Improved Bounds for Multipass Pairing Heaps and Path-Balanced Binary Search Trees

Dani Dorfman
Blavatnik School of Computer Science, Tel Aviv University, Israel
dannatand@mail.tau.ac.il

Haim Kaplan[1]
Blavatnik School of Computer Science, Tel Aviv University, Israel
haimk@post.tau.ac.il

László Kozma[2]
Eindhoven University of Technology, The Netherlands
lkozma@gmail.com

Seth Pettie[3]
University of Michigan
pettie@umich.edu

Uri Zwick[4]
Blavatnik School of Computer Science, Tel Aviv University, Israel
zwick@tau.ac.il

Abstract

We revisit *multipass* pairing heaps and *path-balanced* binary search trees (BSTs), two classical algorithms for data structure maintenance. The pairing heap is a simple and efficient "self-adjusting" heap, introduced in 1986 by Fredman, Sedgewick, Sleator, and Tarjan. In the multipass variant (one of the original pairing heap variants described by Fredman et al.) the minimum item is extracted via repeated *pairing rounds* in which neighboring siblings are linked.

Path-balanced BSTs, proposed by Sleator (cf. Subramanian, 1996), are a natural alternative to Splay trees (Sleator and Tarjan, 1983). In a path-balanced BST, whenever an item is accessed, the search path leading to that item is re-arranged into a balanced tree.

Despite their simplicity, both algorithms turned out to be difficult to analyse. Fredman et al. showed that operations in multipass pairing heaps take amortized $O(\log n \cdot \log\log n / \log\log\log n)$ time. For searching in path-balanced BSTs, Balasubramanian and Raman showed in 1995 the same amortized time bound of $O(\log n \cdot \log\log n / \log\log\log n)$, using a different argument.

In this paper we show an explicit connection between the two algorithms and improve both bounds to $O\left(\log n \cdot 2^{\log^* n} \cdot \log^* n\right)$, respectively $O\left(\log n \cdot 2^{\log^* n} \cdot (\log^* n)^2\right)$, where $\log^*(\cdot)$ denotes the slowly growing iterated logarithm function. These are the first improvements in more than three, resp. two decades, approaching the information-theoretic lower bound of $\Omega(\log n)$.

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

Keywords and phrases data structure, priority queue, pairing heap, binary search tree

[1] Research supported by The Israeli Centers of Research Excellence (I-CORE) program (Center No. 4/11), Israel Science Foundation grant no. 1841-14.
[2] Supported by ERC grant no. 617951. Work done while at Tel Aviv University, research supported by The Israeli Centers of Research Excellence (I-CORE) program (Center No. 4/11).
[3] Supported by NSF grants CCF-1514383 and CCF-1637546.
[4] Research supported by BSF grant no. 2012338 and by The Israeli Centers of Research Excellence (I-CORE) program (Center No. 4/11).

Digital Object Identifier 10.4230/LIPIcs.ESA.2018.24

Related Version A full version of the paper is available at `https://arxiv.org/abs/1806.08692`.

1 Introduction

Binary search trees (BSTs) and heaps are the canonical comparison-based implementations of the well-known *dictionary* and *priority queue* data types.

In a balanced **BST** all standard dictionary operations (*insert*, *delete*, *search*) take $O(\log n)$ time, where n is the size of the dictionary. Early research has mostly focused on structures that are kept (approximately) balanced throughout their usage. (AVL-, red-black-trees, and randomized treaps are important examples, see e.g., [11, § 6.2.2]). These data structures re-balance themselves when necessary, guided by auxiliary data stored in every node.

By contrast, Splay trees (Sleator, Tarjan, 1983 [17]) achieve $O(\log n)$ amortized time per operation without any explicit balancing strategy and with no bookkeeping whatsoever. Instead, Splay trees re-adjust the search path *after every access*, in a way that depends only on the shape of the search path, ignoring the global structure of the tree. Besides the $O(\log n)$ amortized time, Splay trees are known to satisfy stronger, adaptive properties (see [9, 3] for surveys). They are, in fact, conjectured to be optimal on every sequence of operations (up to a constant factor); this is the famous "dynamic optimality conjecture" [17]. Splay trees and data structures of a similar flavor (i.e., local restructuring, adaptivity, no auxiliary data) are called "self-adjusting".

The efficiency of Splay trees is intriguing and counter-intuitive. They re-arrange the search path by a sequence of double rotations ("zig-zig" and "zig-zag"), bringing the accessed item to the root. It is not hard to see that this transformation results in "approximate depth-halving" for the nodes on the search path; the connection between this depth-halving and the overall efficiency of Splay trees is, however, far from obvious.

An arguably more natural approach for BST re-adjustment would be to turn the search path, after every search, into a balanced tree.[5] This strategy combines the idea of self-adjusting trees with the more familiar idea of balancedness. Indeed, this algorithm was proposed early on by Sleator (see e.g., [19, 1]). We refer to BSTs maintained in this way as *path-balanced* BSTs (see Figure 1).

Path-balanced BSTs turn out to be surprisingly difficult to analyse. In 1995, Balasubramanian and Raman [1] showed the upper bound of $O(\log n \cdot \log\log n / \log\log\log n)$ on the cost of operations in path-balanced BSTs. This bound has not been improved since. Thus, path-balanced BSTs are not known to match the $O(\log n)$ amortized cost (let alone the stronger adaptive properties) of Splay. This is surprising, because broad classes of BSTs are known to match several guarantees of Splay trees [19, 2], path-balanced BSTs, however, fall outside these classes.[6] Without evidence to the contrary, one may even conjecture path-balanced BSTs to achieve dynamic optimality; yet our current upper bounds do not even match those of a *static* balanced tree. This points to a large gap in our understanding of a natural heuristic in the fundamental BST model.

[5] The restriction to touch only the search path is natural, as the cost of doing this is proportional to the *search cost*. (A BST can be changed into any other BST with a linear number of rotations [16].)

[6] Intuitively, path-balance is different, and more difficult to analyse than Splay, because it may increase the depth of a node by an additive $O(\log n)$, whereas Splay may increase the depth of a node by at most 2. In a precise sense, path-balance is not a *local* transformation (see [2]).

