strategy that greedily merges adjacent buckets [16, 35]. However, two key challenges remain:

1. Firstly, to support `DecreaseKey`(r, x_j, v) on an element $x_j \in H$, we need to invoke `DecreaseKey`(r', x_j, v) on the Fibonacci heap $T(H_i)$ where $x_j \in H_i$ and where r' is a pointer to the root of $T(H_i)$. However, we already remarked that it is nontrivial to obtain r'. We want to meld Fibonacci heaps in constant time, so we cannot explicitly maintain a pointer from each element x_j to the root of the melded heap. Rather, we need to spend time during a `DecreaseKey` operation to determine the current heap that x_j is in.
2. Secondly, to support `Peek` and `Pop` on the heap H, we need to maintain the index i such that H_i contains the minimum element.

These issues are essentially bypassed on a word RAM. For the first problem, the buckets can be stored in an array. Consider an element x_j where $i := \lfloor \log \Gamma(x_j) \rfloor$. These bucketing schemes ensure that x_j is in some H_k with $k = i \pm O(1)$. Let x_j be the j'th element that was pushed onto H. If we at all times maintain a pointer to the last element x_k that was pushed onto the heap, then we can compute for any element x_j its age via the formula $k - j + 1$. As a result, one can use the age of an element x_j to compute the index of bucket that contains x_j, and obtain the root of the corresponding Fibonacci heap. For the second problem, one maintains the minimum of at most $\log n$ values. By representing these values as a bitstring (where a 0 at index i denotes that the value in H_i is smaller than in H_r for all $r > i$) and using constant-time bitstring manipulation one can maintain the index i such that H_i stores the minimum of H at no additional overhead.

A naive pointer machine implementation. On a pointer machine, existing solutions to these two problems yield a working set heap with `DecreaseKey` overhead $f(n) = \log\log\log n$: The union-find data structure [31] is a pointer machine data structure that maintains, for a partition of data nodes into sets, a representative node from each set. It supports the `MakeSet` operation in amortized $O(1)$ time, which creates a new singleton set; the `Link` operation in amortized $O(1)$ time, which takes as input two representatives of different sets, replaces these sets by their union, and returns a new representative; and the `Find` operation in amortized $O(\alpha(n))$ time, which takes a data node and returns the representative of its set. Here, n is the total number of `MakeSet` operations performed. By applying `Link` whenever two Fibonacci trees get melded, and storing pointers between the Fibonacci tree and the representative, one can use `Find` to return for each element x_j the root of its Fibonacci tree in $O(\alpha(n))$ time. For the second problem, note that when using doubly exponentially growing buckets there are $O(\log\log n)$ such minima. Maintaining these in a separate binary heap incurs an additive $O(\log\log\log n)$ overhead onto `Push`, `DecreaseKey`, and `Pop`. If we use a suitably biased binary heap, we only incur this $O(\log\log\log n)$ overhead on `DecreaseKey`. The challenge in this paper is to maintain a working set heap that supports the `Push` and `Peek` operation in amortized $O(1)$ time and `DecreaseKey` in amortized $O(\alpha(n))$ time.

3.1 Our pointer machine implementation

In Section 4, we design an auxiliary data structure that supports the following:

▶ **Theorem 5.** *(Index structure) We can maintain a collection of m data nodes $A = \langle A[1], A[2], \ldots, A[m]\rangle$ subject to the following operations:*

- $p_m \leftarrow$ *`Push`*$(A[m])$*: increases the size of the structure by adding a new last element $A[m]$ and returns a pointer p_m to be used by the other operations to change $A[m]$. This runs in worst-case $O(m)$ time.*
- $p \leftarrow$ *`Peek`*(A)*: returns a pointer p to the minimum value in A, in worst-case $O(1)$ time.*
- *`DecreaseKey`*(p_i, v)*: decreases the value of $A[i]$ to v in worst-case $O(\alpha(i))$ time.*
- *`ChangeKey`*(p_i, v)*: changes the value of $A[i]$ to v in worst-case $O(i)$ time.*

We now describe our main data structure, using the index data structure as a black box. Our data structure will maintain a partition of the current stored data into sets $H_1, H_2, \ldots H_m$, which we refer to as *buckets* with the following three invariants:

