Smooth heaps and a dual view of self-adjusting data structures[*]

László Kozma [†]   Thatchaphol Saranurak [‡]

Abstract

We present a new connection between self-adjusting binary search trees (BSTs) and heaps, two fundamental, extensively studied, and practically relevant families of data structures (Allen, Munro, 1978; Sleator, Tarjan, 1983; Fredman, Sedgewick, Sleator, Tarjan, 1986; Wilber, 1989; Fredman, 1999; Iacono, Özkan, 2014). Roughly speaking, we map an *arbitrary* heap algorithm within a natural model, to a corresponding BST algorithm with the same cost on a *dual* sequence of operations (i.e. the same sequence with the roles of *time* and *key-space* switched). This is the first general transformation between the two families of data structures.

There is a rich theory of dynamic optimality for BSTs (i.e. the theory of competitiveness between BST algorithms). The lack of an analogous theory for heaps has been noted in the literature (e.g. Pettie; 2005, 2008). Through our connection, we transfer all instance-specific lower bounds known for BSTs to a general model of heaps, initiating a theory of *dynamic optimality for heaps.*

On the algorithmic side, we obtain a new, simple and efficient heap algorithm, which we call the *smooth heap.* We show the smooth heap to be the heap-counterpart of Greedy, the BST algorithm with the strongest proven and conjectured properties from the literature, widely believed to be instance-optimal (Lucas, 1988; Munro, 2000; Demaine et al., 2009). Assuming the optimality of Greedy, the smooth heap is also optimal within our model of heap algorithms. As corollaries of results known for Greedy, we obtain instance-specific upper bounds for the smooth heap, with applications in adaptive sorting.

Intriguingly, the smooth heap, although derived from a non-practical BST algorithm, is simple and easy to implement (e.g. it stores no auxiliary data besides the keys and tree pointers). It can be seen as a variation on the popular *pairing heap* data structure, extending it with a "power-of-two-choices" type of heuristic.

The paper is dedicated to *Raimund Seidel* on the occasion of his sixtieth birthday.

Contents

1 Introduction

We revisit two popular families of comparison-based data structures that implement the fundamental *dictionary* and *priority queue* abstract data types.

A dictionary stores a set of keys from an ordered universe, supporting the operations *search*, *insert*, and *delete*. Other operations required in some applications include finding the smallest or the largest key in a dictionary, finding the successor or predecessor of a key, merging two dictionaries, or listing the keys of a dictionary in sorted order.

A priority queue stores a set of keys from an ordered universe (often referred to as priorities), with typical operations such as finding and deleting the minimum key ("extract-min"), inserting a new key, decreasing a given key, and merging two priority queues. The fact that efficient search is *not* required allows priority queues to perform some operations faster than dictionaries.

Perhaps the best known and most natural implementations of dictionaries and priority queues are binary search trees (BSTs), respectively, multiway heaps. In both cases, a key is associated[1] to each node of the underlying tree structure, according to the standard in-order, respectively, (min-)heap order conditions. Both structures need an occasional re-structuring in order to remain efficient over the long term. In BSTs, the costs of the main operations are proportional to the depths of the affected nodes, the goal is therefore to keep the tree reasonably balanced. For heaps, the desired structure is, in some sense, the opposite; the crucial extract-min operations are easiest to perform if the heap is fully sorted, i.e. if it is a path. The design and analysis of efficient algorithms for heap- and BST-maintenance continues to be a central topic of algorithmic research, and a large number of ingenious techniques have been proposed in the literature.

Our contribution is a new connection between the two tasks, showing that, with some reasonable restrictions, an arbitrary algorithm for heap-maintenance encodes an algorithm for BST-maintenance on a related (dual) sequence of inputs. This is the first general transformation between the two families of data structures. The connection allows us to transfer results (both algorithms and lower bounds) between the two settings.

On the algorithmic side, we obtain a new, simple and efficient heap data structure, which we call the *smooth heap*. We show the smooth heap to be the heap-equivalent of Greedy, the BST algorithm with the strongest proven and conjectured properties from the literature. We thus obtain, for the smooth heap, instance-specific guarantees (upper bounds) that go beyond the logarithmic amortized guarantees known for existing heap implementations. Key to our result is a new, non-deterministic interpretation of Greedy, which may be of independent interest.

On the complexity side, we transfer all instance-specific lower bounds known for BSTs to a natural class of heap algorithms. Pettie observed [Pet05, Pet08] that a theory of dynamic optimality for heaps, analogous to the rich set of results known for BSTs, is missing. We see the instance-specific lower and upper bounds that we transfer to heaps as essential components of such a theory.

Self-adjusting BSTs. The study of dictionaries based on BSTs goes back to the beginnings of computer science.[2] Standard balanced trees such as AVL-trees [AVL62], red-black trees [Bay72, GS78],

---

[1]Typically, a data record is also stored with each key, whose retrieval is the main purpose of the data structure; we ignore this aspect throughout the paper, as it does not affect the data structuring mechanisms we study.

[2]Arguably, as an abstraction of searching in ordered sets, binary search trees have been studied (explicitly or implicitly) well before the existence of computers, see e.g. methods for root-finding in mathematical analysis.

or randomized treaps [SA96] keep the tree balanced by storing and enforcing explicit bookkeeping information. We refer to Knuth [Knu98, § 6.2] for an extensive overview of classical methods.

By contrast, splay trees, invented by Sleator and Tarjan [ST85], do not store any auxiliary information (either in the nodes of the tree or globally) besides the keys of the dictionary and the pointers representing the tree. Instead, they employ an elegant local re-structuring whenever a key is accessed. In short, splay trees bring the accessed key to the root via a sequence of *double-rotations*. Splay trees can be seen as a member of a more general family of *self-adjusting* data structures.[3] Splay trees support all main operations in (optimal) logarithmic amortized time. Furthermore, they adapt to various usage patterns, providing stronger than logarithmic guarantees for certain structured sequences of operations; we call such guarantees *instance-specific*. Several instance-specific upper bounds are known for splay trees (see e.g. [ST85, Tar85, Sun92, CMSS00, Col00, Elm04, Pet08]) and others have been conjectured. In fact, Sleator and Tarjan conjectured splay trees to be *instance-optimal*, i.e. to match, up to a constant factor, the optimum among all BST algorithms.[4] This is the famous *dynamic optimality conjecture*, still open. It is not known whether *any* polynomial-time BST algorithm has such a property, even with a priori knowledge of the entire sequence of operations.

Lucas [Luc88] proposed, as a theoretical construction, a BST algorithm that takes into account *future* operations; in her algorithm, the search path is re-arranged such as to bring the soonest-to-be-accessed key to the root (and similarly in a recursive way in the left and right subtrees of the root). Following Demaine, Harmon, Iacono, Kane, and Pătraşcu [DHI$^+$09], we refer to this algorithm as GreedyFuture. GreedyFuture is widely believed to be $O(1)$-competitive [Luc88, Mun00, DHI$^+$09], and is known to match essentially all instance-specific guarantees proven for splay trees, as well as some stronger guarantees not known to hold for splay trees [CGK$^+$15b, IL16]. Unfortunately, $o(\log n)$-competitiveness has not been shown for either Splay or GreedyFuture.

It was shown in [DHI$^+$09] that GreedyFuture can be simulated by an *online* algorithm (which we refer to simply as Greedy), with only a constant factor increase in cost. In a different line of work, Demaine et al. described an online BST algorithm with a competitive ratio of $O(\log \log n)$, called Tango tree [DHIP07]. Strictly speaking, neither Greedy (in the online version), nor Tango are self-adjusting, as they both store auxiliary information in the nodes of the tree. Furthermore, due to their rather intricate details, neither can be considered a practical alternative to splay trees. It would be of great interest to find simple and practical BST algorithms that match the theoretical guarantees of Greedy or Tango.

Self-adjusting heaps.   In 1984, Fredman and Tarjan introduced the Fibonacci heap [FT87], a data structure that achieves the theoretically optimal amortized bounds of $O(\log n)$ for extract-min, and $O(1)$ for all other heap operations. Fibonacci heaps are rather complicated to implement and simpler alternatives tend to outperform them in practice [LST14]. In the past three decades, a central goal of research in data structures has thus been to find a simpler heap implementation that would match the theoretical guarantees of Fibonacci heaps (see e.g. [SV87, DGST88, Tak03, KT08, Elm10, HST11, BLT12, Bro13, Cha13, HKTZ15]).

Arguably, this goal has not been fully achieved. In particular, Fibonacci heaps, as well as

---

[3]Two simpler heuristics proposed earlier by Allen and Munro [AM78] also fall into this category, but are known to be inefficient on certain inputs.

[4]The term "instance-optimal" is, in our context, equivalent with "$O(1)$-competitive". An algorithm is c-competitive, if its running time is at most c times the running time of the optimal algorithm (for every given input).

most of their proposed alternatives store auxiliary bookkeeping information, or perform operations outside the comparison/pointer model, or maintain a pointer structure that is not "forest-like" (not following, as Iacono and Özkan [IÖ14b] put it, *"the spirit of what we usually think of as a heap".*) The following general question is still open.

Question 1. *Is there, by analogy to search trees, a simple, self-adjusting heap data structure with optimal amortized cost, possibly achieving instance-optimality?*

The closest in simplicity and elegance to self-adjusting trees are *pairing heaps*, introduced by Fredman, Sedgewick, Sleator, and Tarjan [FSST86]. Pairing heaps do not store auxiliary information besides the keys and the pointers of a single (multiway) heap, are easy to implement and efficient in practice [LST14]. They implement all operations using the unit-cost *link* primitive (Figure 1). Pairing heaps perform extract-min, the only operation with a non-trivial implementation, in a *two-pass re-structuring* of the children of the root. (We describe pairing heaps in more detail in §3.)
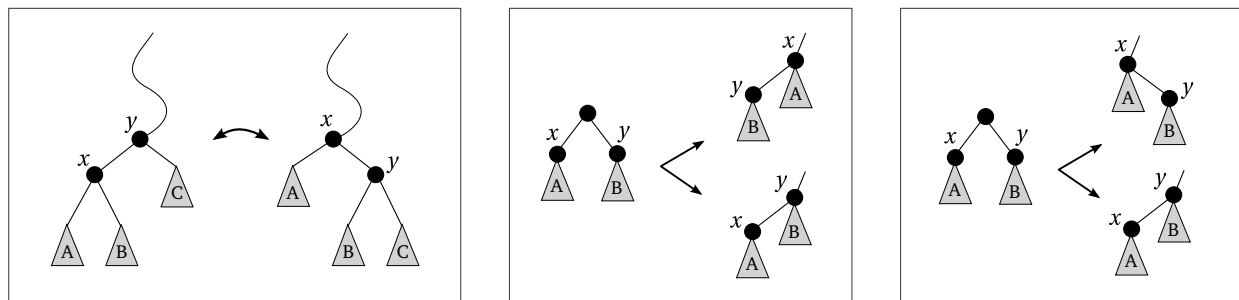

Figure 1: Elementary operations in BSTs and heaps. Letters x and y denote keys, A, B, C are arbitrary subtrees. (i) Rotation in a binary search tree. (ii) Classical ("unstable") link between neighboring siblings in a heap. The larger of x and y becomes the leftmost child of the smaller. (iii) Stable link between neighboring siblings in a heap. The larger of x and y becomes the leftmost child of the smaller, if it was originally to the left; otherwise it becomes the rightmost child.

Fredman et al. showed that pairing heaps support all operations in logarithmic amortized time. They also conjectured that pairing heaps match the optimal bounds of Fibonacci heaps. This conjecture was disproved by Fredman [Fre99a], who showed that in certain sequences, *decrease-key* may cost $\Omega(\log \log n)$. A similar lower bound was later shown by Iacono and Özkan [IÖ14b, IÖ14a].[5]

The Fredman-, and Iacono-Özkan lower bounds hold for broad classes of "pairing-heap-like" data structures that include all natural variants of pairing heaps, as well as some other heap data structures from the literature. The heap models defined in these works are "in the spirit" of self-adjusting trees; to the extent that these models capture the idea of a "self-adjusting heap", the answer to Question 1 has to be negative. We argue, however, that there exist natural self-adjusting heaps that fall outside these models, leaving Question 1 open.

---

[5]We remark in passing that the complexity of the standard paring heap is not fully understood. Iacono [Iac00] showed that assuming $O(\log n)$ for decrease-key, insert takes constant amortized time. Pettie [Pet05] showed that $O(4^{\sqrt{\log \log n}})$ can be simultaneously achieved for both insert and decrease-key. For natural variants of pairing heaps, even the logarithmic cost of extract-min has only been conjectured [FSST86, DKKZ18, DKK+18]. Improving these bounds remains a challenging open problem.

For some concrete heaps, certain instance-specific *upper bounds* were studied by Iacono and Langerman [IL05], Elmasry [Elm06], and Elmasry et al. [EFI12]. On the other hand, a theory of instance-specific *lower bounds* (in a sense similar to BSTs) has not yet been proposed for heaps in *any heap model*. We argue that the existence of such lower bounds is *necessary* for dynamic optimality to be possible (even approximately). Informally speaking, even a candidate instance-optimal heap must have high cost on certain input sequences (by information-theoretic arguments); these input sequences must therefore be hard *for every heap*. Put differently, some restriction on the allowed comparisons and links is necessary, to prevent a hypothetical optimal algorithm from "guessing" the correct permutation of the input.

Connecting BSTs and heaps. Self-adjusting BSTs and heaps serve different purposes, and the underlying trees they maintain have, in general, different characteristics. Yet, they are similar in their approach of performing local re-adjustments (using rotations, respectively links), seemingly ignoring global structure. The two families of data structures have been introduced and studied in the past decades by largely the same authors.

The following technical connection between splay trees and the standard variant of pairing heaps has been observed by Fredman et al. [FSST86]. If we view a heap as a binary tree (interpreting "leftmost-child" and "right-sibling" pointers as "left-child" and "right-child", see e.g. [Knu97, § 2.3.2]), the re-structuring of pairing heaps after an extract-min operation resembles the re-adjustment of splay trees during an access. Using this observation, Fredman et al. adapt the splay tree potential function [ST85] to show that pairing heap operations take logarithmic time.[6] Despite the productive use of this connection, it seems rather specific to splay trees and pairing heaps. Given the intriguing similarity between self-adjusting data structures, the following question suggests itself.

Question 2. Is there a fundamental, general correspondence between self-adjusting BST and self-adjusting heap data structures?