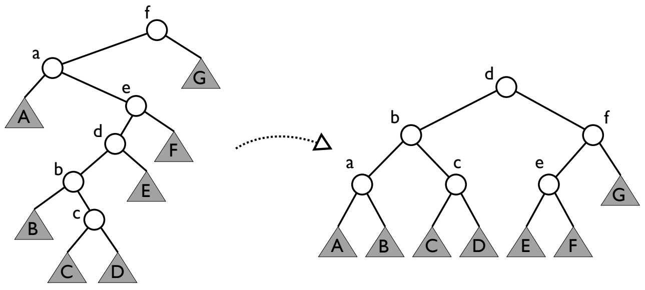

Figure 1 Access in a path-balanced BST. Search path (f, a, e, d, b, c) from root f to accessed item c is re-arranged into a balanced tree with subtrees (denoted by capital letters) re-attached.

In this paper we show that the amortized time of an access[7] in a path-balanced BST is $O\left(\log n \cdot (\log^* n)^2 \cdot 2^{\log^* n}\right)$. The result, probably not tight, comes close to the information-theoretic lower bound of $\Omega(\log n)$. Closing the gap remains a challenging open problem.

Priority queues support the operations *insert*, *delete-min*, and possibly *meld*, *decrease-key* and others. Pairing heaps, a popular priority queue implementation, were proposed in the 1980s by Fredman, Sedgewick, Sleator, and Tarjan [5] as a simpler, self-adjusting alternative to Fibonacci heaps [6]. Pairing heaps maintain a multi-ary tree whose nodes (each with an associated key) are in heap order. Similarly to Splay trees, pairing heaps only perform key-comparisons and simple local transformations on the underlying tree, with no auxiliary data stored. Fredman et al. showed that in the standard pairing heap all priority queue operations take $O(\log n)$ time. They also proposed a number of variants, including the particularly natural *multipass pairing heap*. In multipass pairing heaps, the crucial *delete-min* operation is implemented as follows. After the root of the heap (i.e., the minimum) is deleted, repeated pairing rounds are performed on the new top-level roots, reducing their number until a single root remains. In each pairing round, neighboring pairs of nodes are *linked*. Linking two nodes makes the one with the larger key the *leftmost* child of the other (Figure 2).

Pairing heaps perform well in practice [18, 14, 12]. However, Fredman [4] showed that all of their standard variants (including the multipass described above) fall short of matching the theoretical guarantees of Fibonacci heaps (in particular, assuming $O(\log n)$ cost for delete-min, the average cost of *decrease-key* may be $\Omega(\log \log n)$, in contrast to the $O(1)$ guarantee for Fibonacci heaps). The exact complexity of the standard pairing heap on sequences of intermixed delete-min, insert, and decrease-key operations remains an intriguing open problem, with significant progress through the years (see e.g., [8, 15]). However, for the multipass variant, even the basic question of whether deleting the minimum takes $O(\log n)$ amortized time remains open, the best upper bound to date being the $O(\log n \cdot \log \log n / \log \log \log n)$ originally shown by Fredman et al. Similarly to the case of path-balanced BSTs, we have thus a basic combinatorial transformation on trees, whose complexity is not well-understood.

In this paper we show that in multipass pairing heaps delete-min[8] takes amortized time $O\left(\log n \cdot \log^* n \cdot 2^{\log^* n}\right)$, the first improvement since the original paper of Fredman et al. The improvement is, from a practical perspective, not significant. Nonetheless, it reduces the gap to the theoretical optimum from $(\approx \log^{(2)} n)$ to less than $\log^{(k)} n$ for any fixed k.

[7] We only focus on successful search operations (i.e., accesses). The results can be extended to other operations at the cost of technicalities. For simplicity, we assume that the keys in the tree are unique.
[8] To keep the presentation simpler, we only focus on *delete-min* operations, omitting the extension of the result to other operations.

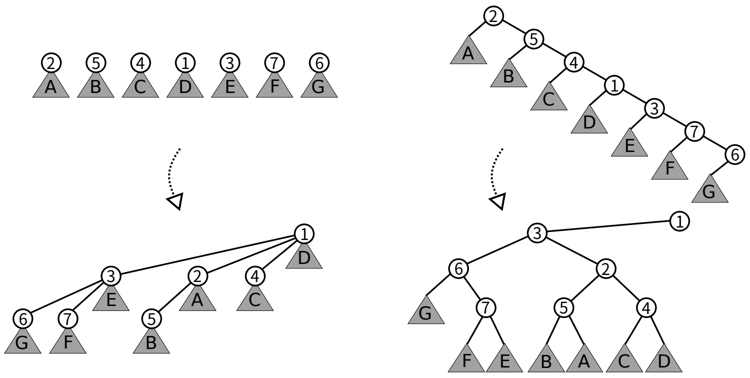

Figure 2 Delete-min in a multipass pairing heap. (above) state after deleting the root, with list of siblings; (below) state after three pairing rounds, with links $(2, 5)$, $(4, 1)$, $(3, 7)$, $(2, 1)$, $(3, 6)$, $(1, 3)$. (left) multi-ary view; (right) binary view. Numbers denote keys, capital letters denote subtrees.

The reader may notice that the old bounds for multipass pairing heaps and path-balanced BSTs are the same. The two data structures are, indeed, quite similar: if one views multipass pairing heaps as binary trees (see e.g., [10, § 2.3.2]), the multipass re-adjustement is equivalent to balancing the right-spine of a binary tree.[9] The multipass analysis, however, does not immediately transfer to path-balanced BSTs; the fact that the BST search path may be arbitrary (not necessarily right-leaning) complicates the argument for path-balanced BSTs.

Our analysis of multipass pairing heaps (§ 2) is based on a new, fine-grained scaling of the sum-of-logs potential function used by Sleator and Tarjan in the analysis of Splay trees, and by Fredman et al. in the analysis of pairing heaps. At a high level, we argue that certain link operations are information-theoretically efficient, and that such links happen sufficiently often. The subsequent, rather intricate analysis notwithstanding, we believe that the ideas of the proof may have further applications in the analysis of data structures.

In § 3 we show our result for path-balanced BSTs. Informally, we decompose the path-balancing operation into several stages, each of which resembles the multipass transformation, allowing us to adapt and reuse the result of § 2. For lack of space, we omit several proofs (marked $\star$) in this version of the paper and refer to the longer preprint[10] for details and additional illustrations.