- Each bucket H_i either corresponds to a strict Fibonacci heap F_i containing H_i (as strict Fibonacci heaps exist in the pointer machine model), or, is *vacant*. The *vacancy invariant* requires that only buckets H_i with odd i may be vacant. We use the word vacant to mean there is no structure F_i, to distinguish from *empty*, which refers to a Fibonacci heap F_i that is present but stores no data.
- The *size invariant* requires the maximum size of H_i to be $2^{\lfloor i/2\rfloor}$.
- The age of all items in bucket i is at least the sum of the maximum sizes of the non-vacant smaller buckets. Formally, the *age invariant* states that:

$$\text{If } x \in H_i \text{ then } \Gamma(x) \geq \sum_{\substack{j<i \\ j \text{ non-vacant}}} 2^{\lfloor j/2\rfloor}$$

Since all H_i for even i are non-vacant, for $x \in H_i$, this yields $\Gamma(x) \geq 2^{i/2-2}$.

The above properties are designed so that (a) swapping H_{2i} and H_{2i+1} where one of them is vacant and (b) melding nonvacant H_{2i} and H_{2i+1} and placing the result into the vacant H_{2i+2} (leaving H_{2i} and H_{2i+1} vacant) both maintain the size and age invariants (but not the vacancy invariant). The minimum value in each H_i is stored as $A[i]$ with pointer p_i in the index structure (Theorem 5). If H_i is vacant or empty then we store ∞ instead.

Additionally, each data node is stored in a union-find structure which maintains the current partition into the buckets. We use r_i to denote the union-find data structure's representative of H_i. I.e., given a pointer ρ to a data node x in H_i, $\texttt{Find}(\rho)$ returns r_i.

To be clear: our data structure stores three things for each bucket H_i, the Fibonacci heap pointer F_i, the pointer p_i into the index structure, and the representative from the union-find structure r_i. Pointers to and between all of these, for a given i, are all stored in a linked list ordered by i. The general idea is to use Bentley-Saxe [2] inspired insertions on the exponentially-sized H_i's, made faster through the constant-time `Meld` operation of Fibonacci heaps. This is augmented by the union-find structure that allows us to find out efficiently which H_i a given data node is in, as well as the indexing structure whose primary role is to quickly determine which H_i currently stores the minimum element.

We now describe how to implement each heap operation. We define here one recurrent subroutine $\texttt{Update}(p_i)$ which updates an out-of-date value in the index structure by obtaining the minimum value of F_i via $m_i = \texttt{Peek}(F_i)$ and then invokes $\texttt{ChangeKey}(p_i, m_i)$ in the indexing data structure to update $A[i]$. This takes worst-case time $O(i)$.

`DecreaseKey:` To perform $\texttt{DecreaseKey}(\rho, v)$, given the pointer ρ to the data node x to decrease, we first invoke $r_i = \texttt{Find}(\rho)$ on our union-find data structure in amortized $O(\alpha(m))$ time to obtain the representative r_i such that $x \in H_i$. With r_i we also have F_i and p_i. Since we now know which Fibonacci heap x belongs to, we can invoke $\texttt{DecreaseKey}(F_i, \rho, v)$ on the strict Fibonacci heap F_i. Next, let $y = \texttt{Peek}(F_i)$ be the possibly new minimum of F_i. Finally, we need to update the index structure, this is done by invoking $\texttt{DecreaseKey}(p_i, y)$ on the index structure in $O(\alpha(i)) \subseteq O(\alpha(m))$ worst-case time. For now, we assume that $m \in O(n)$, so that `DecreaseKey` takes $O(\alpha(n))$ amortized time.

`Push:` We first ensure that H_0 is vacant (temporarily breaking the vacancy invariant) by the following recursive process: To ensure H_{2i} is vacant we do one of three things. If H_{2i} is not vacant but H_{2i+1} is vacant, we move H_{2i} to H_{2i+1}; if both H_{2i} and H_{2i+1} are vacant, we recursively ensure H_{2i+2} is vacant and meld H_{2i} and H_{2i+1} and place the result in the now-vacant H_{2i+2}.

Given that H_0 is now empty, we place x into a new Fibonacci heap F_0, create a new set with representative r_0 in the union-find structure, and call $\texttt{Update}(p_0)$ to update the index structure.