1.1   Our results

We propose a general model of heaps, which we call the *stable heap model*.[7] This model allows us to approach both **Question 1** and **Question 2** in surprising ways. The model aims to capture self-adjusting heap algorithms, and is somewhat similar to the "pure heap" model of Iacono and Özkan [IÖ14b, IÖ14a].

The new element of our model is the *stable link* operation that replaces the *link* operation of existing heap models. Whereas the standard link operation makes the larger of the two linked items the leftmost child of the smaller, our stable link makes the larger item either the leftmost or the rightmost child of the smaller (see Figure 1). More precisely, if x and y are linked, where x is the left neighbor of y, then, a stable link operation makes x the *leftmost* child of y, or it makes y the *rightmost* child of x, respecting the (min-)heap order condition.

---

[6]We remark that Fredman et al.'s analogy between pairing heaps and splay trees is far from trivial; to maintain it, the identities of nodes may need to be permuted, and some left-right children pairs may need to be swapped.

[7]By *model* we understand the description of the pointer-based structure and the set of primitive operations that may be used to implement the priority queue.

An intuitive a priori reason for stable links is that they "better preserve" the original left-to-right ordering of keys, which we expect to make algorithms that use stable links more amenable to instance-specific analysis.[8] A second justification is that the lower bounds known in existing heap models [Fre99a, IÖ14b, IÖ14a] crucially exploit that the links are not stable, raising the possibility that an algorithm that uses stable links may circumvent the existing lower bounds. The main motivation, however, is that with stable links, the heap model turns out to be deeply connected to the BST model.

General transformation. Our connection addresses ***Question 2*** as follows. We show that within the stable heap model, *every* algorithm $\mathcal{A}$ for heap re-structuring corresponds to some algorithm $\mathcal{B}$ for BST re-arrangement. This is the first general connection between *models* of heaps and BSTs. The costs of the two algorithms in *sorting-mode* may differ only by a constant factor, when executing "dual" sequences of operations. By *sorting-mode* execution we mean the following (see Figure 2 for illustration).

Sorting-mode for a heap: A sequence of n keys $(x_1, \ldots, x_n)$ is inserted into an initially empty heap, followed by n extract-min operations. For simplicity, assume that $\{x_1, \ldots, x_n\} = [n]$. The execution can be interpreted as sorting the permutation $(x_1, \ldots, x_n)$ via *selection-sort*, using a particular heap-based method for selecting the minimum.

Sorting-mode for a BST: A sequence of n keys $(x_1, \ldots, x_n)$ is inserted into an originally empty BST. Again, assume $\{x_1, \ldots, x_n\} = [n]$. The execution can be interpreted as sorting $(x_1, \ldots, x_n)$ via *insertion-sort*, using a particular BST-based method for insertion. (The sorted keys can be read out in an $O(n)$-time traversal of the final tree.)

The connection between $\mathcal{A}$ and $\mathcal{B}$ is the following. The number of elementary operations performed by $\mathcal{A}$ when sorting $X = (x_1, \ldots, x_n)$ equals (up to a constant factor) the number of elementary operations performed by $\mathcal{B}$ when sorting the *inverse permutation X'*, i.e. the permutation $X' = (y_1, \ldots, y_n)$, where $y_i = j$ iff $x_j = i$. (See Figure 2 for an illustration.)

We say that the roles of *time* and *key-space* are switched between the two problems. The time of insertion, a.k.a. the *index* of a key in the heap input is mapped to the *rank* of a key in the BST input. Similarly, the *rank* of a key in the heap input is mapped to the time of insertion in the BST input.

We remark that the described duality between heaps and trees is quite subtle. As we execute the heap algorithm, the individual rotations that make up the BST execution are revealed in an order that is neither the temporal, nor the spatial order; the order in which the BST execution is revealed depends on the internal structure of the input. It is instructive to compare our connection with the connection of Fredman et al. between pairing heaps and splay trees. There, individual link operations map (essentially) to individual rotations; the pairing heap and the splay tree evolve in parallel and the correspondence between them is maintained throughout the execution. This is emphatically not the case in our connection. In fact, we do not even map individual operations such as *search*, *extract-min*, or *insert* to each other. Instead, we "globally" match the execution trace of a sequence of operations in a heap with the execution trace in a BST for the "dual" sequence of operations. An intermediate state of one structure, in our connection, corresponds to an *abstract*

---

[8]The name is inspired by the idea of "stable sorting" (see e.g. [Knu98, §5]). Stable sorting preserves the original ordering of key-pairs that are *equal*. As seen later, stable linking preserves the original ordering of key-pairs that are *incomparable*, according to our current knowledge, i.e. in the partial order described by the heap.
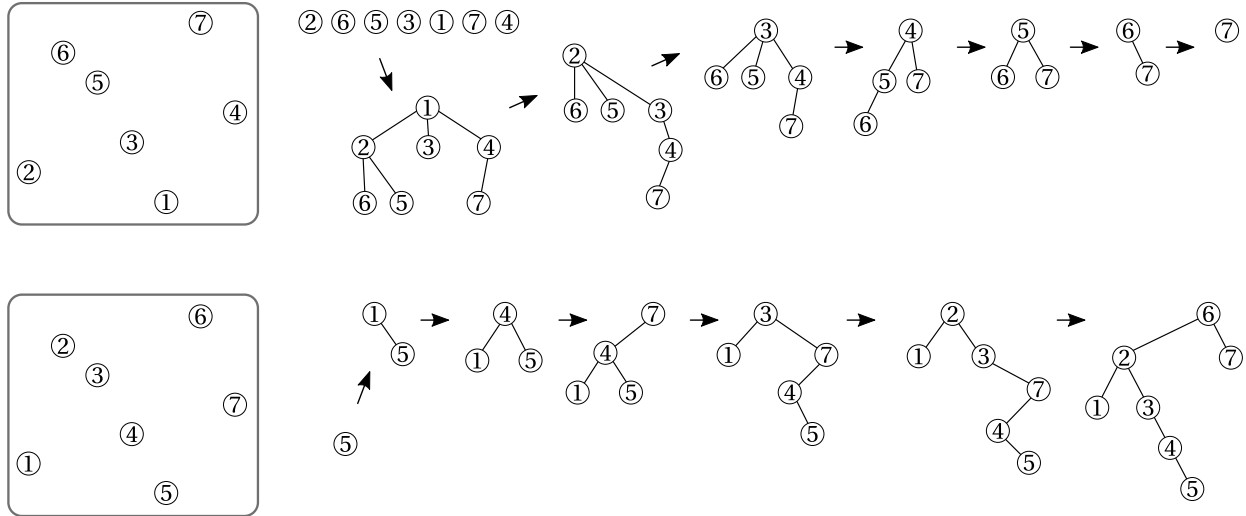
Figure 2: Sorting-mode execution for heaps and BSTs. Input permutation $P = (2, 6, 5, 3, 1, 7, 4)$ and its inverse $P' = (5, 1, 4, 7, 3, 2, 6)$. Above left, P shown with values on y-axis, below left, P' shown with values on x-axis. Above: sequence of extract-mins in a heap, following insertion sequence P, using an unspecified stable heap algorithm. Below, insertion sequence P' in an initially empty BST, using the "move-to-root" restructuring (the inserted items are rotated to the root).

state of the other structure, where some decisions pertaining to future operations have been comitted to, others are still left undecided.

Smooth heaps.   Perhaps the most interesting consequence of our connection is a new, simple heap algorithm that we call the *smooth heap*. We show it to be the heap-counterpart of Greedy, which is conjectured to be instance-optimal for BSTs. Interpreting the proven guarantees for Greedy, we obtain a logarithmic bound on the cost of extract-min in smooth heaps, as well as a number of instance-specific upper bounds, not previously known for any heap implementation (e.g. we show that smooth heaps adapt to locality of reference, and to pattern-avoidance in the input). Assuming the conjectured optimality of Greedy, smooth heaps are also optimal in sorting-mode among all stable heap algorithms. Key to our result is a new, non-deterministic interpretation of Greedy which may be of independent interest.

The intriguing aspect of this connection is that although Greedy is rather complicated to implement as an online BST algorithm, the smooth heap is not; it is comparable in simplicity to pairing heaps. There are *several equivalent descriptions* of the smooth heap. In one view, it appears, roughly speaking, as a pairing heap equipped with a "power-of-two-choices" type of heuristic. In another view, it appears as a structure built of nested treaps. The smooth heap is self-adjusting (no auxiliary information), and uses only the standard pointer structure, thus it partially addresses **Question 1**.

In the following, we briefly describe the smooth heap data structure. (See §4 for more details and variants.) In this section, we view the smooth heap as a single multiway (min-)heap (i.e. the key of each node is greater than its parent). The crucial operation is extract-min, implemented as follows. After deleting the root, we are left with a list of nodes that need to be consolidated into a

single tree, using link operations. We proceed by repeatedly finding a *local maximum* (i.e. an item that is larger than its neighboring siblings). We then link the local maximum with *the larger* of its two neighbors (or with its only neighbor, if it is the leftmost or the rightmost item in the list). This operation, in effect, removes the local maximum, "smoothing" the sequence of items; this gives the name of the data structure. Linking with the larger of the two neighbors can be seen as a locally greedy choice: as we are moving towards the sorted order, intuitively the smaller the rank-difference between linked items, the more progress we make.[9]

A high-level description of the algorithm is given in Figure 3. More explicit descriptions are given in §4 and Appendix D.

Input: a list of siblings X
while there is an item x in X larger than its neighbors:
  let y be the largest of the neighbors of x.
  link x and y.

Figure 3: Smooth heap re-structuring for extract-min.

Finding local maxima and linking them with a neighbor can be done in a single left-to-right pass (similarly to the first pass of pairing heaps). The remaining top-level items after this pass are in sorted order, they can therefore be collected in a second, right-to-left accumulation pass (again, similar to the second pass of pairing heaps, except that here, no more comparisons need to be made). The total number of comparisons is easily seen to be at most twice the number of link operations performed. See Figure 4 for an illustration. We remark that the link operations used in the smooth heap are all stable links.

Figure 4: Smooth heap re-structuring. Top-level siblings, left-to-right with keys $(1, 3, 7, 4, 6, 2, 5, 9, 8)$, key values shown on y-axis, subtrees of nodes not shown. (i) Initial state. Small numbers indicate order of linking. (ii) State after left-to-right smoothing round. Remaining top-level nodes appear hollow. (iii) State after final, right-to-left round. Lowest node is new root after extract-min.

As our focus is on sorting-mode, we postpone the description of other operations to §4. It remains an intriguing open question whether the smooth heap (or any other self-adjusting heap) can match

---

[9]There are limits to this intuition, and we do not claim smooth heaps to be exactly, globally optimal, see Appendix E.

Fibonacci heaps in the asymptotic cost of all operations (i.e. the remainder of **Question 1**). The arguments that rule out constant-time decrease-key in the case of pairing heaps do not immediately apply to stable heaps. It remains to be seen whether this is a mere technicality, or if an efficient decrease-key can in fact be added to smooth heaps. There are several natural ways to implement this operation, but their analysis does not appear easy.

Consequences of the transformation. Several standard heap algorithms can be transferred to the stable heap model, including the variants of pairing heaps described in [FSST86] (see § 3). Through our duality between heap and tree algorithms, these new heap algorithms also yield new *offline* BST algorithms that have not been previously described. (In fact, for every stable heap algorithm we obtain a *family* of offline BST algorithms, with cost at most a constant factor larger; e.g. we obtain a set of (non-trivially) different BST executions that are $O(1)$-competitive with Greedy.) In the other direction, we can transfer strong instance-specific *lower bounds* [Wil89, DHI+09] from BSTs to heaps. We thus obtain instance-specific lower bounds for all stable heap algorithms in sorting-mode, which we see as a necessary step towards a theory of dynamic optimality for heaps.[10]

Adaptive sorting. Our connection between self-adjusting BSTs and heaps can be interpreted as an equivalence between broad classes of selection-sort and insertion-sort algorithms on inverted input permutations. Sorting has been extensively studied, not just in the worst-case, but also in an instance-specific sense. (Sorting with some "pre-sortedness" in the input is often called "adaptive sorting"). As an entry point into the literature on adaptive sorting, we refer to [MS76, Meh79, Man84, LP89, LP94, ECW92, MP92, BN13], [Knu98, § 5.3], and references therein.

The proven instance-specific properties of Splay and GreedyFuture subsume many of the structural properties in the adaptive sorting literature (Appendix A). Until now, however, there has been no easy way to implement GreedyFuture as an insertion-sort algorithm. (As mentioned, the online Greedy algorithm is rather complicated.) By using smooth heaps, we obtain an easy-to-implement selection-sort algorithm[11] that inherits all the instance-specific properties of Greedy (on the inverse of the input permutation).

As a concrete example, we mention sorting sequences with an avoided permutation pattern (pattern-avoidance is a well-studied combinatorial property of sequences, see e.g. [Knu68, Tar72, Pra73, Kit11, GM14, NRRS17]). Sorting pattern-avoiding permutations was studied by Arthur [Art07]. As a corollary of our pattern-avoiding bounds for Greedy [CGK+15b], we obtain a practical sorting algorithm that sorts permutations with an arbitrary fixed avoided pattern in quasi-linear time (the exact bound involves the inverse Ackerman function), improving the earlier bound of [Art07]. Furthermore, in contrast to [Art07], our algorithm need not know the avoided pattern in advance; pattern-avoidance of the input is used only in the analysis of Greedy [CGK+15b]. (We use here the easy fact that if a permutation avoids a pattern, then the inverse permutation also avoids a certain pattern of the same size.)

---

[10]The lack of such a theory was noted e.g. by Pettie [Pet05, Pet08]: *"Despite the connection between splay trees and pairing heaps, there is no obvious analogue of dynamic optimality for priority queues."*

[11]We remark that *every* stable heap algorithm yields a *stable sorting* algorithm in the classical sense, assuming that ties are broken according to the "left is smaller" rule.

1.2 Related work and structure of the paper

A "dual" view of binary search trees and heaps brings to mind the Cartesian tree data structure [Vui80], also known as treap [SA96]. A treap is a binary tree built over pairs of values, respecting the in-order condition with respect to the first, and the heap-order condition with respect to the second entry. Assuming distinct priorities, the treap is easily seen to be unique. Swapping the roles of the two entries yields a "dual treap".