2 Multipass pairing heaps

A *pairing heap* is a multi-ary heap, storing a key in each node, with the regular (min)heap-condition: the key of a node is smaller than the keys of its children. Priority queue operations are implemented using the unit-cost *linking* step. Given nodes x, y, $\mathsf{link}(x, y)$ "hangs" the node with the larger key as the *leftmost* child of the other. The operations *insert*, *meld*, and *decrease-key* can be implemented in a straightforward way using a single *link* (we refer to [5] for details). The only nontrivial operation is *delete-min*. Here, after deleting the root, we are left with a number of top-level nodes, which we combine into a single tree via a sequence of *links*. In multipass pairing heaps we achieve this by performing repeated *pairing rounds*, until a single top-level node remains (i.e., the new root of the heap). A single pairing round

[9] We note that the previous analysis of path-balanced BSTs [1] did not use this correspondence. By connecting the two data structures, we also simplify (to some extent) the proof of [1].
[10] `https://arxiv.org/abs/1806.08692`

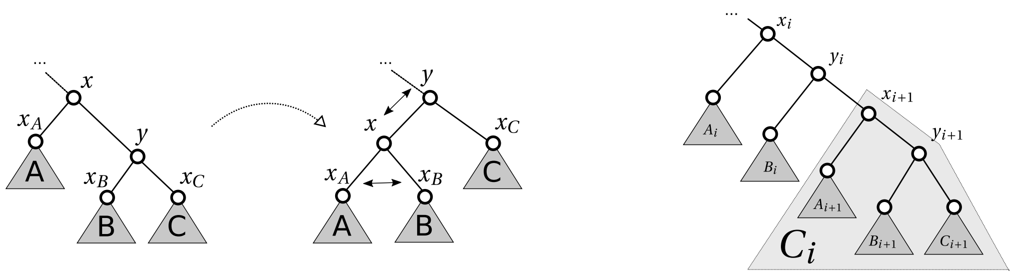

Figure 3 Left: $\mathsf{link}(x, y)$ in binary tree view. Dots (...) indicate the sequence of nodes that have already been linked in the current round, subtree C contains the yet-to-be-linked nodes. Arrows indicate possible switching depending on the outcome of the comparison between x and y. The roots of A,B, and C are denoted x_A, x_B, and x_C. Right: i-th link in a round (between x_i and y_i). The subtree rooted at the right child of y_i is denoted C_i; observe that C_i contains C_{i+1}.

is as follows. Let $x_1, \dots, x_\ell$ be the top-level nodes, ordered left-to-right, before the round. For all $1 \le i \le \lfloor \ell/2 \rfloor$ we perform $\mathsf{link}(x_{2i-1}, x_{2i})$. Observe that if ℓ is odd, then the rightmost node is unaffected in the current round. The number of rounds is $\lceil \log(k) \rceil$, where k is the number of children of the (deleted) root.[11] (See Figure 2.)

We now analyse delete-min operations implemented by multipass pairing heaps. Let k be the number of children of the deleted root, defined to be the real cost of the operation (observe that the number of links is exactly $k - 1$). Let n be the size of the heap before the operation. We use the binary tree view of multi-ary heaps, where the *leftmost child* and *next sibling* pointers are interpreted as *left child* and *right child*. A single link operation is shown in Figure 3. Let a, b, c denote the sizes of subtrees A, B, and C, respectively.

We define a potential function that refines the Sleator-Tarjan "sum-of-logs" potential [17]. Let $\Phi = \sum_{x \in T} \phi(x)$, over all nodes x of the heap T, where

$$\phi(x) = \frac{H(x)}{\log^2(2 + H(x))}, \quad \text{and} \quad H(x) = \log\left(\frac{s(p(x))}{s(x)}\right),$$

where $s(x)$ denotes the size of the subtree rooted at x, and $p(x)$ is the parent of x.[12] Note that both *subtrees* and *parents* are meant in the binary tree view.

For convenience, define the functions

$$f(x) = \log x / \log^2(2 + \log x), \quad \text{and} \quad g(x) = x / \log^2(2 + x).$$

With this notation, $f(x) = g(\log(x))$, and $\phi(x) = f\left(\frac{s(p(x))}{s(x)}\right)$. Clearly, both $f(x)$ and $g(x)$ are positive, monotone increasing, and concave, for all $x \ge 1$, respectively, $x \ge 0$.

By simple arithmetic, the increase in potential due to a single link (as in Figure 3) is:

$$\begin{aligned}\Delta\Phi \;=\; & f\left(\frac{a+b+1}{a}\right) + f\left(\frac{a+b+1}{b}\right) + f\left(\frac{a+b+c+2}{a+b+1}\right) + f\left(\frac{a+b+c+2}{c}\right) \\ & - f\left(\frac{a+b+c+2}{a}\right) - f\left(\frac{a+b+c+2}{b+c+1}\right) - f\left(\frac{b+c+1}{b}\right) - f\left(\frac{b+c+1}{c}\right). \quad (1)\end{aligned}$$

[11] The function $\log(\cdot)$ is base 2 everywhere, the base e logarithm is written as $\ln(\cdot)$.
[12] Using $\phi(x) = H(X)$ instead, would essentially recover the original "sum-of-logs" potential. Such an "edge-based" potential function was used earlier, e.g., in [7, 13].

For a suitably large constant γ (for concreteness let $\gamma = 3000$), we consider the quantities $\gamma^2 a$, γb, and c, i.e., the scaled sizes of the subtrees A, B, and C. We distinguish different kinds of links, depending on the ordering of the three quantities (breaking ties arbitrarily). We first look at the cases when $\gamma^2 a$ or γb is the largest (called respectively type-(1) and type-(2) links), and show that the possible increase in potential due to such links is small. In particular, for type-(1) links, $\Delta\Phi$ is dominated by a term $f(a/c)$, and for type-(2) links the positive and negative contributions cancel out, leaving $\Delta\Phi = O(1)$. The proofs (omitted here) use standard (although somewhat delicate) analysis.

▶ **Lemma 1** ($\star$). *A type-(1) link ($\gamma^2 a \geq \max\{\gamma b, c\}$) increases the potential Φ by at most $2 \cdot g\big(\log(a/c) + O(1)\big)$, where the $O(1)$ term is a constant independent of a, b, c, n, and k.*

▶ **Lemma 2** ($\star$). *A type-(2) link ($\gamma b \geq \max\{\gamma^2 a, c\}$) increases Φ by at most $O(1)$.*

The case when c is the greatest of the three quantities (called type-(3) link) is the most favorable. Here, the potential of x_A, x_B before the linking is (roughly) the logarithm of $s(x_C)$ (very large) divided by $s(x_A), s(x_B)$; after the linking, the potential becomes (essentially) the logarithm of the ratio between $s(x_A)$ and $s(x_B)$ (much smaller), resulting in a significant saving in potential. We use this saving to "pay" for the operations. First we make the following, easier claim.