Moving H_{2i} to H_{2i+1} is just a matter of swapping pointers: F_{2i} and F_{2i+1}; r_{2i} and r_{2i+1} and the running of $\texttt{Update}(p_{2i})$ and $\texttt{Update}(p_{2i+1})$. Melding H_{2i} with H_{2i+1} to yield H_{2i+2} involves melding the heaps via $F_{2i+2} = \texttt{Meld}(F_{2i}, F_{2i+1})$ as well as melding the union-find structure via $r_{2i+2} = \texttt{Link}(r_{2i}, r_{2i+1})$. Finally the index structure is updated with calls to `Update` on p_{2i}, p_{2i+1} and p_{2i+2}. In the special case where we are using a H_i for the first time, it needs to be added to the index structure via `Push`.

The worst-case running time is $O(\ell^2)$ where ℓ is the largest value where H_ℓ was changed in this operation. This can be bounded by $O(1)$ amortized time by giving each non-vacant bucket H_i a potential of $\Theta(i^2)$ and by applying the textbook binary counter analysis in [5, Chapter 17]. Note that this does not affect other operations as no other operations change whether or not a bucket is vacant.

`Peek:` Call $p_i = \texttt{Peek}()$ from the index structure and return the value of $A[i]$, which can be obtained from p_i.

`Pop:` Call $p_i = \texttt{Peek}()$ on the index structure. With p_i, we obtain F_i. We then invoke $x = \texttt{Pop}(F_i)$ on that heap in worst-case $O(\log |H_i|) = O(i)$ time. Note that after the pop, F_i may be empty, but we do not change it to vacant. Finally, invoke $\texttt{Update}(p_i, m)$ to update index structure in $O(i)$ time, before returning x. Since for each element x in H_i, $\Gamma(x) \geq 2^{i/2-2}$, the worst-case running time of $O(i)$ can be expressed as $O(1 + \log \Gamma(x))$.

Let us now address the $m \in O(n)$ assumption required by `DecreaseKey`: At the start of `DecreaseKey`, we check whether $m > n$. If so, then we rebuild the whole data structure from scratch. By the textbook analysis for dynamic arrays [5], this incurs $O(1)$ amortized time.

Thus, once we have proven Theorem 5, we obtain:

▶ **Theorem 3.** *A heap can be implemented on a pointer machine that supports the following operations, where n denotes the heap size:*
- *`MakeHeap` in worst-case $O(1)$ time,*
- *`DecreaseKey` in amortized $O(\alpha(n))$ (inverse-Ackermann) time,*
- *`Push` in amortized $O(1)$ time,*
- *`Peek` in worst-case $O(1)$ time, and*
- *`Pop` in worst-case $O(1 + \log \Gamma(x_j))$ time.*

All worst-case operations build no amortized potential. I.e., invoking them does not affect the amortized analysis of the other operations.

4 A Fast Index Structure

We prove that we can maintain the indexing data structure (Thm. 5) via the following lemma:

▶ **Lemma 6.** *For every integer m, there is a rooted tree with m leaves that can be computed in $O(m)$ time and that has the following properties:*
1. *The i-th leaf has a depth of $O(\alpha(i))$*
2. *Every node on the root-to-leaf path of the i-th leaf has $O(\sqrt{i})$ children.*

Showing that Lemma 6 implies Theorem 5. We first show that Lemma 6 implies Theorem 5 on a pointer machine. The data structure uses the tree T from Lemma 6 as follows: The i-th leaf stores the value $A[i]$. Every node stores a pointer to its parent, and to its first child. We store the children of every node in a doubly linked list. Every node stores a pointer to the leaf that contains the minimum value in its subtree. And finally, we store a global pointer to the root of the tree. As a result, all nodes have $O(1)$ outgoing pointers. Given the edges of the tree, all of this can be computed in $O(m)$ time. We use this tree to implement the operations specified in Theorem 5 as follows:
- `Peek`(A): since the root node stores a pointer to the minimum values in the whole tree, we can return this value in $O(1)$ time.
- `DecreaseKey`$(A[i], v)$: We update the value in the i-th leaf. Then, we walk up the root-to-leaf path and update the subtree minimum pointer if $A[i]$ has become the new subtree minimum. Since the i-th leaf has depth $O(\alpha(i))$, this takes $O(\alpha(i))$ time in total.
- `ChangeKey`$(A[i], v)$: We update the value in the i-th leaf. Then, we walk up the root-to-leaf path and recompute the subtree minimum pointers from scratch by looking at the subtree minimum pointers of all children. On the root-to-leaf path, there are $O(\alpha(i))$ nodes with $O(\sqrt{i})$ children each, so this takes $O(\sqrt{i}\,\alpha(i)) \subseteq O(i)$ time in total.