The operation performed by smooth heaps on a list of items can be seen as a linear-time "on-the-fly treapification", where the left-to-right indices of items play the role of key-values. Such a description of the smooth heap is reminiscent of the Cartesian tree sorting algorithm of Levcopoulos and Petersson [LP89]. There too, a treap of the input permutation is built. Afterwards, a secondary heap structure (e.g. a binary heap or a Fibonacci-heap) is used to store the children of the root; in this secondary heap, the minimum is repeatedly deleted and its children inserted, resulting in a sorted deletion-order. Cartesian tree sorting was not proposed as a general-purpose data structure, but it could be turned into one, by implementing insert and other operations. (A natural way to insert is perhaps to add a new key into the auxiliary heap.)

Despite the apparent similarity, the behavior and performance of smooth heaps and Cartesian tree sorting are different. Smooth heaps do not require a secondary heap structure, instead, they (implicitly) build a new treap from the top-level list of nodes after every deletion of the minimum. The running time of Cartesian tree sorting can be expressed in closed form. It was shown by Eppstein that this quantity is subsumed by known instance-specific bounds for splay trees [Epp]. It is easy to construct examples where the running time of sorting by smooth heaps is $O(n)$, whereas the running time of Cartesian tree sorting is $\Omega(n \log n)$. An example where Cartesian tree sorting is faster than sorting by smooth heaps would disprove the conjectured dynamic optimality of Greedy (Appendix B).

In §2 we define the self-adjusting BST model that we use in the rest of the paper. In §3 we describe the stable heap model and we adapt known algorithms to this model. In §4 we describe the smooth heap, our new heap data structure. In §5 we present the main connection between the two models, and describe some of its immediate consequences. As a key ingredient of our results, we present a nondeterministic description of Greedy in §6. We then give the full proofs in §7. In §8 we conclude with an extensive list of open questions and possible further directions.

2 BST model

We adopt a standard model of binary search trees (BSTs), see e.g. [Wil89, Luc88, DHIP07, DHI+09], which can be seen as a restricted pointer machine model. Each node of the BST is associated with a key, respecting the in-order condition (each item is larger than those in its left subtree and smaller than those in its right subtree). Dictionary operations are performed using a cursor that points, at the beginning of every operation, to the root of the tree. The elementary (unit-cost) operations allowed are moving the cursor to the parent, left-child, or right-child of the current node, and performing a rotation on the edge between the node at the cursor and its parent (Figure 1).

In this paper we consider only search and insert operations; we further assume that all searches are successful. Throughout, we assume that all key-values are distinct.[12]

---

[12]See [DHI+09, CGK+15b] for discussion of these standard assumptions.

Search and insert are implemented in the standard way. For search, it is required that the cursor visits, at some point during the operation, the node with the given key. For insert, the cursor must visit both the predecessor and the successor of the new key (or only one of them, in case the key is the new minimum or maximum). A node containing the new key is then attached as the right child of its predecessor, or as the left child of its successor (whichever slot is free). In addition, during both search and insert, a BST algorithm may perform arbitrary re-arrangement of the tree, using rotations at the cursor. The cost of a search or insert operation is the number of nodes that are *visited* by the cursor ("touched") during the operation. Observe that the touched nodes form a subtree containing the root. This cost model is easily seen to be equivalent up-to constant factors, with most other reasonable cost models [Luc88, Wil89, DHI+09].

In this paper we only consider "offline" BST algorithms, i.e. we assume that the entire sequence of insert and search operations is known in advance.[13] We mainly consider two possible modes of operation: a *search-only mode*, in which, starting from some initial tree a sequence of n distinct searches are performed, and an *insert-only mode*, in which, starting from an empty tree, a sequence of n distinct keys are inserted. (We also call the insert-only mode the *sorting-mode*, as it can be seen as a particular implementation of insertion-sort; the n keys can be read out in sorted order by the in-order traversal of the resulting tree.)

As the operations are performed on distinct keys (in both search-only and insert-only modes), we view a sequence of operations of length n as a permutation over $[n]$. The *execution trace* of a BST algorithm $\mathcal{A}$ serving the sequence X, denoted $\mathcal{A}(X)$, is the set of touched keys for all operations. That is, $\langle i, j \rangle \in \mathcal{A}(X)$, if algorithm $\mathcal{A}$ touches node i when executing the j-th operation of X ("at time j"). The cost of the execution is $|\mathcal{A}(X)|$, i.e. the total number of nodes touched at all times. For every permutation X over $[n]$, we denote the *offline optimum* in search-only mode and insert-only mode as $\mathsf{OPT}_{\mathsf{bst}}(X)$ and $\mathsf{OPT}_{\mathsf{ibst}}(X)$ respectively, where $\mathsf{OPT}_{\mathsf{bst}}(X) = \min_{\mathcal{A}}\{|\mathcal{A}(X)|\}$ when X is treated as a sequence of search operations, and similarly $\mathsf{OPT}_{\mathsf{ibst}}(X) = \min_{\mathcal{A}}\{|\mathcal{A}(X)|\}$ when X is treated as a sequence of insert operations.

Demaine et al. [DHI+09] introduced an elegant geometric view for the study of BST algorithms, which we use throughout the paper. In the remainder of the section we review some of their definitions and results, and we slightly extend the model, to handle insertion-sequences.

Satisfied point sets. Let $P \subseteq \mathbb{Z} \times \mathbb{Z}$ be a finite set of points. For every point $p \in P$, we write $p = \langle p.x, p.y \rangle$ where $p.x$ is the x-coordinate of p (the first coordinate) and $p.y$ is the y-coordinate of p (the second coordinate). We denote $P_{x \leq i} = \{p \in P \mid p.x \leq i\}$ and $P_{y \leq i} = \{p \in P \mid p.y \leq i\}$, and similarly, $P_{x=i}$, $P_{x<i}$, and so on. We call $P_{x=i}$ and $P_{y=i}$ the i-th column and the i-th row of P respectively. We say that P is a *permutation* (of size n) if $|P_{x=i}| = |P_{y=i}| = 1$ iff $i \in [n]$.

Given two points $p, q \in P$, we denote as $\square_{pq} \subseteq \mathbb{Z} \times \mathbb{Z}$ the rectangle (possibly degenerate) whose opposite corners are p and q. We say that $\square_{pq}$ is a *satisfied rectangle* if $\square_{pq}$ contains another point $r \in P \setminus \{p, q\}$ (where r can be at a corner or a border of $\square_{pq}$), or if p and q are aligned vertically or horizontally.

A point set P is *satisfied* if, for every pair of points $p, q \in P$, the rectangle $\square_{pq}$ is satisfied.

Definition 2.1 (Satisfied superset problem)**.** *Given a set of points P, find a satisfied set $Q \supseteq P$.*

---

[13]For online algorithms, operations are revealed one by one.

Let $\mathcal{A}_{\mathtt{sat}}$ be some algorithm for solving the satisfied superset problem. Given a point set P, we denote by $\mathcal{A}_{\mathtt{sat}}(P)$ the output of $\mathcal{A}_{\mathtt{sat}}$ for input P, i.e. a satisfied superset of P. The *cost* of $\mathcal{A}_{\mathtt{sat}}$ on P is defined to be $|\mathcal{A}_{\mathtt{sat}}(P)|$. Let $\mathsf{OPT}_{\mathtt{sat}}(P)$ be the size of the smallest satisfied superset of P.

The following definition is new.

Definition 2.2 (Insertion-compatible superset). *Given a set of points P, a superset $Q \supseteq P$ is* insertion-compatible, *if for every $i \in [n]$ we have* $\min\{p.y \mid p \in P_{x=i}\} = \min\{q.y \mid q \in Q_{x=i}\}$. *In words, Q does not add a point* below *a point of P.*

We also consider the variant of the satisfied superset problem where the goal is to find an *insertion-compatible* satisfied superset of the input. For every point set P, let $\mathsf{OPT}_{\mathtt{isat}}(P)$ be the size of the smallest insertion-compatible satisfied superset of P. Clearly, $\mathsf{OPT}_{\mathtt{isat}}(P) \geq \mathsf{OPT}_{\mathtt{sat}}(P)$.

Let $X = (x_1, \ldots, x_n) \in [n]^n$ be a permutation, i.e. $x_i \neq x_j$ for every $i \neq j$. We call $P^X = \{\langle x_i, i \rangle \mid i \in [n]\}$ the *point set of X*.

Given X, we also consider the *inverse* permutation X'. (The i-th element of X is x_i iff the x_i-th element of X' is i.) Treating P^X as an n-by-n matrix, we can define the transpose $(P^X)'$. Observe that $(P^X)' = P^{X'}$. We also consider the *reverse* permutation X^r. (The i-th element of X^r is x_{n-i+1}.) For a point set $P \subseteq [n] \times [n]$, its reverse P^r is such that $P^r_{y=i} = \{\langle x, i \rangle \mid \langle x, n-i+1 \rangle \in P_{y=n-i+1}\}$ for each i. Note that $(P^X)^r = (P^{X^r})$. Throughout this paper, we usually use X to denote a sequence of numbers, and use P or Q to denote a point set.

Demaine et al. [DHI+09] show a precise equivalence between execution traces of BST algorithms and satisfied sets.

Theorem 2.3 (Geometry of BSTs, search-only [DHI+09]). *Let $X \in [n]^n$ be an arbitrary permutation. A set $Q \subset [n]^2$ is a satisfied superset of P^X iff there is an offline BST algorithm $\mathcal{A}_{\mathtt{bst}}$ in search-only mode with some initial tree such that $\mathcal{A}_{\mathtt{bst}}(X) = Q$.*

Corollary 2.4. *For every permutation $X \in [n]^n$, $\mathsf{OPT}_{\mathtt{bst}}(X) = \mathsf{OPT}_{\mathtt{bst}}(X') = \mathsf{OPT}_{\mathtt{bst}}(X^r)$.*

Proof. By Theorem 2.3, $\mathsf{OPT}_{\mathtt{bst}}(X) = \mathsf{OPT}_{\mathtt{sat}}(P^X)$. It is clear that $\mathsf{OPT}_{\mathtt{sat}}(P^X) = \mathsf{OPT}_{\mathtt{sat}}(P^{X'}) = \mathsf{OPT}_{\mathtt{sat}}(P^{X^r})$. Another application of Theorem 2.3 implies the claim. ∎

The following theorem is new. Its proof closely follows the proof of Theorem 2.3 from [DHI+09]. For completeness, we give it in Appendix C.

Theorem 2.5 (Geometry of BSTs, insertion-compatible). *Let $X \in [n]^n$ be an arbitrary permutation. A set $Q \subset [n]^2$ is an insertion-compatible satisfied superset of P^X iff there is an offline BST algorithm $\mathcal{A}_{\mathtt{bst}}$ in insert-only mode such that $\mathcal{A}_{\mathtt{bst}}(X) = Q$.*

Remark 2.6. By Theorems 2.3 and 2.5, given any BST algorithm $\mathcal{A}$ in insert-only mode (a.k.a. sorting-mode), there is a BST algortihm $\mathcal{A}'$ in search-only mode, such that $|\mathcal{A}'(X)| \leq |\mathcal{A}(X)|$ holds for all permutations X.[14] That is, an upper bound for the insert-only mode is also an upper bound for the search-only mode. It is not known whether the reverse statement holds. In this paper we always prove a stronger statement by upper bounding the cost of BST algorithms in insert-only mode.

---

[14] With the same reasoning, we can even show that there is a BST algorithm $\mathcal{A}''$ that handles a sequence of arbitrarily intermixed search and insert operations, where $|\mathcal{A}''(X)| \leq |\mathcal{A}(X)|$ for every permutation X.

A BST algorithm $\mathcal{A}$ is c-competitive (w.r. to the offline optimal BST algorithm) if for all permutations $X \in [n]^n$, we have $|\mathcal{A}(X)| \le c \cdot \mathsf{OPT}_{\mathtt{bst}}(X)$. If $\mathcal{A}$ is $O(1)$-competitive, we also say that $\mathcal{A}$ is *instance-optimal.*

GreedyFuture and Greedy. Perhaps the most natural algorithm for the satisfied superset problem is Greedy, a straightforward geometric sweepline algorithm.

Definition 2.7 (Greedy [DHI$^+$09]). *Given a set $P \subseteq [n] \times [n]$, Greedy works in n steps as follows. Initially, $Q = P$. At the i-th step, if there is a point $p \in P_{y=i}$ and $q \in Q_{y<i}$ where $\square_{pq}$ is not satisfied, it adds a point $r = \langle q.x, i \rangle$ to Q (so now $\square_{pq}$ is satisfied). The output of Greedy is denoted Greedy$(P) = Q$.*

Observe that Greedy always produces an insertion-compatible satisfied superset. It is known that Greedy is equivalent, in the sense of Theorems 2.3 and 2.5, to the offline BST algorithm GreedyFuture.

Theorem 2.8 ([DHI$^+$09]). *For all permutations X it holds that GreedyFuture$(X) =$ Greedy(P^X).*

Similarly to Splay [ST85], GreedyFuture is conjectured to be instance-optimal.

Conjecture 2.9 ([DHI$^+$09, Mun00, Luc88]). *For all permutations $X \in [n]^n$ it holds that GreedyFuture$(X) = O(\mathsf{OPT}_{\mathtt{bst}}(X))$.*

Demaine et al. also define *online* satisfied superset algorithms and show a stronger form of Theorem 2.3 that applies to online algorithms. In this way, they obtain an online BST algorithm competitive with GreedyFuture. This online equivalence is not our concern in the current paper.

3 Stable heap model

In this section we introduce a family of heap data structures that include close variants of several of the previously proposed heaps. The key feature of our model is the stable link operation that replaces the standard (unstable) link (Figure 1). We call the new model the stable heap model.

3.1 Description of the model

Structure. A heap in the stable heap model is organized as a *forest* of multiway min-heaps.[15] Every node has an associated key (we refer interchangeably to a node and its key), and every non-root key is larger than its parent. For simplicity, we assume keys to be distinct. Internally, a forest of multiway heaps is represented in the standard way by storing the children of every node as a linked list, see e.g. [Knu97, §2.3.2]. The collection of top-level roots in the forest is also stored in a linked list. We assume that all lists are *doubly-linked*, providing the pointers `next` and `prev` between neighboring siblings, with the appropriate markers at the two ends of the list. We further assume that every node has a pointer to its *leftmost* and to its *rightmost* child, and that we have a global pointer to the *leftmost* and *rightmost* among the top-level roots. The presence of parent pointers is not assumed (Figure 5).
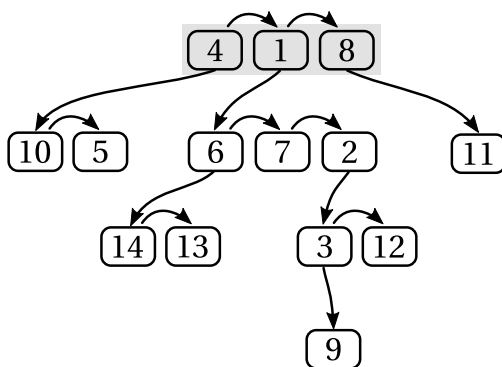

Figure 5: Forest-of-heaps structure with lists of siblings. Arrows show *leftmost* and *next* pointers. (The pointers *rightmost* and *prev* are not shown.) Top-level list appears shaded.