▶ **Lemma 3** ($\star$). *A type-(3) link ($c \geq \max\{\gamma^2 a, \gamma b\}$) can not increase Φ.*

It remains to balance the *decrease* in potential due to type-(3) links and the *increase* in potential due to all other links. First, we show that almost all links are type-(3).

▶ **Lemma 4.** *There are at most $O(\log n)$ type-(1) and type-(2) links within a pairing round.*

Proof. Let a_i, b_i, c_i denote the subtree-sizes corresponding to the i-th link *from left to right*, see Figure 3(right). Let the subsequences a_{i_t}, b_{i_t}, c_{i_t}, $t = 1, \dots, m$ be the subtree-sizes corresponding to type-(1) and type-(2) links. Observe that $c_{i_1} \geq \cdots \geq c_{i_m}$. If the i-th link is of type-(1) or type-(2), then $c_{i-1} = 2 + a_i + b_i + c_i \geq (1 + 1/\gamma^2) \cdot c_i$, since in each of these cases $a_i \geq 1/\gamma^2 c_i$ or $b_i \geq 1/\gamma^2 c_i$. Since $c_{i_1} \leq n$, and $c_{i_m} \geq 1$ the claim follows. ◀

▶ **Lemma 5.** *All type-(1) and type-(2) links within a single pairing round increase the potential by at most $O(\log n)$.*

Proof. Look at a single round of pairing. Let a_{i_t}, b_{i_t}, c_{i_t} ($t = 1, \dots, m$) be as in the proof of Lemma 4 and recall that $m = O(\log n)$. If the i_t-th link is type-(1), then by Lemma 1, the increase in potential is at most $2 \cdot g\big(\log(a_{i_t}/c_{i_t}) + O(1)\big)$.

Otherwise, if the i_t-th link is type-(2), then by Lemma 2, the increase in potential is at most $O(1)$, which we can write as $2 \cdot g(c')$, for a suitable constant c'.

Let q_t denote $\log(a_{i_t}/c_{i_t}) + O(1)$, or c', corresponding to the i_t-th link (according to its type). We have $\sum q_i \leq \alpha \cdot \log n$ (for a fixed constant $\alpha \geq 1$), since the sum of the $\log(a_i/c_i)$ terms telescopes, and the additive $O(1)$ (or c') terms appear at most $m = O(\log n)$ times.

The total increase in potential is at most $\Delta\Phi = 2 \cdot \sum_{t=1}^{m} g(q_t)$. By the concavity of $g(\cdot)$, $\Delta\Phi$ is maximized if all of the arguments of $g(\cdot)$ are equal. We thus obtain a bound on the total increase in potential in the pairing round.

$$\Delta\Phi \leq 2m \cdot g\left(\frac{\alpha \cdot \log n}{m}\right) = \frac{2\alpha \log n}{\log^2\left(2 + \alpha \cdot (\log n)/m\right)} = O\left(\log n\right). \quad ◀$$

The last proof yields, in fact, the following stronger claim.

▶ **Lemma 6.** *All type-(1) and type-(2) links within the last* $(\log\log n)$ *pairing rounds increase the potential by at most* $O(\log n)$*.*

Proof. Observe that for $j < \log\log n$, the j-th to the last pairing round has at most $m \leq 2^j < \log n$ links. Thus, as in Lemma 5, we obtain:

$$\Delta\Phi \leq \frac{2\alpha\log n}{\log^2(2+\alpha\cdot(\log n)/m)} \leq \frac{2\alpha\log n}{\log^2(\alpha\cdot(\log n)/2^j)} = \frac{2\alpha\log n}{((\log\log n+\log\alpha)-j)^2}.$$

Note that the second inequality holds since $2^j < \log n$. The sum of this expression over all $(\log\log n)$ levels j is $O(\log n)$. (Using the fact that $\sum_k 1/k^2$ converges to a constant.) ◀

Now we estimate more carefully the decrease in potential due to type-(3) links. Let x_A and x_B be nodes as denoted in Figure 3. We want to express the potential-change in terms of $H_A = H(x_A)$ and $H_B = H(x_B)$ (before the link operation). Recall that $H_A = \log\left(\frac{a+b+c+2}{a}\right)$ and $H_B = \log\left(\frac{b+c+1}{b}\right)$.

Among type-(3) links ($c \geq \max\{\gamma^2 a, \gamma b\}$) we distinguish two subtypes: type-(3A) ($\gamma^2 a \geq \gamma b$), and type-(3B) ($\gamma b \geq \gamma^2 a$). We have the following two (symmetric) observations:

▶ **Lemma 7** ($\star$)**.** *A type-(3A) link* ($c \geq \gamma^2 a \geq \gamma b$) *decreases the potential by at least*

$$\Omega(1)\cdot\frac{H_A}{\log^2(2+H_B)} - O(1).$$

It follows that for some constant d_1, if $H_A \geq d_1\cdot\log^2(2+H_B)$, then $\Delta\Phi \leq -1$.

▶ **Lemma 8** ($\star$)**.** *A type-(3B) link* ($c \geq \gamma b \geq \gamma^2 a$) *decreases the potential by at least*

$$\Omega(1)\cdot\frac{H_B}{\log^2(2+H_A)} - O(1).$$

It follows that for some constant d_2, if $H_B \geq d_2\cdot\log^2(2+H_A)$, then $\Delta\Phi \leq -1$.

▶ **Corollary 9.** *There exists a constant* d $(=\max(d_1,d_2))$ *such that all type-(3A) links with* $H_A \geq d\cdot\log^2(2+H_B)$ *and all type-(3B) links with* $H_B \geq d\cdot\log^2(2+H_A)$ *decrease the potential by at least* 1*.*

We now define the *category* of a node with respect to its $H(\cdot)$ value. Intuitively, nodes of the same category are those that, when linked, release the most potential. Let us denote $h(x) = d\cdot\log^2(2+x)$. Using the notation of function composition, let

$$h^{(0)}(x) = x,\quad h^{(i)}(x) = h\left(h^{(i-1)}(x)\right).$$

The category of a node is based on the values $h^{(i)}(\log n)$, $i = 1,\ldots,\log^* n$. Note that $h^{(0)}(\log n) = \log n$, $h^{(1)}(\log n) = d\cdot\log^2(2+\log n),\ldots,h^{(\log^* n)}(\log n) = O(1)$, where the $O(1)$ depends on d, since (using the *star* notation) $h^*(n) \leq \left(\log^3\right)^*(n) + O(1) = \log^* n + O(1)$.