Thus, Theorem 5 follows from Lemma 6.

4.1 Proving Lemma 6

We show Lemma 6 via the following weaker helper lemma:

▶ **Lemma 7.** *For every integer m, there is a rooted tree with m leaves, that can be computed in $O(m)$ time, that has the following properties:*
1. *The i-th leaf has a depth of $O(\alpha(i))$*
2. *Every node on the root-to-leaf path of the i-th leaf has $O(i)$ children.*

From Lemma 7, we can obtain Lemma 6 via a subdivision step: For every internal node u with $k \geq 4$ children, we add $\lceil \sqrt{k} \rceil$ new nodes that each have u as their parent and $\Theta(\sqrt{k})$ of the (former) children of u as their children. This at most doubles the depth of every leaf, and reduces the number of children by a square root. What remains is to show Lemma 7.

▶ **Definition 8.** Knuth's up-arrow notation *for integers* $a, b, k \geq 1$ *is defined as follows:* $a \uparrow b := a^b$, $a \uparrow^2 b := a^{a^{\cdots^a}}$ *(b levels), and more generally*

$$a \uparrow^k b := \begin{cases} a^b & k = 1 \\ a & b = 1, k \geq 2 \\ a \uparrow^{k-1} (a \uparrow^k (b-1)) & b, k \geq 2 \end{cases}$$

Note that $\min\{k \mid 3 \uparrow^k 2 > m\} = \alpha(m) \pm O(1)$. *(This follows from the fact that the bivariate definition of the Ackerman function as from [24] satisfies* $A(x, y) = 2 \uparrow^{x-2} (y + 3) - 3$*.)*

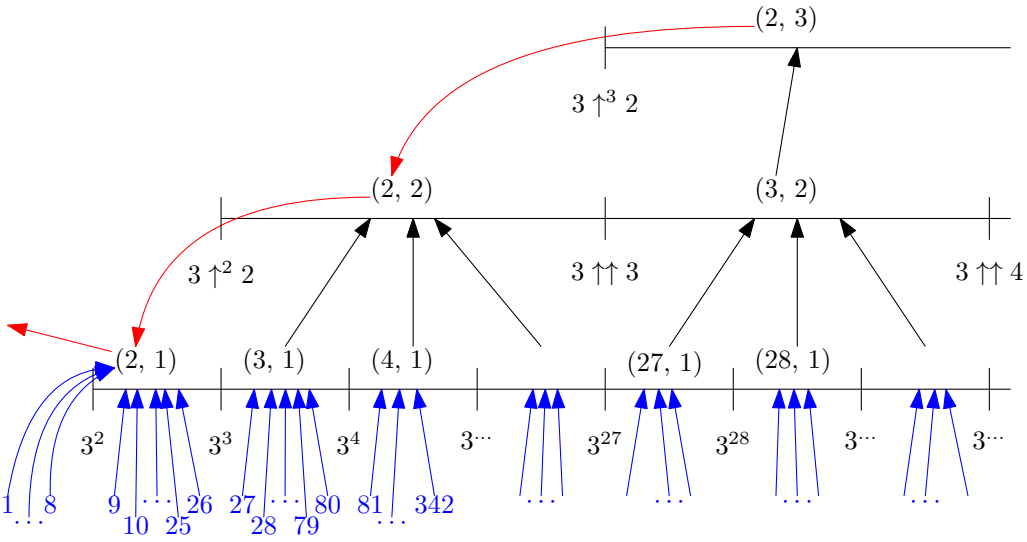

Figure 1 The tree in Lemma 7. Internal nodes are black, leaves are blue. In the proof, the red edges get added to turn the forest into a tree.