In practice, to make the heap more space-efficient, it is desirable to have only two pointers per node, instead of four, as described. To this effect, algorithms in the stable heap model can also be implemented using only one of `prev` or `next`. Furthermore, one of the pointers `leftmost` and `rightmost` can be simulated in constant time by making the lists circularly linked. The discussion of similar issues of implementation by Fredman et al. [FSST86] for pairing heaps also applies to our model.

---

[15]We opt for a forest-based representation in order to simplify the implementation of insert, meld, and decrease-key operations, which can now be performed lazily. It is not hard to change our model such that it is based on a single heap. Fredman [Fre99b] describes a general transformation that turns various heap implementations into the forest-of-heaps representation.

Stable link.   The defining feature of the stable heap model is the stable link operation (Figure 1). We denote by `link`(x) the operation of linking x and its right sibling $y = x$.`next` with a stable link. If the key of x is smaller than the key of y, then y becomes the rightmost child of x. Otherwise, x becomes the leftmost child of y. (Contrast this again with the standard, "unstable", link operation where the larger item always becomes the leftmost child of the smaller.) Clearly, all necessary pointer changes can be performed in constant time. With a careful implementation, the stable link has, in practice, similar cost as the standard link.

In the following we assume (without explicitly describing the low-level details) that the link operations correctly update all pointers to reflect the structure-change.

Operations.   The heap operations *makeheap*, *insert*, *decrease-key*, and *meld* can be implemented in a straightforward way, identically to pairing heaps, apart from our use of stable links instead of standard links.

Makeheap creates a new, empty heap with the structure described above. Insert creates a singleton root with the new key, and appends it to the list of top-level roots of the heap. Melding two heaps concatenates their top-level root-lists. Decrease-key detaches the tree rooted at the node whose key is decreased and appends it to the top-level root list.[16] All four operations require constant number of pointer moves and pointer changes.

We now describe extract-min, the most complex operation. To find the minimum, the roots in the top-level list are consolidated into a single tree, through a sequence of stable links. In our model *only neighboring siblings* can be linked. Every link operation removes one root from the top-level list (the one with the larger key of the two linked). Thus, the number of links performed during extract-min is exactly one less than the initial size of the top-level list. After a single top-level root remains (the minimum), it is deleted, and its list of children becomes the new top-level list.

Algorithms in the stable heap model differ only in the order in which they link items during the extract-min operation. At the start of extract-min, a cursor is assumed to point to the leftmost node in the top-level list. Algorithms are allowed to move the cursor to the right or to the left, make comparisons on keys of visited nodes, and perform stable links at the cursor.

We call algorithms in the stable heap model *stable heap algorithms*, and we call the entire structure a *stable heap data structure*.

Cost model.   We define the cost of operations to be the *link-only cost*, i.e. the number of stable link operations performed. Thus, the worst-case cost of operations other than extract-min is constant, whereas the cost of extract-min equals the number of top-level nodes before the operation.

It may seem unrealistic to ignore the cost of comparisons and pointer moves. We justify this choice as follows: (1) In all algorithms that we consider, the number of pointer moves and comparisons will in fact be proportional to the link-only cost, therefore, the use of link-only cost is accurate for these algorithms. (2) We can prove *lower bounds* even for the link-only cost, that

---

[16]The easiest choice is to append the node at one of the ends of the top-level list. An alternative implementation (more in the spirit of stable heaps) would be to "sift-up" the node with decreased key, placing it between its successor and predecessor in the top-level list, according to the insertion-times.

is, our lower bounds hold even if pointer moves and comparisons are free. In particular, our lower bounds hold even if the outcomes of all comparisons are known in advance.[17] [18]

In the above description, we only link nodes at the top level. This is only for simplicity, and our lower bounds also apply to algorithms that can link siblings at any level of the heap.[19]
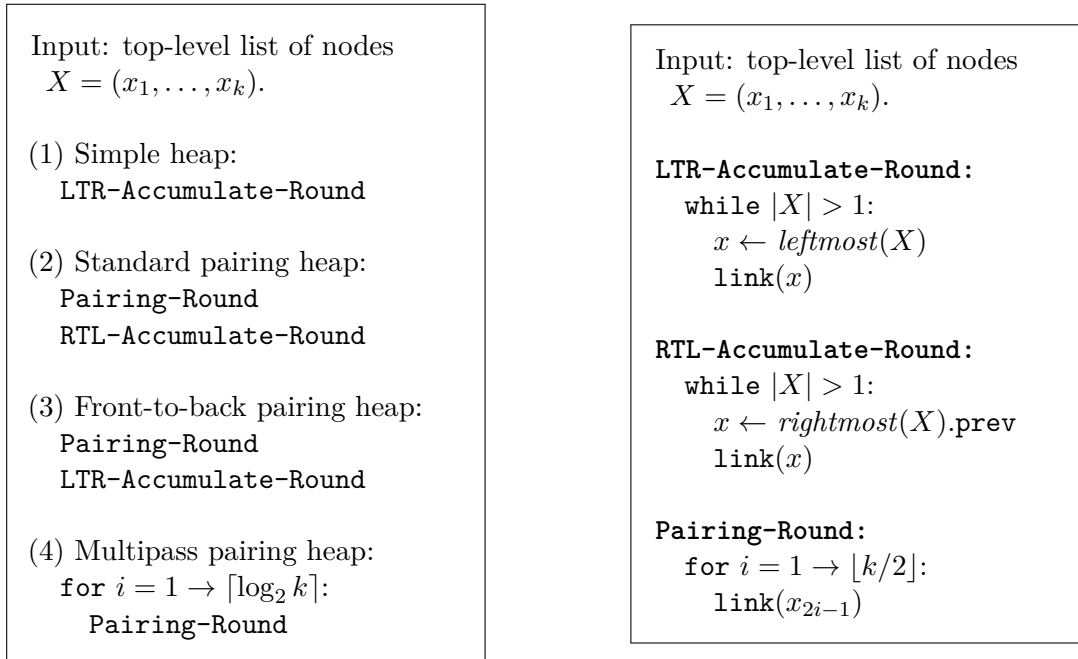
<div style="border:1px solid;padding:1em">

Input: top-level list of nodes
  $X = (x_1, \ldots, x_k)$.

(1) Simple heap:
  ```
  LTR-Accumulate-Round
  ```

(2) Standard pairing heap:
  ```
  Pairing-Round
  RTL-Accumulate-Round
  ```

(3) Front-to-back pairing heap:
  ```
  Pairing-Round
  LTR-Accumulate-Round
  ```

(4) Multipass pairing heap:
  ```
  for i = 1 → ⌈log₂ k⌉:
    Pairing-Round
  ```

</div>

<div style="border:1px solid;padding:1em">

Input: top-level list of nodes
  $X = (x_1, \ldots, x_k)$.

```
LTR-Accumulate-Round:
  while |X| > 1:
    x ← leftmost(X)
    link(x)

RTL-Accumulate-Round:
  while |X| > 1:
    x ← rightmost(X).prev
    link(x)

Pairing-Round:
  for i = 1 → ⌊k/2⌋:
    link(x_{2i-1})
```

</div>

Figure 6: (left) Four heap algorithms. (1) A naive restructuring. (2)-(4) Variants of pairing heaps introduced by Fredman et al. [FSST86]. (right) Subroutines implementing linking rounds on the list of nodes. The operation $\texttt{link}(x)$ links the node x with its right neighbor. To transform the algorithms into stable heap algorithms, $\texttt{link}(x)$ needs to be implemented as stable link.

Sorting-mode. In this paper we look at stable heap algorithms in *sorting-mode* (see §1), and analyse the amortized cost of smooth heap operations in this mode only. That is, we assume that a makeheap operation is followed by n insert operations with distinct keys, and finally, by n extract-min operations.[20]

---

[17] This assumption can be seen as the "dual" of the BST assumption of knowing in advance the order of operations.

[18] It may seem that knowing the sorted order of keys, a stable heap algorithm can arrange the nodes into a path with a linear number of links, contradicting the information-theoretic lower bound for sorting. Observe however, that we are limited to linking neighboring siblings, which removes the contradiction.

[19] The assumption to link only at the top level is analogous to the BST assumption of only rotating the search path. The restriction does not seem significant, since links at lower levels can be postponed to a later time, when the siblings reach the top level. One case where this may make a difference is if decrease-key operations are involved; performing links at lower levels may allow some items to "travel together" with the item whose key is decreased.

[20] While sorting-mode is somewhat restrictive, we remark that the complexity of classical pairing heap variants such as the front-to-back or multipass heuristics is not known, even in sorting-mode [FSST86, DKKZ18, DKK⁺18].

Algorithms. We mention four algorithms for heap re-structuring: a *simple* (folklore) algorithm, and the three *pairing heap variants* introduced by Fredman et al. [FSST86]. All four algorithms can be implemented with the standard (unstable) link operation, and can be turned into stable heap algorithms if stable link is used instead of the standard link.

We describe the implementation of extract-min in the four algorithms (see Figure 6).

The first, **simple heap** collects the roots in a left-to-right accumulation round, repeatedly linking the (current) leftmost item with its right neighbor. It is easy to construct examples where the amortized cost of extract-min operations is linear, i.e. prohibitively large (e.g. $1, 2, 3, \ldots$ for the stable variant and $1, 3, 5, \ldots, 6, 4, 2$ for the unstable variant).

The **standard pairing heap** works in two passes: a left-to-right pairing round in which neighboring items are linked in pairs and a subsequent right-to-left accumulation round. The amortized cost of operations is $O(\log n)$ [FSST86].

The **"front-to-back" variant of pairing heaps** differs from the standard variant only in the fact that the second round is performed left-to-right, rather than right-to-left. In [FSST86] only an $O(\sqrt{n})$ bound is given for the amortized cost of extract-min using this method. This was recently improved in [DKKZ18] to $O(\log n \cdot 4^{\sqrt{\log n}})$, still far from the conjectured logarithmic bound.

The **"multipass" variant of pairing heaps** repeatedly executes pairing rounds (identical to the first round of the standard and front-to-back variants), until a single root remains. In [FSST86] the bound $O(\log n \cdot \log \log n / \log \log \log n)$ is given for the cost of extract-min using this method. This was recently improved in [DKK+18] to $O(\log n \cdot 2^{\log^* n} \cdot \log^* n)$, again, not known to be tight.

The mentioned bounds for the three pairing heap variants refer to the implementations with the standard link operation. Only the analysis of [DKKZ18] transfers readily to the stable setting, giving a bound of $O(\log n \cdot 2^{O(\sqrt{\log n})})$ for all three variants [DKKZ18, §2.2]. It may well be that the true amortized costs of all three pairing heap variants (both stable and unstable) are $O(\log n)$.

3.2 Comparison with other heap models

Fredman [Fre99a] and Iacono and Özkan [IÖ14b, IÖ14a] define general heap models, for the purpose of proving lower bounds for all algorithms within the model. The models of Fredman and Iacono and Özkan are similar to each other and to our stable heap model in that they work on a forest-of-heaps structure, allowing traversal by pointer moves, comparisons, and link operations.

In Fredman's "generalized pairing heap" model, the amount of information stored in the nodes of the underlying tree is restricted. Comparisons between keys are only allowed together with a subsequent link operation. Pointer moves (for reaching the keys to be compared) are free.

In the "pure heap" model of Iacono and Özkan, comparisons are decoupled from links. A wider set of pointer moves and pointer changes are allowed, but their cost is accounted for.

Our stable heap model borrows elements from both models, but is not directly comparable to them. It is less restrictive in that it allows arbitrary information to be stored in the nodes, and allows arbitrary pointer moves and comparisons for free. It is more restrictive in that changing the structure can happen only via link operations, and only neighboring siblings can be linked.

The main difference between our model and the other two models is in the link operation: whereas the other two models use the classical, unstable, "link-as-leftmost" method, the stable heap model uses stable links. We find the restrictions of our model fairly natural (in hindsight), and in

the spirit of classical algorithms, such as pairing heaps – apart from the use of stable links, which we see as a natural replacement of standard links. The main advantage of our model is its surprising connection to the standard BST model.

The lower bounds in our model are of a different kind than the lower bounds shown in the existing models. The lower bounds of Fredman and Iacono-Özkan are concrete constructions of particularly costly sequences of operations, i.e. they are lower bounds *for the worst-case*. The lower bounds we show in the stable heap model are *instance-specific*, i.e. they apply to all sequences of operations, and describe structural properties that capture the difficulty of these sequences. (In this sense, they are similar to the lower bounds in the BST model [Wil89, DHI+09]; in fact, they are the *exact same* bounds, transferred from the BST to the stable heap model.)

3.3 Stable heaps in sorting-mode

Let $X \in [n]^n$ be a permutation sequence. We consider the *execution trace* of a stable heap algorithm for n extract-min operations, after inserting the keys in $[n]$ in the order given by X into an initially empty heap (i.e. in sorting-mode).

For a stable heap algorithm $\mathcal{A}$, we denote the set of links performed during the sorting-mode execution of $\mathcal{A}$ on X as $\mathcal{A}(X)$, i.e. $(a, b) \in \mathcal{A}(X)$ if, at some point during the execution of $\mathcal{A}$ on X, the nodes with keys a and b are linked. Observe that a pair of nodes can be linked only once during the execution (once a node is in the subtree of the other, it stays there). Thus, the cost of the execution is $|\mathcal{A}(X)|$. We denote by $\mathsf{OPT}_{\texttt{stable}}(X)$ the minimum cost of a stable heap algorithm when serving X in sorting-mode.

In the following we describe the combinatorial *star-path problem* as an intermediate step towards proving the formal connection between the stable heap model and the BST model (§5).

Recall that a point set $P \subseteq [n] \times [n]$ is a permutation if $|P_{x=i}| = |P_{y=i}| = 1$ for each i. In this case, we denote as $p_{y=i}$ the unique point in $P_{y=i}$, and similarly for $p_{x=i}$. Let P_0 denote the set $\{\langle 0, 0 \rangle\} \cup P$, i.e. the set P augmented with the origin.