▶ **Definition 10** (Category)**.** Let u be a node. For $i = 1,\ldots,\log^* n$, we let $\mathsf{cat}(u) = i$ if:

$$H(u.\mathsf{left}) \in (h^{(i)}(\log n), h^{(i-1)}(\log n)].$$

If $H(u.\mathsf{left}) \leq h^{(\log^* n)}(\log n)$ we say that u is of category 0.

The following crucial observations connect categories and savings in potential.

▶ **Lemma 11.** *Let link(u, v) be type-(3). If* $\mathsf{cat}(u) = \mathsf{cat}(v) \neq 0$*, then the link decreases the potential by at least* 1*.*

Proof. Note that if $i = \mathsf{cat}(u) = \mathsf{cat}(v) \neq 0$ then

$$H(u.\mathsf{left}) \geq h^{(i)}(\log n) \geq d \cdot \log^2(2 + H(v.\mathsf{left})),$$

$$H(v.\mathsf{left}) \geq h^{(i)}(\log n) \geq d \cdot \log^2(2 + H(u.\mathsf{left})).$$

Thus, by Corollary 9, the claim follows. ◀

▶ **Lemma 12.** *In each pairing round there are at most* $O(\log n)$ *nodes of category* 0*.*

Proof. Let x be of category 0, then $H(x.\mathsf{left}) = O(1)$. Denoting $a = s(x.\mathsf{left})$, $c = s(x.\mathsf{right})$, we get $H(x.\mathsf{left}) = \log \frac{a+c+1}{a} = O(1)$. Therefore, $a = \Omega(c)$, an occurrence that can happen at most $O(\log n)$ times in each round (by the same argument as in Lemma 4). ◀

▶ **Lemma 13.** *Let w denote the "winner" of linking x and y (neither of category* 0*), i.e., w is the one with the smaller key. Then* $\mathsf{cat}(w) \geq \max\{\mathsf{cat}(x), \mathsf{cat}(y)\}$*.*

Proof. Let $y = x.\mathsf{right}$, $a = s(x.\mathsf{left}), b = s(y.\mathsf{left}), c = s(y.\mathsf{right})$ as in Figure 3. We have that $H(x.\mathsf{left}) = \log \frac{a+b+c+2}{a}$, $H(y.\mathsf{left}) = \log \frac{b+c+1}{b}$, and $H(\mathsf{link}(x, y).\mathsf{left}) = \log \frac{a+b+c+2}{a+b+1}$.

Clearly $\frac{a+b+c+2}{a+b+1} \leq \min\{\frac{a+b+c+2}{a}, \frac{b+c+1}{b}\}$, finishing the proof. ◀

As seen in Figure 2, a delete-min operation transforms the "spine" of the heap (in binary view) into a balanced tree. We denote this tree by T. Each level of T corresponds to a pairing round; specifically, level i of T consists of nodes at distance i from the leaves, containing the *losers* of the i-th pairing round. The following lemma captures the potential reduction that yields the main result.

▶ **Lemma 14.** *Let T' be a subtree of T of depth* $\log^* n$*, whose leaves correspond to* $2^{\log^* n}$ *consecutive link operations. If T' contains only type-(3) links and no links involving nodes of category* 0*, then the total decrease in potential caused by the links of T' is at least* 1*.*

Proof. Assume towards contradiction that there is no link between two nodes of the same category in T'. By Lemma 13 in each round the minimal overall category increases by at least 1, leaving us with two nodes of maximal category in the last round, a contradiction. By Lemma 11, a link between nodes of equal category decreases the potential by at least 1. ◀

▶ **Theorem 15.** *The amortized time of delete-min in multipass pairing heaps is* $O(\log n \cdot \log^* n \cdot 2^{\log^* n})$*.*

Proof. Let the real cost (number of link operations) be k. Note that there are at most $\lceil \log k \rceil$ pairing rounds.

Thus, if $k \leq \log n \cdot \log^* n \cdot 2^{\log^* n}$, then there are at most $\log\log n + \log\log^* n + \log^* n + 1$ rounds. Using Lemma 5 we get that the first $\log\log^* n + \log^* n + 1 = O(\log^* n)$ pairing rounds increase the potential by at most $O(\log n \cdot \log^* n)$. Also, as shown in Lemma 6, the total increase in potential for the last $\log\log n$ levels is $O(\log n)$. Thus, the total potential increase is at most $O(\log n) + O(\log n \cdot \log^* n)$.

To analyse the case $k > \log n \cdot \log^* n \cdot 2^{\log^* n}$, we use the potential decrease of type-(3) links. First, we look at the first $\log^* n$ pairing rounds.

By Lemma 14, the links in every complete subtree of T of depth $\log^* n$, in which there are only type-(3) links and no category-0 nodes, decrease the potential by at least 1.

In the first $\log^* n$ levels of T we can find $\frac{k}{2^{\log^* n}}$ *disjoint* subtrees of this size. In these levels there are at most $O(\log^* n \cdot \log n)$ type-(1),(2) links, or links containing category-0 nodes (Lemmas 4 and 12). Thus, at least $\frac{k}{2^{\log^* n}} - O(\log^* n \cdot \log n)$ of the subtrees answer the conditions of Lemma 14, decreasing the potential by at least $\frac{k}{2^{\log^* n}} - O(\log^* n \cdot \log n)$. Also, the total increase in potential caused by type-(1),(2) links is at most $O(\log n \cdot \log^* n)$ (Lemma 5). Therefore, the first $\log^* n$ levels give us a decrease in potential of at least $\frac{k}{2^{\log^* n}} - O(\log^* n \cdot \log n)$.

Note that by using the same argument on the next $\log^* n$ levels, we get a decrease in potential of at least $\frac{k'}{2^{\log^* n}} - O(\log^* n \cdot \log n)$, where k' is the number of links in level $\log^* n+1$. Thus, levels which contain $\Omega\left(\log n \cdot \log^* n \cdot 2^{\log^* n}\right)$ links only decrease the potential.