To show Lemma 7, we will construct a tree as depicted in Figure 1. Every internal node corresponds to some subinterval of $[1, m]$, and these nodes are arranged in levels. More precisely, the j-th node on level k corresponds to the half-open interval $[3 \uparrow^k j, \min(m+1, 3 \uparrow^k (j+1)))$ where $j \geq 2, k \geq 1$, and $3 \uparrow^k j \leq m$. We denote this node by (j, k). Note that the intervals of nodes on adjacent levels are nested. That is, every interval on level k is the disjoint union of intervals on level $k-1$. This defines a forest structure on the set of nodes: Nodes (j, k) with $j = 1$ are root nodes. All other nodes (j, k) have a parent on level $k+1$. To turn this forest into a tree, we additionally add an edge from node $(1, k+1)$ to node $(1, k)$ for $k \geq 1$. This yields a tree with root node $(2, 1)$. Finally, to every node $(j, 1)$, we attach $O(3^j)$ leaves, numbered $3^j, 3^j + 1, \ldots, \min(m, 3^{j+1} - 1)$. To the root node $(2, 1)$, we attach eight additional leaves, numbered $1, 2, \ldots, 8$.

We check that the resulting tree satisfies all properties of Lemma 6: The tree has m leaves, numbered $1, \ldots, m$. The eight leaves $1, 2, \ldots, 8$ have depth $O(1)$. For the i-th leaf, with $i \geq 8$, there is a unique node $(2, k)$ whose interval contains i. Then $i \geq 3 \uparrow^k 2$, i.e., $k \leq \alpha(i)$. We claim that the i-th leaf has depth $\leq 2k$. Indeed, from the leaf, it takes at most k steps to the node $(2, k)$, and another k steps to the root node $(2, 1)$. This shows property (1) of Lemma 6. What remains is to check property (2). The root node has $O(1)$ children. A node $(j, 1)$ where

$j > 2$ has $O(3^j)$ children. Since the leaves in the subtree of $(j, 1)$ are $3^j, \ldots, \min(m, 3^{j+1} - 1)$, property (2) holds for these nodes. Consider now an arbitrary node (j, k) where $k \geq 2$, with interval $[3 \uparrow^k j, 3 \uparrow^k (j + 1))$. Since $3 \uparrow^k (j + 1) = 3 \uparrow^{k-1} (3 \uparrow^k j)$, the children of this node have an interval of the form $[3 \uparrow^{k-1} \ell, 3 \uparrow^{k-1} (\ell + 1))$ where $\ell + 1 \leq 3 \uparrow^k j$. Thus, the node (j, k) has $\leq 3 \uparrow^k j$ children. On the other hand, if the i-th leaf lies in the subtree of the node (j, k), then $i \in [3 \uparrow^k j, 3 \uparrow^k (j + 1))$, so i is at least the number of children of (j, k). Since (j, k) was arbitrary, this shows property (2).

To compute this tree on a pointer machine in $O(m)$ time, we need two observations: First, the final step of attaching leaves can be implemented in $O(m)$ time. Second, if $3 \uparrow^k j \leq m$, then we can compute $3 \uparrow^k j$ in $O(\log^3 m)$ time. Indeed, if we use the recursion of Definition 8, then there can be at most $\log_3(m)$ multiplications in total, and we can simulate on a pointer machine the multiplication of $O(\log m)$-bit numbers in $O(\log^2 m)$ time. Conversely, if $3 \uparrow^k j > m$, then we can detect this in $O(\log^3(m))$ via the same approach. Thus, we can compute the internal nodes (and their intervals) level-by-level, in $O(\log^3(m))$ time per internal node. Since there are $O(\log m)$ internal nodes, this step takes $o(m)$ time in total.

This shows Lemma 7, which implies Lemma 6 and Theorem 5, which concludes Theorem 3.

5 An age-aware DecreaseKey operation

We briefly note that our result can be extended somewhat. In Theorem 3, the `DecreaseKey` operation takes $O(\alpha(n))$ amortized time. We can perform a slightly tighter analysis and show that the `DecreaseKey` operation on an element x_j takes $O(\alpha(\Gamma(x_j)))$ time.