We consider trees with node-set P_0. We call such a tree a *monotone tree*, if $\langle 0, 0 \rangle$ is the root, and for every edge (u, v) of the tree, $u.y < v.y$ iff u is closer to the root (in graph-theoretical sense) than v. Two particular monotone trees are important: $star(P)$ is the tree in which every point in P is the child of $\langle 0, 0 \rangle$, and $path(P)$ is the path $(\langle 0, 0 \rangle, p_{y=1}, p_{y=2}, \ldots, p_{y=n})$. See Figure 7 for illustration.
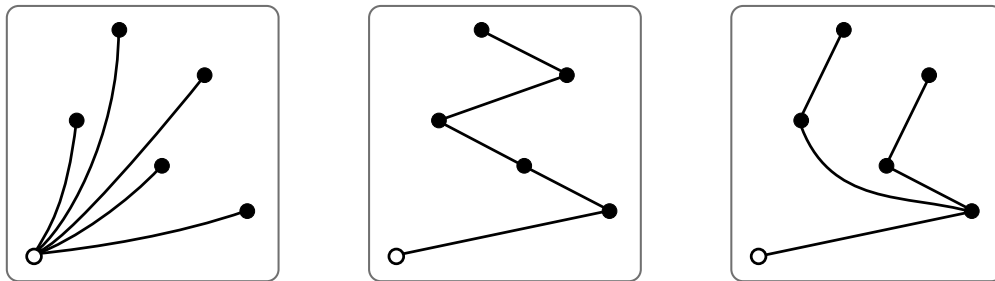

Figure 7: Sequence $X = (3, 5, 2, 4, 1)$ and corresponding point set $P = P^{X'}$ (values shown on y-coordinate). (i) $star(P)$, (ii) $path(P)$, (iii) an arbitrary monotone tree with points in $P \cup \{\langle 0, 0 \rangle\}$.

We define a link operation in monotone trees as follows. Let a and b be two *neighboring* siblings in the tree (according to x-coordinate), and let u be their parent. Suppose $a.y > b.y$. Then $\texttt{link}(a, b)$ deletes the edge (u, a) and adds the edge (b, a) (i.e. changes the parent of a from u to b). Otherwise, if $a.y < b.y$, we delete (u, b) and add (a, b). (See Figure 8.)

A link can be performed on any monotone tree that has a node with at least two children, e.g. $star(P)$. Starting from $star(P)$, after at most $O(n^2)$ links we reach $path(P)$, and no more links are possible. (To see this, we can argue that the total length of the y-components of edges decreases with every link.)
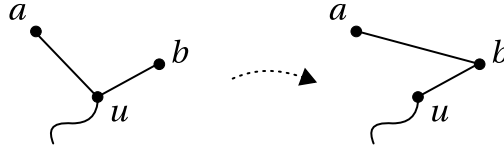

Figure 8: Operation of linking a and b when $a.y > b.y$.

Definition 3.1 (Star-path problem). *Given a permutation point set P, transform $star(P)$ into $path(P)$ using a sequence of link operations.*

For a point set P, we denote by $\mathcal{B}(P)$ the set of links performed during the execution of an algorithm $\mathcal{B}$ for the star-path problem with input P, i.e. $(a, b) \in \mathcal{B}(P)$, if at some point, $\mathcal{B}$ links a and b (again, observe that this can happen at most once). We show that stable heap executions for a permutation sequence X and star-path executions for point set $P^{X'}$ (from $star(P^{X'})$ to $path(P^{X'})$) are, in a precise sense, equivalent.

Theorem 3.2. *Let $X = (x_1, \ldots, x_n)$ be an arbitrary permutation of $[n]$, and let $\mathcal{P} \subset [n]^2$. Then there is a stable heap algorithm $\mathcal{A}$, such that $\mathcal{A}(X) = \mathcal{P}$, iff there is an algorithm $\mathcal{B}$ for the star-path problem, such that $\mathcal{B}(P^{X'}) = \mathcal{P}$.*

Proof. Let $\mathcal{A}$ be an arbitrary stable heap algorithm executed in sorting mode for X. After inserting X, the top-level roots are $x_1, \ldots, x_n$ (in this order). We assume these roots to be the children of a virtual root with key 0. Thus, at this stage, the heap structure is exactly $star(P^{X'})$, i.e. the i-th child of the root is $\langle i, x_i \rangle$. Furthermore, we assume that after each extract-min, the old root is "kept around", considering the children of the old root the new top level roots. Thus, after n extract-min operations, the structure of the heap is exactly $path(P^{X'})$, i.e. the sorted path of X, with a dummy root in front. These adjustments to $\mathcal{A}$ are only for convenience, and do not affect its behavior.

To show the equivalence between the two executions, we identify a node with key x_i in the heap with the point $\langle i, x_i \rangle$ in the star-path problem, and we describe a bijection between link operations in the two executions. We maintain the following two invariants:

(Invariant 1): The current heap state is the same as the current monotone tree in the star-path execution (with the virtual roots added in both cases). The keys in the heap view correspond to the y-coordinates in the star-path view, and the left-to-right ordering of siblings in the heap view corresponds to the left-to-right ordering by x-coordinates in the star-path view.

(Invariant 2): For every node q in the monotone tree, we have $parent(q).\texttt{prev}.x < q.x < parent(q).\texttt{next}.x$. Here, $\texttt{prev}$ and $\texttt{next}$ denote the left and right neighboring siblings of a node (by

x-coordinate) in the monotone tree. For convenience we assume that if q has no left neighbor, then $q.\texttt{prev}.x = -\infty$, and if q has no right neighbor, then $q.\texttt{next}.x = +\infty$.

Initially, in the case of a star, both invariants clearly hold. We need to show that linking maintains these invariants.

Consider a link between a and b with parent u, where a is to the left of b. Assume $a.y > b.y$, as in Figure 8. After the link, a becomes the child of b.

In the heap, by Invariant (1), the corresponding items x_a and x_b are neighboring siblings ($a < b$), and $x_a > x_b$. Therefore, linking x_a and x_b is a valid operation and x_a becomes the leftmost child of x_b. (Conversely, if x_a and x_b are neighboring siblings in the heap, then linking a and b is a valid star-path link.)

We need to show that a becomes the leftmost child of b, and thus, the ordering of siblings is by x-coordinate. By Invariant (2), before the link operation for all children c of b we have $c.x > parent(c).\texttt{prev}.x = a.x$. Thus, Invariant (1) is maintained.

We need to show that Invariant (2) is not violated. This could only happen if, after the link operation, $a.x$ were smaller than $b.\texttt{prev}.x$. But this is impossible, since the left neighbor of b is the earlier left neighbor of a. The case $a.y < b.y$ is symmetric, and omitted. ∎

Observe that in the star-path problem only the set of edges changes during the execution, the locations of points remain the same. In order to maintain this simple geometric model, the use of stable links is essential. (The classical link operation would move entire subtrees from one place to another.) In the remainder of the paper we view stable heap executions mostly in the "geometric view" of the star-path problem.

4   The smooth heap

In this section we describe the smooth heap, our new heap data structure. The smooth heap conforms to the stable heap model described in §3 and is based on a forest-of-heaps representation. The implementations of makeheap, insert, decrease-key and meld are those from the description of stable heap algorithms in §3.

The crucial operation is the restructuring of the top-level list of nodes during extract-min. In the following we give three *equivalent* descriptions of this operation; a *non-deterministic* description, a *treap-based* description, and a *two-pass* description. We refer to Figure 9 for the pseudocodes of the three variants.

Non-deterministic view.   In the top-level list X of items, we repeatedly find an arbitrary *local maximum* x. A local maximum is an item x that is larger than both its neighboring siblings, or, in case x is the leftmost or rightmost item, larger than its only neighbor.

We link x with the *larger* of its two neighboring siblings (or its only sibling, in case it is the leftmost or rightmost item). As x becomes the child of one of its neighbors, it drops out of X, reducing the size of X by one. As long as X has at least two elements, there is a suitable next choice of x (for instance, the global maximum of X). When X becomes a singleton, we are done; as this item is the minimum, it can be deleted.

Input: top-level list of roots $X = (x_1, \ldots, x_k)$

```
while |X| > 1:
    let x be an arbitrary local maximum in X
    if x.prev.key > x.next.key              (assuming null.key = −∞)
        link(x.prev)
    else
        link(x)
```

Smooth heap (*non-deterministic* view)

```
Transform X into a treap with keys (1, 2, ..., k) and priorities (x₁, ..., xₖ).
```

Transform X into a treap with keys $(1, 2, \ldots, k)$ and priorities $(x_1, \ldots, x_k)$.

Smooth heap (*treap* view)

```
Smoothing-Round:
while there is x in X s.t. x.key > x.next.key:
    let x be the leftmost such node
    if (x.prev = null) or
    (x.prev.key < x.next.key)
        link(x)
    else
        link(x.prev)

RTL-Accumulate-Round:
    while |X| > 1:
        x ← rightmost(X).prev
        link(x)
```

Smooth heap:
   Smoothing-Round
   RTL-Accumulate-Round

Smooth heap (*two-pass* view)

Linking rounds

Figure 9: Smooth heap re-structuring (three different views). Recall that `link(y)` denotes a stable link between y and its *right* neighbor y.`next`.

The non-deterministic view is useful in analysing smooth heaps, because of its resemblance to our non-deterministic view of Greedy (§6).

Two-pass view. This description differs from the non-deterministic description only in the choice of the local maximum x: we always choose the *leftmost* such item. Observe that if the only remaining local maximum is the rightmost item, then the items in the list are *sorted*. In this case, the remaining items can be linked in a single right-to-left pass, which we can execute without further comparisons. It is thus convenient to view the execution in two passes that resemble the description of pairing heaps: a left-to-right *smoothing pass* followed by a right-to-left accumulation pass. The two-pass view of smooth heaps is perhaps the most convenient to implement. We describe this implementation in more detail in Appendix D.

Treap view. We associate each item x_i in X with a pair of values (i, x_i), and we transform the list into a treap over the pairs (using an arbitrary method for treap-building). Recall that a treap is a binary tree with a pair of values in every node, respecting the in-order (i.e. search tree order) according to the first entry, and the min-heap order according to the second entry of every pair. As mentioned, such a tree is unique. (The item with the unique minimum priority is the root, and the items with smaller, resp. larger key values form its recursively-built left and right subtrees). We use the treap view in order to connect the non-deterministic and two-pass views of the smooth heap.

A remark is in order: as a treap is a binary tree, each node may have a left and a right child. For every node in X, its left child in the treap will become its *leftmost* child in the underlying tree, and its right child in the treap will become its *rightmost* child. In other words, the existing children of x_i end up *between* the two new children possibly gained during the treap-building.

Theorem 4.1. *The non-deterministic, two-pass, and treap-based descriptions of smooth heaps describe the same transformation.*

We show that the non-deterministic algorithm constructs a treap regardless of the order in which local maxima are chosen, and thus the non-deterministic and treap-based descriptions are equivalent. As the two-pass description is just a particular way of choosing the local maxima, it follows that it also produces a treap, and since the treap is unique, all three views are equivalent.

Although there are several linear-time algorithms known for constructing a treap from an array, e.g. [GBT84], we are not aware of a previous mention in the literature of the particular method implicit in the description of the smooth heap. Due to its simplicity, we find it interesting in its own right; in the following, we prove its correctness.

Proof. We refer to the non-deterministic description in Figure 9, and denote by $X = (x_1, \ldots, x_k)$ the initial top-level list of items.

Let T be the tree built from X through repeatedly linking local maxima of X with their larger neighbor, until a single item remains. Three properties of T together imply that T is a treap.

(1) T is a *heap* according to the key-values x_i,
(2) T is a *binary* tree,
(3) T is a *search tree*, according to the indices i.

Property (1) follows from the definition of linking (links only create edges where the parent is smaller than the child).

To see Property (2), observe that once an item x is larger than its left neighbor x.prev, it continues to be larger as long as it stays in X (this is because if the left neighbor of x drops out of X, then it must have become the child of even smaller item, which is the new left neighbor of x). Similarly, once x is larger than its right neighbor x.next, it will remain so, as long as it stays in X. Suppose x and y are linked, and x gains y as a child. Then, according to our "two-choices" strategy, x must have gained a new neighbor that is smaller than x (the old neighbor of y), unless x is already the leftmost or rightmost in the list. It follows that x can gain at most two children (one as a new leftmost child, and one as a new rightmost child).

Property (3) follows from the fact that the list is initially ordered by the indices i and stable links maintain this order. (If a node has a single child, we consider it a right child or left child, depending on whether the child was linked as rightmost or leftmost.) ▮

Remark: As mentioned, the smooth heap is implemented using stable links. In the non-deterministic and two-pass descriptions it is in fact also possible to use the classical, unstable link. In this case, however, the three descriptions are no longer equivalent. It is an interesting open question how efficient the resulting unstable two-pass algorithm is.

It is instructive to compare smooth heaps and pairing heaps. Besides the implementation of the link operation (stable or unstable) there are several differences between the two algorithms.

In pairing heaps, as in Fredman's model of generalized pairing heaps, comparisons only occur within a link operation, i.e. a comparison is always followed by the corresponding link. By contrast, in smooth heaps, comparisons are decoupled from links, and are also used to decide *which pair of nodes* to link. In fact, comparisons are used *only* for this purpose: it can be observed in the two-pass description in Figure 9 or in the description of Appendix D, that at the time of linking, it is always already known, which of the two linked items is greater, therefore, the link operation can proceed without any comparison.

Recall that in our model, the cost of the algorithm is its "link cost", i.e. the number of link operations performed. In the detailed description of Appendix D it can be observed that the number of other elementary operations is proportional to the link cost. In particular, every comparison is followed by moving the cursor to the right, or by a link at the cursor. It follows that the number of comparisons is at most twice the number of links. (The number of items to the right of the cursor *plus* the total number of items decreases after every comparison.)