We repeat this argument until we reach a level in T containing $\tilde{k} \leq \log n \cdot \log^* n \cdot 2^{\log^* n}$ links. Now, applying the same argument as for the first case, we get that the total increase in potential for the last $\log \tilde{k}$ levels (starting from the level of $\tilde{k}$ links) is at most $O(\log n \cdot \log^* n)$.

Summarizing, the total amortized time (in both cases) is at most

$$k + O(\log n \cdot \log^* n) - \left(\frac{k}{2^{\log^* n}} - \log^* n \cdot \log n\right).$$

Scaling the potential by $2^{\log^* n}$, we get that the amortized time is $O(\log n \cdot \log^* n \cdot 2^{\log^* n})$. ◀

3 Path-balanced binary search trees

Consider the operation of accessing a node x in a BST T with n nodes (we refer interchangeably to a node and its key). Let $\mathcal{P}^x$ denote the search path to x (i.e., the path from the root of T to x). The path-balance method re-arranges $\mathcal{P}^x$ into a complete balanced BST (with all levels complete, except possibly the lowest). Subtrees hanging off $\mathcal{P}^x$ are re-attached in the unique way given by the key-order (Figure 1). There are multiple ways to implement this transformation such that the number of pointer moves and pointer changes is linear in the length of the search path. For instance, we may first rotate the search path into a *monotone* path, then apply a *multipass transformation* (described next) to this monotone path.

Multipass transformation. A multipass transformation of a monotone path $\mathcal{P}$ (of which the deepest node might not be a leaf) converts $\mathcal{P}$ into a balanced tree (in which the last level may be incomplete) by a sequence of *pairing rounds*. In each pairing round we rotate every other edge in a prefix of $\mathcal{P}$ (i.e., a subpath of the shallowest nodes on $\mathcal{P}$). Each rotation pushes one node off $\mathcal{P}$. We denote by $\mathcal{P}^i$ the path remaining of $\mathcal{P}$ after i pairing rounds. The pairing rounds are defined as follows. We assume that the path consists of right child pointers; in the case it consists of left child pointers everything is symmetric.

Let $\ell(\mathcal{P})$ denote the length of $\mathcal{P}$ (i.e., the number of nodes on $\mathcal{P}$). In the first round we do just enough rotations so that the length of the path after the round (i.e., $\mathcal{P}^1$) is one less than a power of 2. Specifically, we do α rotations where α is the smallest integer such that $\ell(\mathcal{P}^1) = \ell(\mathcal{P}) - \alpha = 2^j - 1$. In the second round we do $2^{j-1} - 1$ rotations on $\mathcal{P}^1$, and in round $i > 1$ we do $2^{j-i+1} - 1$ rotations on $\mathcal{P}^{i-1}$. We maintain the invariant that after $i+1$ rounds all the nodes that were pushed off $\mathcal{P}$ (excluding those that were pushed off $\mathcal{P}$ at the first round) are arranged in balanced binary trees of height $(i-1)$, hanging as children of the nodes of $\mathcal{P}^{i+1}$.

The proof of the following theorem is analogous to the proof of Theorem 15 (one can verify that all steps of the proof still hold for the slightly modified pairing rounds of the multipass transformation, replacing rotations by links).

▶ **Theorem 16.** *For every monotone path $\mathcal{P}$ with $\ell(\mathcal{P}) = k$, the change in Φ caused by applying a multipass transformation on $\mathcal{P}$ is bounded by $\Delta\Phi \le c(n,k) := -\dfrac{k}{2^{\log^* n}} + O(\log n \cdot \log^* n)$, where n is the size of the subtree of the root of $\mathcal{P}$.*

Warm-up: a simplified path-balance. We first look at an easier-to-analyse variant of path-balance, where, instead of a complete balanced tree, we build an *almost* balanced tree out of the search path $\mathcal{P}^x$, as follows: we first make the accessed item x the root, then turn the parts of $\mathcal{P}^x$ containing items smaller (resp. larger) than x into balanced subtrees rooted at the left (resp. right) child of x. The depth of this tree is at most one larger than the depth of a complete balanced tree built from $\mathcal{P}^x$.

For the purpose of the analysis, we view the simplified path-balance transformation as a two-step process. The actual implementation may be different but the analysis applies as long as the transformation takes time $O(\ell(\mathcal{P}^x))$.

Step 1. Rotate the accessed element x all the way to the root. (Observe that after this step, $\mathcal{P}^x$ is split into two monotone paths, $\mathcal{P}^{<x}$ to the left of x consisting only of "right child" pointers, and $\mathcal{P}^{>x}$ to the right of x, consisting only of "left child" pointers.)

Step 2. Apply a multipass transformation to $\mathcal{P}^{>x}$ and to $\mathcal{P}^{<x}$.

We show that the amortized time of an access using simplified path-balance is $O(\log n \cdot \log^* n \cdot 2^{\log^* n})$. We use the same potential function as in § 2, and we assume the two-step implementation described above. We first state an easy observation.

▶ **Lemma 17.** *Let $\mathcal{P}$ be a path in T rooted at a node r, then $\Phi(\mathcal{P}) = O(\log s(r))$, where $\Phi(\mathcal{P}) = \sum_{x \in \mathcal{P}} \phi(x)$ and $s(r)$ is the size of the subtree of r.*

Proof. Denote $\ell = \ell(\mathcal{P})$. Let $a_1 \le \ldots \le a_\ell = s(r)$ be the subtree-sizes of the nodes on $\mathcal{P}$ from the deepest node to r. Then

$$\Phi(\mathcal{P}) = \sum_{k=1}^{\ell-1} f\left(\frac{a_{k+1}}{a_k}\right) = \sum_{k=1}^{\ell-1} g\left(\log \frac{a_{k+1}}{a_k}\right) \le \ell \cdot g\left(\frac{\log s(r)}{\ell}\right) = O(\log s(r)),$$

due to g's concavity and since the terms $\log \frac{a_{k+1}}{a_k}$ sum to $\log s(r) - \log a_1 \le \log s(r)$. ◀

We proceed with the analysis. We argue that rotating x to the root (Step 1) increases Φ by at most $O(\log n)$. To see this, observe first, that the potential of nodes hanged on the nodes of $\mathcal{P}^x$ excluding x, can only decrease. This is because their subtree remains the same, whereas the subtree of their parent (a node on the search path) can only lose elements. The two children of x may increase the potential by at most $O(\log n)$.