There are two steps of the `DecreaseKey` operation that take $\omega(1)$ time: The find operation on the union-find structure, and the `DecreaseKey` operation on the index. Since the element x_j lies in a bucket of index $i \in O(\log \Gamma(x_j))$, the `DecreaseKey` operation on the index takes $O(\alpha(i)) \subseteq O(\alpha(\Gamma(x_j)))$ time. For the find operation, we observe that element x_j lies in a set of size $O(\Gamma(x_j))$. Thus, it suffices to prove the following:

▶ **Lemma 9.** *There is a union-find data structure that supports make set and link operations in amortized $O(1)$ time, and the find operation on an element x_j in amortized $O(\alpha(|S|))$ time where S is the set that currently contains x_j.*

The proof of this lemma follows immediately from refining the existing analysis of union-find, making the cost analysis more sensitive to the current set size. Such adaptions are common [1, 24], but for completeness we provide the full argument. We follow the analysis of [30], starting on page 255. There, the cost of a find operation is k, which is later bounded by $O(\alpha(n))$. In the proof of Lemma 3 of [30], we replace k by the variable g that is used in the proof. Since g is bounded by the number of distinct levels in S, we have $g \in O(\alpha(|S|))$. Thus, the find operation takes amortized $O(\alpha(|S|))$ time. This shows Lemma 9 and therefore:

▶ **Theorem 10.** *A heap can be implemented on a pointer machine that, when n denotes the heap size, supports the following operations:*

- *`MakeHeap` in worst-case $O(1)$ time,*
- *`DecreaseKey` in amortized $O(\alpha(\Gamma(x_j)))$ (inverse-Ackermann) time,*
- *`Push` in amortized $O(1)$ time,*
- *`Peek` in worst-case $O(1)$ time, and*
- *`Pop` in worst-case $O(1 + \log \Gamma(x_j))$ time.*

All worst-case operations build no amortized potential. I.e., invoking them does not affect the amortized analysis of the other operations.

References

1 Stephen Alstrup, Mikkel Thorup, Inge Li Gørtz, Theis Rauhe, and Uri Zwick. Union-find with constant time deletions. *ACM Transactions on Algorithms*, 11(1), August 2014. doi:10.1145/2636922.

2 Jon Louis Bentley and James B Saxe. Decomposable searching problems I. Static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301–358, 1980. doi:10.1016/0196-6774(80)90015-2.

3 Gerth Stølting Brodal, George Lagogiannis, and Robert E. Tarjan. Strict Fibonacci heaps. *ACM Trans. Algorithms*, 21(2), January 2025. doi:10.1145/3707692.

4 Timothy M. Chan. *Quake Heaps: A Simple Alternative to Fibonacci Heaps*, pages 27–32. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-40273-9_3.

5 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3 edition, 2009.

6 Clark Allan Crane. *Linear Lists and Priority Queues as Balanced Binary Trees*. PhD thesis, Stanford University, Stanford, CA, USA, 1972. PhD Thesis.

7 Edsger Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. doi:10.1007/BF01386390.

8 Amr Elmasry. A priority queue with the working-set property. *International Journal of Foundations of Computer Science*, 17(06):1455–1465, December 2006. doi:10.1142/S0129054106004510.

9 Amr Elmasry. The violation heap: A relaxed Fibonacci-like heap. In My T. Thai and Sartaj Sahni, editors, *Computing and Combinatorics*, pages 479–488, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. doi:10.1007/978-3-642-14031-0_51.

10 Amr Elmasry, Arash Farzan, and John Iacono. A priority queue with the time-finger property. *Journal of Discrete Algorithms*, 16:206–212, October 2012. doi:10.1016/j.jda.2012.04.014.

11 Michael Fredman and Robert Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987. doi:10.1145/28869.28874.

12 Michael Fredman and Dan Willard. Surpassing the information theoretic bound with fusion trees. *Journal of computer and system sciences*, 47(3):424–436, 1993. doi:10.1016/0022-0000(93)90040-4.

13 Michael L. Fredman. On the efficiency of pairing heaps and related data structures. *Journal of the ACM (JACM)*, 46(4):473–501, July 1999. doi:10.1145/320211.320214.

14 Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1–4):111–129, January 1986. doi:10.1007/BF01840439.