The distinctive feature of smooth heaps is their *power-of-two-choices* linking, whereby an item x is linked with the larger of its two neighbors (both of which are smaller than x). In other words, of the two possible edges we could create, we choose the one with smaller rank-difference. Intuitively, with this choice, we expect to move closer to the totally ordered state (i.e. a path in which the rank-difference along every edge is one). Depending on the subtrees of the nodes, this choice may, of course, not be globally optimal, as shown in Figure 25 in Appendix E. As shown in §5 there is good reason to believe that the smooth heap is not far from being optimal; a large gap from the optimum would disprove the conjectured optimality of Greedy. In Figure 10 we show examples that illustrate the behavior of smooth heaps compared to pairing heap variants. An intuitive reason for the efficiency of the smooth heap is that it does the optimal transformation for both increasing and
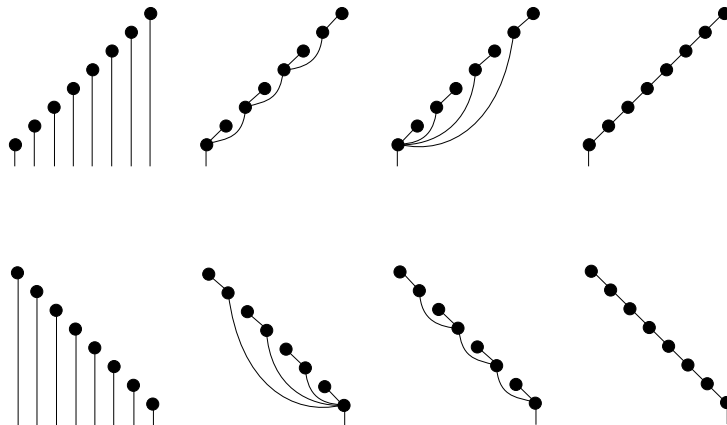
Figure 10: Extract-min operation illustrating the difference between smooth heap and pairing heap with increasing (above) and decreasing (below) initial state. From left-to-right: initial state, standard pairing heap, front-to-back pairing heap, smooth heap. The behavior of smooth heap is similar to the *better* of the two other algorithms, in both cases. A transformation is better, if its result is closer to the fully sorted state, i.e. to a path. This is the case for the standard pairing heap for increasing, for the front-to-back pairing heap for decreasing, and the smooth heap for both increasing and decreasing input. In the other cases, the resulting heaps resemble the initial, unsorted state.

decreasing (sub)sequences, see Figure 10. (By contrast, the standard and back-to-front pairing heap variants are efficient in one and inefficient in the other case.)

Finally, we give some remarks on how smooth heaps (in the two-pass view) can be implemented with only two pointers per node. We refer to the description in Appendix D. First, we make the lists circularly linked, such that the `next` pointer of the rightmost item points to the leftmost item in every list, marking the rightmost item appropriately. We can now omit the `leftmost` pointer, as it can be simulated via the `rightmost` and `next` pointers. Getting rid of the `prev` pointer is trickier. Observe that in the first, left-to-right pass we may need, after a link, to make a step to the left with the cursor, if an item was linked with its left neighbor. Finally, in the second, right-to-left pass we need to make several steps to the left. In order to simulate these steps without using `prev` pointers, we can *reverse* the links of the list as we traverse it left-to-right, such that the `next` pointers of the already visited items point to their left neighbors in the list. (If we move left, we undo the reversal.) This allows us to move the cursor both left and right from its current position using only `next` pointers, and a constant-size buffer to hold the two items at the cursor (and a constant factor overhead in cost). We omit further details.

5 The transformations and their consequences

In this section we present the main connections between stable heap algorithms and BST algorithms and we discuss the consequences of these connections.

We show that the cost of smooth heaps in sorting-mode, once the role of key-space and time are swapped, matches the cost of GreedyFuture, conjectured to be instance-optimal in the BST model.

Theorem 5.1 (Smooth-Greedy transformation). *For every permutation X,*

$$|GreedyFuture(X')| = \Theta(|Smooth(X)|).$$

As an immediate consequence of Theorem 5.1 and the amortized analysis of GreedyFuture [Fox11], we have the following.

Corollary 5.2 (Amortized analysis of smooth heaps). *For every permutation $X \in [n]^n$,*

$$|Smooth(X)| = O(n \log n).$$

For the amortized cost of GreedyFuture several stronger, instance-specific bounds are known. We describe two from the literature.

The *weighted dynamic finger $WDF(X)$* of an input sequence X is a quantity that describes its *locality of reference*. It was introduced in [IL16], and it subsumes the earlier *dynamic finger* bound [ST85, CMSS00, Col00], as well as other bounds (see e.g. [CGK+16]). For the exact definition of $WDF(\cdot)$ we refer to [IL16].

A permutation $X \in [n]^n$ *avoids* a pattern permutation $\pi \in [k]^k$ if there is no subsequence of X (not necessarily contiguous) that is *order-isomorphic* to π. (We refer the reader to [Knu68, Tar72, Pra73, Kit11, GM14, NRRS17] for more information on this extensively studied property.) As a simple observation, we mention that if X avoids π then X' avoids π'.

We have the following results.

Theorem 5.3. *For every permutation $X \in [n]^n$:*

- *[IL16]* $|Greedy(X)| = O(WDF(X))$.

- *[CGK+15b] If X avoids some permutation $\pi \in [k]^k$, then $|Greedy(X)| = n \cdot 2^{\alpha(n)^{O(k)}}$, where $\alpha(\cdot)$ is the slowly growing inverse Ackermann function.*

We immediately obtain the following.

Corollary 5.4 (Instance-specific upper bounds for smooth heaps). *For every permutation $X \in [n]^n$:*

- $|Smooth(X)| = O(WDF(X'))$.

- *If X avoids some permutation $\pi \in [k]^k$, then $|Smooth(X)| = n \cdot 2^{\alpha(n)^{O(k)}}$.*

More generally, we show that there is a general transformation from *an arbitrary* stable heap algorithm to an offline BST algorithm:

Theorem 5.5 (General transformation)**.** *For every stable heap algorithm $\mathcal{A}_{\mathtt{stable}}$ there is an offline BST algorithm $\mathcal{A}_{\mathtt{bst}}$, such that for every permutation X, the costs of $\mathcal{A}_{\mathtt{stable}}$ and $\mathcal{A}_{\mathtt{bst}}$ for sorting X are asymptotically the same, i.e.*

$$|\mathcal{A}_{\mathtt{bst}}((X')^r)| = \Theta(|\mathcal{A}_{\mathtt{stable}}(X)|),$$

and in particular,

$$\mathsf{OPT}_{\mathtt{ibst}}((X')^r) = O(\mathsf{OPT}_{\mathtt{stable}}(X)).$$

Observe that the permutation input of $\mathcal{A}_{\mathtt{bst}}$ is inverted *and* reversed. From Remark 2.6, we note again that bounding from above the running time of a BST algorithm in sorting mode is stronger than bounding it in search-only mode.

For search-only mode, we can obtain infinitely many offline BST algorithms from a single stable heap algorithm, all of which have at most a constant factor larger cost. (Here, *infinitely many* is understood as n tends to infinity, i.e. the number of BST algorithms generated depends on the input size n.) In particular, we obtain multiple (non-trivial) offline BST executions that are $O(1)$-competitive with GreedyFuture.

The proofs of these results are presented in §7. Below we discuss some of their interesting consequences.

5.1 Consequences of the optimality of **GreedyFuture**

GreedyFuture is widely conjectured to be instance-optimal [DHI+09, Mun00, Luc88]. In case the conjecture is true, we obtain the following two statements. First, our smooth heap algorithm is an instance-optimal stable heap algorithm (at least) for sorting. Second, the optimal costs of (1) selection-sort with stable heaps ($\mathsf{OPT}_{\mathtt{stable}}$), (2) insertion-sort with BSTs ($\mathsf{OPT}_{\mathtt{ibst}}$), and (3) searching with BSTs ($\mathsf{OPT}_{\mathtt{bst}}$) are the same within some constant factor. Moreover, each of these quantities are invariant under applying inversion and/or reversion to the input.

Without the assumption, we only know that, for every permutation X, we have $\mathsf{OPT}_{\mathtt{ibst}}(X) \geq \mathsf{OPT}_{\mathtt{bst}}(X)$ by Remark 2.6, and $\mathsf{OPT}_{\mathtt{stable}}(X) \geq \Omega(\mathsf{OPT}_{\mathtt{ibst}}((X')^r))$ by Theorem 5.5. It is not clear how to prove the inequalities in the other direction, or how to compare $\mathsf{OPT}_{\mathtt{stable}}(X)$ and $\mathsf{OPT}_{\mathtt{ibst}}(X)$.

Corollary 5.6. *Assuming Conjecture 2.9, for every permutation X,*

(i)

$$\mathit{Smooth}\,(X) = \Theta(\mathsf{OPT}_{\mathtt{stable}}(X)),$$

(ii)

$$\mathsf{OPT}_{\mathtt{stable}}(X) = \Theta(\mathsf{OPT}_{\mathtt{ibst}}(X)) = \Theta(\mathsf{OPT}_{\mathtt{bst}}(X)),$$

and, for each model $\in \{\mathtt{stable}, \mathtt{ibst}, \mathtt{bst}\}$,

$$\mathsf{OPT}_{model}(X) = \Theta(\mathsf{OPT}_{model}(X')) = \Theta(\mathsf{OPT}_{model}(X^r)).$$

Proof. We use $f \lesssim g$ to denote $f = O(g)$. Consider the following inequalities, for an arbitrary permutation X.

$$\begin{aligned}
\mathsf{OPT_{bst}}(X) &\leq \mathsf{OPT_{ibst}}(X) && \text{by Remark 2.6} \\
&\lesssim \mathsf{OPT_{stable}}((X')^r) && \text{by Theorem 5.5} \\
&\leq |\mathsf{Smooth}\,((X')^r)| && \\
&\lesssim |\mathsf{GreedyFuture}\,(X^r)| && \text{by Theorem 5.1} \\
&\lesssim \mathsf{OPT_{bst}}(X^r) && \text{by Conjecture 2.9} \\
&= \mathsf{OPT_{bst}}(X) && \text{by Corollary 2.4.}
\end{aligned}$$

Assuming Conjecture 2.9, the inequalities collapse, implying (i).

For (ii), we have $\mathsf{OPT_{bst}}(X) = \Theta(\mathsf{OPT_{ibst}}(X))$ and $\mathsf{OPT_{stable}}((X')^r) = \Theta(\mathsf{OPT_{bst}}(X))$ from the above inequalities. By Corollary 2.4, $\mathsf{OPT_{bst}}(X) = \Theta(\mathsf{OPT_{bst}}(X')) = \Theta(\mathsf{OPT_{bst}}(X^r))$ for every permutation X. ∎

5.2 Dynamic optimality for stable heaps

The theory of instance-specific lower bounds for BSTs is much richer than the corresponding theory for heaps. There are several *concrete* lower bounds known for $\mathsf{OPT_{bst}}$, Wilber's first bound (a.k.a. the "interleave bound"), Wilber's second bound (a.k.a. the "funnel bound") [Wil89] and the maximum independent rectangle (MIR) bound [DHI+09]. (See also [Iac13] for the precise definitions of these quantities.) It is typically easier to reason about these lower bounds than to analyse $\mathsf{OPT_{bst}}$ directly. Currently, all BST algorithms shown to be $o(\log n)$-competitive (i.e. Tango [DHIP07], Multi-splay [WDS06] and Chain-splay [Geo08]) have been shown to be competitive with Wilber's first bound. Some connections between these bounds are known. In [DHI+09] it is shown that both Wilber's bounds are subsumed by the MIR bound, which is computable by a sweepline algorithm (intriguingly) similar to Greedy. Harmon [Har06] also shows that the MIR bound is captured (up to a constant factor) by the optimal cost of a network-design problem called *small Manhattan networks* [GKKS07].[21]

For heaps, lower bounds have so far been studied in the *worst-case* setting [Fre99a, IÖ14b, IÖ14a], and a theory of instance-specific lower bounds has not yet been proposed (in any heap model).

Our results connect the two models, and yield an analogous theory for heaps (at least with the restrictions implied by the stable heap model). By Theorem 5.5, for *every* stable heap algorithm, there is a corresponding BST algorithm with the same asymptotic cost. By this result, all known instance-specific lower bounds are immediately transferred from BSTs to stable heaps.

Corollary 5.7. *Let $W_1(X)$, $W_2(X)$, $MIR(X)$ be Wilber's two bounds [Wil89], and the maximum independent rectangle (MIR) bound [DHI+09] for an arbitrary permutation X. Then we have*

$$W_1(X), W_2(X) \leq MIR(X) \leq O(\mathsf{OPT_{stable}}(X)).$$

Proof. In [DHI+09] it is shown that $W_1(X), W_2(X) \leq MIR(X) \leq O(\mathsf{OPT_{bst}}(X))$. The following

---

[21]Wilber's first bound is in fact, a family of bounds, as it is defined with respect to a reference tree. Here, by $W_1(X)$ we denote the best bound in the family, i.e. using the reference tree that maximizes the bound for given X.

concludes the claim:

$$\begin{aligned}
\mathsf{OPT}_{\mathsf{bst}}(X) &= \mathsf{OPT}_{\mathsf{bst}}((X')^r) && \text{by Corollary } 2.4 \\
&\leq \mathsf{OPT}_{\mathsf{ibst}}((X')^r) && \text{by Remark } 2.6 \\
&= O(\mathsf{OPT}_{\mathsf{stable}}(X)). && \text{by Theorem } 5.5
\end{aligned}$$

∎

6 A non-deterministic description of **Greedy**

In this section we describe a non-deterministic algorithm for the satisfied superset problem whose output is exactly the same as the output of Greedy, described in §2. We call this algorithm "non-deterministic Greedy" and denote it Greedy_{nondet}. This alternative description of Greedy can be of independent interest; we use it to show a connection between the smooth heap and Greedy in §7. The equivalence between Greedy and Greedy_{nondet} is shown in Theorem 6.10. We first define the ADD gadget.

Definition 6.1 (ADD gadget). *Let $P \subseteq \mathbb{Z} \times \mathbb{Z}$ be a point set. We say that $G = (a, b, c, d, e)$ is an* ADD *gadget in P if*

- $\{a, b, c, d, e\} \subseteq P$

- *The relative positions of a, b, c, d, e are according to Figure 11 or its horizontal reflection. That is, $a.y > b.y > c.y = d.y = e.y$ and $e.x < a.x = c.x < b.x = d.x$, or $e.x > a.x = c.x > b.x = d.x$.*

- *Let $f = \langle b.x, a.y \rangle$. Then $f \notin P$*

- *Let $\square_G = \square_{ef} \setminus (\{\langle e.x, i \rangle \mid i\} \cup \{\langle i, e.y \rangle \mid i\} \cup \{\langle b.x, i \rangle \mid i\} \cup \{\langle i, a.y \rangle \mid i\})$. In words, $\square_G$ is $\square_{ef}$ with the borders removed. Then, $\square_G \cap P = \emptyset$. See Figure 11. We call $\square_G$ the rectangle inside the gadget G.*
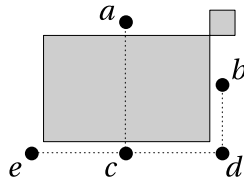
Figure 11: ADD gadget $G = (a, b, c, d, e)$. The larger shaded rectangle is $\square_G$. The small square is $f = \langle b.x, a.y \rangle$. Dotted lines indicate horizontal or vertical alignment. Shaded areas are free of points.