For nodes *on the search path*, we look at the potential after the transformation. We have two separate paths, and by Lemma 17 the potential of each path is bounded by $O(\log n)$. This concludes the analysis for Step 1.

In Step 2, as we apply the multipass transformation to both $\mathcal{P}^{<x}$ and $\mathcal{P}^{>x}$, Theorem 16 applies. Thus, $\Delta\Phi$ is at most $c(s(x.\mathsf{left}), \ell(\mathcal{P}^{<x})) + c(s(x.\mathsf{right}), \ell(\mathcal{P}^{>x}))$ where $c(n,k)$ is defined in Theorem 16. The claim on the amortized running time follows by scaling $\Delta\Phi$ by $2^{\log^* n}$ and adding it to the actual cost (the length of $\mathcal{P}^x$). This concludes the proof.

Analysis of path-balance. The original path-balance heuristic (where we insist on building a *complete* balanced tree) is trickier to analyse. Here, instead of moving the accessed item x to the root, we move the *median* item m of the search path $\mathcal{P}^x$ to the root. Here, "median" is meant with respect to the ordering of keys; m is, in general, *not* the node with median depth on $\mathcal{P}^x$. It is instructive to prove the earlier $O(\log n \cdot \log\log n/\log\log\log n)$ result first, by re-using parts of the Fredman et al. proof for multipass. We defer this to the full version of the paper. In the remainder of this section we prove the new, stronger result.

▶ **Theorem 18.** *The amortized time of search in a path-balanced BST of size n is* $O\left(\log n \cdot (\log^* n)^2 \cdot 2^{\log^* n}\right)$.

For the purpose of the analysis, we view the path-balance transformation as a sequence of recursive calls on search paths in *some* subtree of T. The total *real* cost is proportional to the original length of the search path to x which we denote by k. We define a threshold $\tau = \log n$, and distinguish between recursive calls on paths shorter than τ ("short paths") and recursive calls on paths longer than τ ("long paths").

A **long path** $\mathcal{P}^x$ is processed as follows. Rotate the median m of the nodes on $\mathcal{P}^x$ to the root, splitting $\mathcal{P}^x$ into two paths of equal lengths. One of these paths contains the path from m to x in $\mathcal{P}^x$, and the other path, which is monotone, contains either the elements smaller than m on $\mathcal{P}^x$ or the elements larger than m on $\mathcal{P}^x$ (depending on whether x is in the right or left subtree of m). In the sequel we assume without loss of generality that the monotone part contains all elements larger than m and denote it by $\mathcal{P}^{>m}$. Let Q^x denote the other (non-monotone) path that ends with x. We perform a multipass transformation on $\mathcal{P}^{>m}$, and make a recursive call on Q^x (i.e., Q^x becomes the P^x of the next recursive call).

A **short path** $\mathcal{P}^x$ is transformed into a balanced binary tree in two phases, as follows. In the first phase, rotate up the median m_1 of $\mathcal{P}^x = \mathcal{P}^1$ until it becomes the root of the subtree rooted at the shallowest node of $\mathcal{P}^1$. This decomposes $\mathcal{P}^1$ into a monotone path and a general path $\mathcal{P}^2$, one starting at the left child of m_1 and the other at the right child of m_1. We repeat this recursively with the median m_2 of $\mathcal{P}^2$, and so on, until we get a general path $\mathcal{P}^\ell$ of length 1. After this transformation, the medians m_j form a path, each m_j having the next median m_{j+1} as one child and a monotone path as the other child. The lengths of these monotone paths decrease exponentially by a factor of 2. In the second phase we apply a multipass transformation on each monotone path, obtaining a complete balanced tree.

Before we analyse each case, we argue that Theorem 16 also holds with a modified potential Φ (defined below). As we only use the new potential from now on, there is no risk of confusion. The modification consists in changing the exponent of the logarithmic term in the denominator from 2 to 3, and changing the additive constant inside the $\log(\cdot)$ to make sure Φ is still increasing everywhere.

Formally, $\Phi = \sum_{x \in T} \phi(x)$, where $\phi(x) = \frac{H(x)}{\log^3(4+H(x))}$, and $H(x) = \log \frac{s(p(x))}{s(x)}$. As earlier, $s(x)$ is the size of the subtree rooted at x, and $p(x)$ is the parent of x.

It can be shown that the entire analysis in § 2 extends to this new potential. Therefore, Theorem 16 holds also for the modified potential function Φ. Now, the analysis of transforming long paths is straightforward. For short paths, we need two new observations.

▶ **Lemma 19.** *The total increase in potential for performing multipass transformation on a path $\mathcal{P}$ of length $k < \log n$ where n is the size of the subtree of the root of $\mathcal{P}$, is at most*

$$\sum_{j=1}^{\log k} \frac{O(\log n)}{(\log\log n + 1 - j)^3}.$$

The proof is identical to that of Lemma 6. As before, the sum can be bounded as $O(\log n)$, but here we use the quantity explicitly inside another sum where the exponent 3 in the denominator will be crucial. The next observation can be shown in a way similar to Lemma 17.

▶ **Lemma 20** (⋆). *Given a search path $\mathcal{P}$ of length $k < \log n$, the total increase in Φ due to recursively rotating all medians $m_1, m_2, \ldots$ of $\mathcal{P}$ to the root is $O(\log n)$.*

We are ready to prove Theorem 18. We split the proof into three cases according to the length of the search path, denoted by k.

Short paths ($k \leq \tau = \log n$). Notice that $\log k \leq \log\log n$. Recall that in the first phase, we repeatedly rotate up the medians, decomposing the path into monotone paths of lengths $1, 2, 4, \ldots, 2^j$, where $j < \log\log n$. By Lemma 20 the total increase in potential due to this transformation is at most $O(\log n)$.

In the second phase, we do a multipass transformation on each of these monotone paths. By Lemma 19, a multipass transformation on a monotone path of length 2^j increases Φ by at most $\sum_{i=1}^{j} \alpha \cdot \log n/(\log\log n + 1 - i)^3$, for some fixed α. Thus, the $j < \log\log n$ multipass transformations increase the potential by at most

$$\sum_{j=1}^{\log\log n} \sum_{i=1}^{j} \frac{\alpha \cdot \log n}{(\log\log n + 1 - i)^3} = \sum_{s=1}^{\log\log n} \frac{\alpha \cdot \log n}{s^2} = O(\log n).$$

The first equality holds since the term $\frac{\alpha \cdot \log n}{s^3}$ appears in the above sum exactly s times $(1 \leq s \leq \log\log n)$. Thus, the total increase in Φ is, in this case, $O(\log n)$.