15 Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci.*, 48(3):533–551, June 1994. doi:10.1016/S0022-0000(05)80064-9.

16 Bernhard Haeupler, Richard Hladík, Václav Rozhoň, Robert Tarjan, and Jakub Tětek. Universal optimality of dijkstra via beyond-worst-case heaps. In *Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2024. doi:10.1109/FOCS61266.2024.00125.

17 Bernhard Haeupler, Siddhartha Sen, and Robert E. Tarjan. Rank-pairing heaps. *SIAM Journal on Computing*, 40(6):1463–1485, 2011. doi:10.1137/100785351.

18 Thomas Dueholm Hansen, Haim Kaplan, Robert E. Tarjan, and Uri Zwick. Hollow heaps. *ACM Transactions on Algorithms (TALG)*, 13(3), July 2017. doi:10.1145/3093240.

19 Maria Hartmann, László Kozma, Corwin Sinnamon, and Robert E. Tarjan. Analysis of Smooth Heaps and Slim Heaps. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 198 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 79:1–79:20, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ICALP.2021.79.

20 John Iacono. Improved upper bounds for pairing heaps. In *Scandinavian Workshop on Algorithm Theory*, pages 32–45. Springer, 2000. doi:10.5555/645900.672600.

21 John Iacono. Alternatives to splay trees with o(log n) worst-case access times. In *Symposium on Discrete Algorithms SODA*, pages 516–522. ACM/SIAM, 2001. URL: http://dl.acm.org/citation.cfm?id=365411.365522.

22 John Iacono. *Distribution-sensitive data structures*. PhD thesis, Rutgers University, 2001. AAI3029688.

23 Haim Kaplan, Nira Shafrir, and Robert E. Tarjan. Meldable heaps and boolean union-find. In *ACM Symposium on Theory of Computing (STOC)*, pages 573–582, New York, NY, USA, 2002. Association for Computing Machinery. doi:10.1145/509907.509990.

24 Haim Kaplan, Nira Shafrir, and Robert E. Tarjan. Union-find with deletions. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, SODA '02, pages 19–28, USA, 2002. Society for Industrial and Applied Mathematics.

25 Haim Kaplan, Robert E. Tarjan, and Uri Zwick. Fibonacci heaps revisited. *arXiv preprint arXiv:1407.5750*, 2014. arXiv:1407.5750.

26 Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 1968.

27 László Kozma and Thatchaphol Saranurak. Smooth heaps and a dual view of self-adjusting data structures. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2018, pages 801–814, New York, NY, USA, June 2018. Association for Computing Machinery. doi:10.1145/3188745.3188864.

28 Seth Pettie. Towards a final analysis of pairing heaps. In *Symposium on Foundations of Computer Science (FOCS)*, FOCS '05, pages 174–183, USA, 2005. IEEE Computer Society. doi:10.1109/SFCS.2005.75.

29 Daniel Rutschmann. Sorting under partial information with optimal preprocessing time via unified bound heaps, 2026. doi:10.48550/arXiv.2604.12653.

30 Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, March 1984. doi:10.1145/62.2160.

31 Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975. doi:10.1145/321879.321884.

32 Robert Endre Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18(2):110–127, 1979. doi:10.1016/0022-0000(79)90042-4.

33 Mikkel Thorup. On AC^0 implementations of fusion trees and atomic heaps. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 699–707. ACM/SIAM, 2003. doi:10.1145/644108.644221.

34 Ivor van der Hoog, Eva Rotenberg, and Daniel Rutschmann. Simpler optimal sorting from a directed acyclic graph. In *SIAM Symposium on Simplicity in Algorithms (SOSA)*, 2025. doi:10.1137/1.9781611978315.26.

35 Ivor van der Hoog, Eva Rotenberg, and Daniel Rutschmann. Simpler Universally Optimal Dijkstra. In Anne Benoit, Haim Kaplan, Sebastian Wild, and Grzegorz Herman, editors, *European Symposium on Algorithms (ESA)*, volume 351 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 71:1–71:9, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ESA.2025.71.

36 Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, April 1978. doi:10.1145/359460.359478.

37 J.W.J. Williams. Algorithm 232: Heapsort. *Commun. ACM*, 7(6):347–348, May 2025. doi:10.1145/512274.3734138.