We begin by proving some basic properties of the ADD gadget during the execution of Greedy. Recall the process of Greedy from Definition 2.7. First, we show that Greedy never adds a point inside $\square_G$. This claim follows from [CGK+15b, Lemma 2.4(iii)], but we include the argument for completeness.

Lemma 6.2. *If there is an* ADD *gadget $G = (a, b, c, d, e)$ in point set P, then $\square_G \cap \mathsf{Greedy}(P) = \emptyset$.*
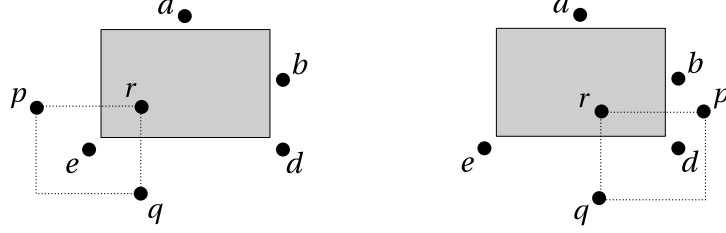
Figure 12: Illustration of the proof of Lemma 6.2. Suppose that r is the lowest point in $\square_G \cap \mathsf{Greedy}\,(P)$. Either e or d contradict the fact that $\square_{pq}$ is unsatisfied.
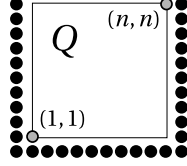

Figure 13: Illustration of $Q \cup \mathtt{box}$ where $Q \subseteq [n] \times [n]$ is a point set.

Proof. We assume that $e.x < d.x$ (the horizontally reflected case is analysed similarly).

Suppose for contradiction that when Greedy runs on P, it adds a point r inside $\square_G$, and let r be the lowest such point. Observe that $e.y < r.y \leq a.y$, and $e.x < r.x < d.x$. Let Q be the point set before processing row $r.y$.

From the definition of Greedy it follows that there are two points p and q where $p \in P_{y=r.y}$ and $q \in Q_{y \leq e.y}$ such that $\square_{pq}$ is unsatisfied and $q.x = r.x$. See Figure 12. We know that either $p.x \leq e.x$ or $d.x \leq p.x$ because $p \notin \square_G$. But since $e.x < q.x < d.x$ (because $q.x = r.x$ and $r \in \square_G$), either e or d must be in $\square_{pq}$, contradicting the claim that $\square_{pq}$ is unsatisfied. $\blacksquare$

By *filling the gadget G* we mean the action of adding the point $f = \langle b.x, a.y \rangle$ into a point set P containing the gadget $G = (a, b, c, d, e)$. We observe that Greedy always fills an $\mathtt{ADD}$ gadget.

Lemma 6.3. *If there is an $\mathtt{ADD}$ gadget $G = (a, b, c, d, e)$ in P, then $\langle b.x, a.y \rangle \in \mathsf{Greedy}\,(P)$.*

Proof. By Lemma 6.2, it holds that $\square_G \cap \mathsf{Greedy}\,(P) = \emptyset$, i.e. Greedy never adds a point within $\square_G$. Consider the step when Greedy processes row $a.y$. There must be b' such that $b'.x = b.x$ and $b.y \leq b'.y < a.y$, and a' such that $a'.y = a.y$ and $a.x \leq a'.x < b.x$ such that $\square_{a'b'}$ is unsatisfied (b' may be b and a' may be a). Therefore, $f = \langle b.x, a.y \rangle$ is added. $\blacksquare$

A straightforward induction shows the following:

Proposition 6.4. *Let P be a point set and $Q \subseteq \mathsf{Greedy}\,(P)$. Then, $\mathsf{Greedy}\,(Q) = \mathsf{Greedy}\,(P)$.*

Let $\mathtt{box} = \{\langle i, 0 \rangle \mid i \in [n]\} \cup \{\langle 0, i \rangle \mid 0 \leq i \leq n\} \cup \{\langle n+1, i \rangle \mid 0 \leq i \leq n\}$. See Figure 13. We state two easy observations.

Proposition 6.5. *Let $Q \subseteq [n] \times [n]$ be an arbitrary point set. Then,*

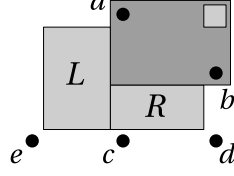*(i)* $\mathsf{Greedy}\,(Q \cup \mathtt{box}) = \mathsf{Greedy}\,(Q) \cup \mathtt{box}$.

28

Figure 14: ADD gadget $G = (a, b, c, d, e)$ from the proof of Lemma 6.6, where $\square_{ab}$ is unsatisfied and $\square_G$ is partitioned into L and R.
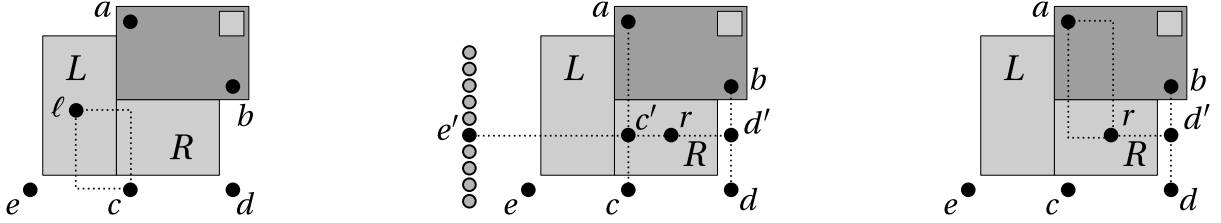

Figure 15: Illustration of the proof of Lemma 6.6. **(left)** Suppose that L is not empty but R is empty. Then $\square_{lc}$ is unsatisfied and $c.y < b.y$ which contradicts the choice of a and b. **(middle)** If $c', d' \in Q \cup \texttt{box}$, then (c', d', e') contradicts the choice of (c, d, e) where $e.y$ is maximal. **(right)** If $c' \notin Q \cup \texttt{box}$, then $\square_{ar}$ is unsatisfied and contradicts the choice of a and b. If $d' \notin Q \cup \texttt{box}$, we reach a contradiction similarly.

(ii) $Q \cup \texttt{box}$ *is satisfied iff* Q *is satisfied.*

Lemma 6.6. *Let $Q \subseteq [n] \times [n]$ be an arbitrary point set. There is no* ADD *gadget in* $(Q \cup \texttt{box})$ *iff* Q *is satisfied.*

Proof. One direction is trivial. Suppose there is an ADD gadget $G = (a, b, c, d, e)$ in $(Q \cup \texttt{box})$. Assume w.l.o.g. that $e.x < d.x$. Let $a' \in (Q \cup \texttt{box})$ be a point such that $a'.y = a.y$ and $a.x \le a'.x \le b.x$ where $a'.x$ is maximal. Let $b' \in Q \cup \texttt{box}$ be a point such that $b'.x = b.x$ and $b.y \le b'.y \le a.y$ where $b'.y$ is maximal. Then $\square_{a'b'}$ is unsatisfied because $\square_G \cap Q = \emptyset$ and $\langle b.x, a.y \rangle \notin Q$. Hence $(Q \cup \texttt{box})$ is unsatisfied, and so is Q by Proposition 6.5(ii).

Now, suppose that Q is unsatisfied, and thus, $(Q \cup \texttt{box})$ is unsatisfied by Proposition 6.5(ii). Let $a, b \in (Q \cup \texttt{box})$ be such that $\square_{ab}$ is unsatisfied and $b.y$ is minimal. Assume by symmetry that $a.x < b.x$. Let $c, d, e \in (Q \cup \texttt{box})$ be such that the relative positions of a, b, c, d, e are according to Figure 11. When there is more than one choice, we first maximize $e.y$, and then maximize $e.x$. We know that such c, d, e exist because $\langle 0, 0 \rangle, \langle a.x, 0 \rangle, \langle b.x, 0 \rangle \in \texttt{box}$. (This is the reason we consider $Q \cup \texttt{box}$ and not just Q.) We claim that $G = (a, b, c, d, e)$ is an ADD gadget in $Q \cup \texttt{box}$.

By construction (1) $\{a, b, c, d, e\} \subseteq (Q \cup \texttt{box})$, (2) the relative positions of a, b, c, d, e are according to Figure 11, and (3) $\langle b.x, a.y \rangle \notin (Q \cup \texttt{box})$ as $\square_{ab}$ is unsatisfied. It remains to show that $\square_G$ is empty (i.e. contains no point from $Q \cup \texttt{box}$). First note that $\square_G \cap \square_{ab}$ is empty because $\square_{ab}$ is unsatisfied. Partition $\square_G \setminus \square_{ab}$ into disjoint parts L and R, where for each point $p \in \square_G \setminus \square_{ab}$, we have $p \in L$ if $p.x < c.x$, and $p \in R$ otherwise. (See Figure 14.) Suppose $\square_G \setminus \square_{ab}$ is not empty. Then we claim that R is not empty.

Claim 6.7. If $\square_G \setminus \square_{ab}$ contains a point in $Q \cup \texttt{box}$, then R contains a point in $Q \cup \texttt{box}$.

Proof. Suppose for contradiction that there is a point in L but R is empty. Observe that (1) $\{\langle i, e.y \rangle \mid e.x < i < c.x\}$ is empty because e is such that $e.x$ is maximal, and (2) $\{\langle c.x, i \rangle \mid e.y < i < a.y\}$ is empty because $R \cup \square_{ab}$ is empty. Consider a rightmost, bottommost point $\ell \in L \cap (Q \cup \texttt{box})$. We have that $\square_{\ell c}$ is unsatisfied and $c.y < b.y$. See Figure 15 (left). But we chose a and b such that $\square_{ab}$ is unsatisfied and $b.y$ is minimal. This is a contradiction. ∎

By the above claim, let $r \in R \cap (Q \cup \texttt{box})$ where $r.y$ is maximal. We will show a contradiction. (See Figure 15.)

Claim 6.8. Either $c' = \langle a.x, r.y \rangle \notin (Q \cup \texttt{box})$ or $d' = \langle b.x, r.y \rangle \notin (Q \cup \texttt{box})$.

Proof. Suppose otherwise. See Figure 15 (middle). Then there are points $e' = \langle 0, r.y \rangle, c', d' \in (Q \cup \texttt{box})$, such that (a, b, c', d', e') are in the desired configuration, but $e'.y > e.y$. This contradicts the choice of c, d, e. ∎

By the above claim, there are two cases. If $\langle a.x, r.y \rangle \notin (Q \cup \texttt{box})$, then we choose r such that $r.x$ is minimal (among those with the same $r.y$). Let $a' \in (Q \cup \texttt{box})$ be such that $a'.x = a.x$ and $r.y < a'.y \leq a.y$ where $a'.y$ is minimal (possibly $a' = a$). We have that $\square_{a'r}$ is unsatisfied and, moreover, $r.y < b.y$. This contradicts the choice of a and b. See Figure 15 (right). If $\langle b.x, r.y \rangle \notin (Q \cup \texttt{box})$, then we choose r such that $r.x$ is maximal (among those with the same $r.y$). Then the argument is symmetric.

We conclude that R is empty, and therefore, $\square_G$ must be empty. ∎

The above lemmas already suggest the definition of Greedy_{nondet}.

Definition 6.9 (Non-deterministic Greedy). *Given a set $P \subseteq [n] \times [n]$, Greedy_{nondet} works as follows. Initially, set $Q = (P \cup \texttt{box})$. Repeatedly find an* arbitrary ADD *gadget $G = (a, b, c, d, e)$ in Q. Then, fill G (i.e. add point $f = \langle b.x, a.y \rangle$ to Q). Let Q be the final point set where there is no* ADD *gadget in Q. Let $\mathsf{Greedy}_{nondet}(P) = (Q \setminus \texttt{box})$.*

Now we show that Greedy_{nondet} and Greedy are identical.

Theorem 6.10. *For every point set $P \subset [n] \times [n]$, we have $\mathsf{Greedy}_{nondet}(P) = \mathsf{Greedy}(P)$.*

Proof. We proceed via two claims: (1) $\mathsf{Greedy}_{nondet}(P)$ is satisfied, and (2) $\mathsf{Greedy}_{nondet}(P) \subseteq \mathsf{Greedy}(P)$. These claims together imply that $\mathsf{Greedy}_{nondet}(P) = \mathsf{Greedy}(P)$, because $\mathsf{Greedy}(P)$ is a *minimally* satisfied set in the sense that every set Q such that $P \subseteq Q \subset \mathsf{Greedy}(P)$ is unsatisfied. (To see this, look at the first row i where Q and $\mathsf{Greedy}(P)$ differ. By definition, the points that Greedy adds in row i are *necessary and sufficient* to satisfy all rectangles defined by points in rows $1, \ldots, i-1$ and $P_{y=i}$. As $Q_{y=i}$ is a proper subset of $\mathsf{Greedy}(P)$ in row i, it leaves some rectangle unsatisfied, which remains unaffected by the points in rows $i+1, \ldots, n$.)

For claim (1), by the definition of Greedy_{nondet}, there is no ADD gadget in $\mathsf{Greedy}_{nondet}(P) \cup \texttt{box}$. By Lemma 6.6, $\mathsf{Greedy}_{nondet}(P) \cup \texttt{box}$ is satisfied, and so is $\mathsf{Greedy}_{nondet}(P)$ by Proposition 6.5(ii).

For claim (2), by Proposition 6.5(i), it suffices to show that $\mathsf{Greedy}_{nondet}(P) \cup \texttt{box} \subseteq \mathsf{Greedy}(P \cup \texttt{box})$. We show this by induction on the steps of Greedy_{nondet}. For the base case, the initial point set is $(P \cup \texttt{box}) \subseteq \mathsf{Greedy}(P \cup \texttt{box})$. For the inductive step, let Q be the points added by Greedy_{nondet} so far. We have $Q \subseteq \mathsf{Greedy}(P \cup \texttt{box})$ by induction, and so $\mathsf{Greedy}(Q) = \mathsf{Greedy}(P \cup \texttt{box})$ by

Proposition 6.4. Let f be the new point added to Q by Greedy_{nondet}. Note that f is added because of an $\mathtt{ADD}$ gadget in Q. Thus, by Lemma 6.3, $f \in \mathsf{Greedy}\,(Q) = \mathsf{Greedy}\,(P \cup \mathtt{box})$. It follows that $Q \cup \{f\} \subseteq \mathsf{Greedy}\,(P \cup \mathtt{box})$. This concludes the proof. ∎

7 Proofs

In this section we present the proofs of Theorem 5.1 and Theorem 5.5 from § 5.