Longish paths ($\tau < k \leq \log n \cdot \log^* n \cdot 2^{\log^* n}$). Notice that $\log k \leq \log\log n + 2 \cdot \log^* n$.

We perform $2 \cdot \log^* n$ recursive calls and a final call on a search path of length $k' \leq \tau$. The final call increases Φ by at most $O(\log n)$, by the analysis in the previous case. The recursive calls consist of rotating the current median up to the root and applying the multipass transformation on a monotone path. As before, rotating the median up increases Φ by at most $O(\log n)$. Also, each multipass transformation is performed on a path of length $\leq \log n \cdot \log^* n \cdot 2^{\log^* n}$. By Theorem 16, the increase in potential is at most $O(\log n \cdot \log^* n)$. Therefore, the $2 \cdot \log^* n$ recursive calls increase Φ by at most $O\left(\log n \cdot (\log^* n)^2\right)$, which also bounds the total increase in Φ.

Long paths ($k = \Omega\left(\log n \cdot \log^* n \cdot 2^{\log^* n}\right)$). We look at the potential change due to the first recursive call. Again, rotating the median m to the root increases Φ by at most $O(\log n)$. The path splits into $\mathcal{P}^{>m}$ and Q^x, of which $\mathcal{P}^{>m}$ is monotone. By Theorem 16, the multipass transformation on $\mathcal{P}^{>m}$ *decreases* Φ by $\frac{k/2}{2^{\log^*(n)}} - O\left(\log^*(n) \cdot \log n\right)$.

By the same argument, Φ decreases during all of the subsequent recursive calls on paths of size $\Omega\left(\log n \cdot \log^* n \cdot 2^{\log^* n}\right)$.

We continue until we have a recursive call on a path of size at most $\left(\log n \cdot \log^* n \cdot 2^{\log^* n}\right)$, which, by the previous case, increases Φ by at most $O\left(\log n \cdot (\log^* n)^2\right)$. Thus, we obtain that the total decrease in Φ in this case is at least $\frac{k/2}{2^{\log^*(n)}} - O\left(\log n \cdot (\log^* n)^2\right)$.

Combining the three cases, after scaling the potential by $2 \cdot 2^{\log^* n}$, we conclude that the amortized time of the access is $k + 2 \cdot 2^{\log^* n} \cdot \Delta\Phi = O\left(\log n \cdot 2^{\log^* n} \cdot (\log^* n)^2\right)$, as required.

References

1 R. Balasubramanian and Venkatesh Raman. Path balance heuristic for self-adjusting binary search trees. In *Proceedings of FSTTCS*, pages 338–348, 1995. doi:10.1007/3-540-60692-0_59.

2 Parinya Chalermsook, Mayank Goswami, László Kozma, Kurt Mehlhorn, and Thatchaphol Saranurak. Self-adjusting binary search trees: What makes them tick? In *ESA 2015*, pages 300–312, 2015. `doi:10.1007/978-3-662-48350-3_26`.

3 Parinya Chalermsook, Mayank Goswami, László Kozma, Kurt Mehlhorn, and Thatchaphol Saranurak. The landscape of bounds for binary search trees. *CoRR*, abs/1603.04892, 2016. `arXiv:1603.04892`.

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

5 Michael L. Fredman, Robert Sedgewick, Daniel Dominic Sleator, and Robert Endre Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986. `doi:10.1007/BF01840439`.

6 Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. In *25th Annual Symposium on Foundations of Computer Science, West Palm Beach, Florida, USA, 24-26 October 1984*, pages 338–346, 1984. `doi:10.1109/SFCS.1984.715934`.

7 George F. Georgakopoulos and David J. McClurkin. Generalized template splay: A basic theory and calculus. *Comput. J.*, 47(1):10–19, 2004. `doi:10.1093/comjnl/47.1.10`.

8 John Iacono. Improved upper bounds for pairing heaps. In *Algorithm Theory - SWAT 2000, 7th Scandinavian Workshop on Algorithm Theory, Bergen, Norway, July 5-7, 2000, Proceedings*, pages 32–45, 2000. `doi:10.1007/3-540-44985-X_5`.

9 John Iacono. In pursuit of the dynamic optimality conjecture. In *Space-Efficient Data Structures, Streams, and Algorithms*, volume 8066 of *Lecture Notes in Computer Science*, pages 236–250. Springer Berlin Heidelberg, 2013. `doi:10.1007/978-3-642-40273-9_16`.

10 Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

11 Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

12 Daniel H. Larkin, Siddhartha Sen, and Robert Endre Tarjan. A back-to-basics empirical study of priority queues. In *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2014, Portland, Oregon, USA, January 5, 2014*, pages 61–72, 2014. `doi:10.1137/1.9781611973198.7`.

13 Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*, volume 1 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1984. `doi:10.1007/978-3-642-69672-5`.

14 Bernard M. E. Moret and Henry D. Shapiro. *An empirical analysis of algorithms for constructing a minimum spanning tree*, pages 400–411. Springer Berlin Heidelberg, Berlin, Heidelberg, 1991. `doi:10.1007/BFb0028279`.

15 Seth Pettie. Towards a final analysis of pairing heaps. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005), 23-25 October 2005, Pittsburgh, PA, USA, Proceedings*, pages 174–183, 2005. `doi:10.1109/SFCS.2005.75`.

16 Daniel D. Sleator, William P. Thurston, and Robert Endre Tarjan. Rotation distance,triangulations,and hyperbolic geometry. Technical Report CS-TR-131-88, Princeton University (NJ US), 1988. URL: `http://opac.inria.fr/record=b1019357`.

17 Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985. `doi:10.1145/3828.3835`.

18 John T. Stasko and Jeffrey Scott Vitter. Pairing heaps: Experiments and analysis. *Commun. ACM*, 30(3):234–249, 1987. `doi:10.1145/214748.214759`.

19 Ashok Subramanian. An explanation of splaying. *J. Algorithms*, 20(3):512–525, 1996. `doi:10.1006/jagm.1996.0025`.