7.1 Reduction to geometric setting

In this subsection we state some results about the connection between the two geometric problems (the star-path problem and the satisfied superset problem). Then, we prove that they imply the main results from § 5.

To prove Theorem 5.1, let $\mathsf{Smooth}_{\mathtt{path}}$ be an algorithm for the star-path problem obtained from the smooth heap algorithm using Theorem 3.2. We have the following key lemma:

Lemma 7.1. $|\mathsf{Greedy}\,(P)| = \Theta(|\mathsf{Smooth}_{\mathtt{path}}(P)|)$ *for every permutation point set* P.

To prove Theorem 5.5, we show the analogous statement in geometric setting.

Lemma 7.2. *For every algorithm* $\mathcal{A}_{\mathtt{path}}$ *for the star-path problem, there is an algorithm* $\mathcal{A}_{\mathtt{sat}}$ *for the satisfied superset problem, such that* $|\mathcal{A}_{\mathtt{sat}}(P)| = \Theta(|\mathcal{A}_{\mathtt{path}}(P)|)$ *for every permutation* P. *Moreover,* $(\mathcal{A}_{\mathtt{sat}}(P))^r$, *i.e. the reverse of the output of* $\mathcal{A}_{\mathtt{sat}}$, *is an insertion-compatible superset of* P^r.

Assuming the two lemmas above, we can prove Theorem 5.1 and Theorem 5.5.

Proof. We use $f \approx g$ to denote $f = \Theta(g)$. Fix some permutation X on $[n]$.

(Lemma 7.1 implies Theorem 5.1):

$$
\begin{aligned}
|\mathsf{GreedyFuture}\,(X')| &= |\mathsf{Greedy}\,(P^{X'})| & \text{by Theorem 2.8}\\
&\approx |\mathsf{Smooth}_{\mathtt{path}}(P^{X'})| & \text{by Lemma 7.1}\\
&= |\mathsf{Smooth}\,(X)|. & \text{by Theorem 3.2}
\end{aligned}
$$

(Lemma 7.2 implies Theorem 5.5): Given a stable heap algorithm $\mathcal{A}_{\mathtt{stable}}$ from Theorem 5.5, there is $\mathcal{A}_{\mathtt{path}}$ where $\mathcal{A}_{\mathtt{stable}}(X) = \mathcal{A}_{\mathtt{path}}(P^{X'})$ by Theorem 3.2. Plugging $\mathcal{A}_{\mathtt{path}}$ into Lemma 7.2, there is an algorithm $\mathcal{A}_{\mathtt{sat}}$ for the satisfied superset problem, where $|\mathcal{A}_{\mathtt{sat}}(P^{X'})| = \Theta(|\mathcal{A}_{\mathtt{path}}(P^{X'})|)$, with $(\mathcal{A}_{\mathtt{sat}}(P^{X'}))^r$ being an insertion-compatible satisfied superset of $(P^{X'})^r$. By treating $(P^{X'})^r$ as the input, there is an algorithm $\mathcal{A}'_{\mathtt{sat}}$ such that $\mathcal{A}'_{\mathtt{sat}}((P^{X'})^r) = \mathcal{A}_{\mathtt{sat}}(P^{X'})$ and $\mathcal{A}'_{\mathtt{sat}}((P^{X'})^r)$ is insertion-compatible for its input $(P^{X'})^r$. As $(P^{X'})^r = P^{(X')^r}$, there is an offline BST algorithm $\mathcal{A}_{\mathtt{bst}}$ in insert-only mode (sorting mode) such that $\mathcal{A}_{\mathtt{bst}}((X')^r) = \mathcal{A}'_{\mathtt{sat}}(P^{(X')^r})$, by Theorem 2.5. To summarize, we have:

$$\begin{aligned}
|\mathcal{A}_{\mathtt{bst}}((X')^r)| &= |\mathcal{A}'_{\mathtt{sat}}(P^{(X')^r})| && \text{by Theorem 2.5} \\
&= |\mathcal{A}_{\mathtt{sat}}(P^{X'})| \\
&\approx |\mathcal{A}_{\mathtt{path}}(P^{X'})| && \text{by Lemma 7.2} \\
&= |\mathcal{A}_{\mathtt{stable}}(X)|. && \text{by Theorem 3.2}
\end{aligned}$$

It remains to prove **Lemma 7.1** and **Lemma 7.2**. We prove them using a similar approach, but with different key ingredients in the proof. The rest of the section is devoted to these proofs.

7.2 The common framework

From now on, we fix an algorithm $\mathcal{A}_{\mathtt{path}}$ for the star-path problem and a permutation P. We describe two different ways of producing a satisfied superset Q of P such that $|Q| = \Theta(|\mathcal{A}_{\mathtt{path}}(P)|)$. The first method works for arbitrary $\mathcal{A}_{\mathtt{path}}$. We will show that Q is insertion-compatible for P^r which gives Lemma 7.2. The second method works only for $\mathcal{A}_{\mathtt{path}} = \mathsf{Smooth}_{\mathtt{path}}$. In this case we will show that $Q = \mathsf{Greedy}(P)$ which gives Lemma 7.1.

In this subsection, we describe a common framework for constructing a satisfied set Q. Then, in the next subsections, we show how to construct Q for general $\mathcal{A}_{\mathtt{path}}$ and for $\mathsf{Smooth}_{\mathtt{path}}$.

Let $P_0 = P \cup \{\langle 0,0 \rangle\}$. Recall the definitions of $star(P)$ and $path(P)$ near Definition 3.1 (they are two particular monotone trees whose nodes are P_0). Initially, we set $Q = Q^{init}$ where the definition of Q^{init} will be described differently in §7.3 and §7.5. For each link that $\mathcal{A}_{\mathtt{path}}$ performs to transform $star(P)$ to $path(P)$, we add a constant number of points to Q while maintaining certain invariants. Once the execution of $\mathcal{A}_{\mathtt{path}}$ is finished (producing $path(P)$), the invariant will imply that Q is satisfied. To state the invariants precisely, we need some definitions.

Let T be the current tree of $\mathcal{A}_{\mathtt{path}}$. For convenience, we write the set of nodes of T as $P_0 \overset{\text{def}}{=} \{u_0, \ldots, u_n\}$ where $u_0 = \langle 0,0 \rangle$ and u_i is the unique point in $P_{y=i}$. Let $Q \supseteq Q^{init}$ be the current point set. For every i, we write $Q_{u_i} = Q_{y=i}$. For every node u we define $I(u) \subset [n]$ as *the interval of u* and we write $I(u) \overset{\text{def}}{=} (I(u).\min, I(u).\max)$. We call I the *interval function*. The precise definition of I, which is a key element of the proofs, will be given later.

The invariants are the following (illustrated in Figure 16).

- **Invariant 1 (Intervals respect tree-structure):** The set of intervals $I(u)$ for all nodes $u \in T$ forms a laminar family whose structure is exactly T. That is, let u be an arbitrary node in T with children $v_1, \ldots, v_k$. Then $I(v_j) \subseteq I(u)$ for all j, and $I(v_j) \cap I(v_{j'}) = \emptyset$ for all $j \neq j'$.

- **Invariant 2 (Tree-structure respects satisfiability):** If u is an ancestor of v in T, then for every $p \in Q_u$ and $q \in Q_v$ the rectangle $\square_{pq}$ is satisfied.

- **Invariant 3 (No points between a node and its parent's interval):** For every node v with parent u, we have $\{\langle v.x, i \rangle \mid u.y < i < v.y\} \cap Q = \emptyset$.
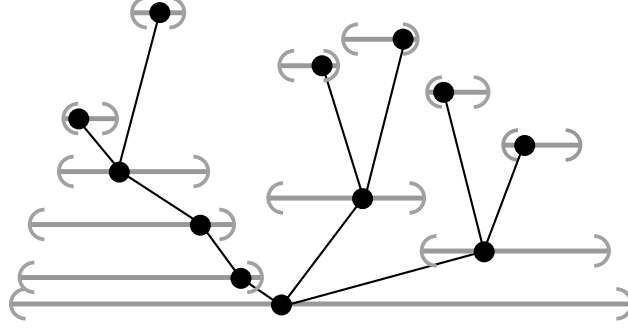
Figure 16: Illustration of Invariant (1). Horizontal segments indicate intervals of nodes.

- **Invariant 4 (Adding interval endpoints):** For every node v with parent u,

$$\langle I(v).\min, v.y\rangle, \langle I(v).\max, v.y\rangle \quad \in \quad Q \text{ (endpoints of } I(v)\text{), and}$$
$$\langle I(v).\min, u.y\rangle, \langle I(v).\max, u.y\rangle \quad \in \quad Q \text{ (endpoints of } I(v) \text{ "projected" to row of parent } u\text{).}$$

We make a crucial observation.

Proposition 7.3. *At the end, when $T = path(P)$, if the invariants hold, then Q is a satisfied set.*

Proof. Let p and q be two arbitrary points in Q. W.l.o.g. $p \in Q_{u_i}$ and $q \in Q_{u_j}$ where $i \leq j$. As $T = path(P)$, u_i is an ancestor of u_j. By Invariant (2), $\square_{pq}$ is satisfied. ∎

7.3 The general transformation

Let $\texttt{base} = \{\langle i, 0\rangle \mid i \in [n]\}$. We define $Q^{init} \stackrel{\text{def}}{=} P \cup \texttt{base}$, and define the interval function as follows. For each node u_i,

$$I(u_i) \stackrel{\text{def}}{=} (\min_{q \in Q_{u_i}} q.x, \max_{q \in Q_{u_i}} q.x). \tag{1}$$

In words, $I(u_i)$ is the open interval between the leftmost and rightmost "touched" positions in the i-th row. With this definition of Q^{init} and I, we observe the following.

Lemma 7.4. *The invariants hold initially.*

Proof. Initially, $Q = (P \cup \texttt{base})$, and $T = star(P)$. Observe that $I(u_0) = (1, n)$ and $I(u_i) = (i, i) = \emptyset$ for all $i \in [n]$ by definition of Q_{init}. Moreover, these intervals form a laminar family whose structure is exactly $star(P)$, so Invariant (1) holds. Next, for every $i \in [n]$, u_i has only u_0 as an ancestor. For every point $p \in Q^{init}_{y=0}$ and $q \in Q^{init}_{y=i}$, we have that $\square_{pq}$ is satisfied as $Q^{init}_{y=0} = \texttt{base} = \{\langle i, 0\rangle \mid 1 \leq i \leq n\}$. Thus, Invariant (2) holds. Invariants (3) and (4) hold trivially. ∎

Next, we specify how to add points to Q for each link performed by $\mathcal{A}_{\texttt{path}}$ and show that the invariants are maintained.

Lemma 7.5. *Let T be the current tree and Q be the current point set. Suppose that the invariants hold for T and Q. Let T' be obtained from T by a stable link operation. We can add **one** or **two** new points to Q and obtain Q' such that the invariants hold for T' and Q'.*

Figure 17: A step in the general transformation and the maintenance of invariants. **(left)** Before the transformation: a and b are children of u in T. Crosses mark locations of newly added points d and e. **(right)** After the transformation: b is a child of a in T'. Intervals shown as horizontal segments, respecting tree-structure (Invariant (1)). Shaded rectangles indicate empty areas (Invariant (3)). Smaller dots indicate interval endpoints and their projections (Invariant (4)), some points omitted for clarity. Non-emptiness of newly created rectangles (Invariant (2)) is guaranteed by newly added points and the points of Invariant (4).

Proof. Suppose that a and b are neighboring nodes in T with the same parent u, and T' is obtained from T by changing the parent of b from u to a. Assume $a.x < b.x$ (the other case is symmetric). Let $Q' = Q \cup \{d, e\}$ where $d = \langle I(b).\min, a.y \rangle$ and $e = \langle I(b).\max, a.y \rangle$ (see Figure 17). We note that it is possible that d is not new, i.e. $d \in Q$ already. But we have the following:

Claim 7.6. e is a new point, i.e. $e \notin Q$. Hence, $1 \le |Q' \setminus Q| \le 2$.

Proof. There are two cases. First, suppose $I(b).\min = I(b).\max = b.x$. Then by Invariant (3), we have that $\{\langle b.x, i \rangle \mid u.y < i < b.y\} \cap Q = \emptyset$. As $e = \langle b.x, a.y \rangle$, so $e \notin Q$. Next, suppose $I(b).\min < I(b).\max$. Then, by Equation (1), for every $p \in Q_a$, we have $p.x \le I(a).\max \le I(b).\min$. As $e.x = I(b).\max$, so $e \notin Q_a$ and hence $e \notin Q$. ∎

It remains to show that the invariants hold for T' and Q'.

For Invariant (1), observe that $I'(x) = I(x)$ for all x, except for $x = a$. As $I(a).\max \le d.x$ by Invariant (1), we have $I'(b) = (d.x, e.x) \subseteq (I(a).\min, e.x) = I'(a)$. For any child c of a, $I'(c).\max = I(c).\max \le I(a).\max \le d.x = I'(b).\min$ (the first inequality holds by Invariant (1)), so we have $I'(b) \cap I'(c) = \emptyset$. As we only change the parent of b (from u to a), all these observations imply that Invariant (1) holds after adding the new points.

For Invariant (2), after linking, a becomes a new ancestor of b and of any descendant of b. It is sufficient to consider rectangles $\square_{pq}$ where $p \in Q'_a$, and $q \in Q'_c$, where c is an ancestor or descendant of a.

First, let c be a descendant of b (possibly b itself). There are three cases: (1) if $p = d$, then $\langle I(b).\min, b.y \rangle \in \square_{pq}$; (2) if $p = e$, then $\langle I(b).\max, b.y \rangle \in \square_{pq}$; (3) if $p \ne d, e$, then $p \in Q_a$ and $q \in Q_c$ and so $d \in \square_{pq}$.

Second, let c be an old descendant of a (possibly a itself). If $p \in \{d, e\}$, then $\langle I(a).\max, a.y \rangle \in \square_{pq}$. (If $p \ne d, e$, then $p \in Q_a$, and $\square_{pq}$ was already satisfied before adding the new points, due to Invariant (2).)

Third, let c be an ancestor of u (possibly u itself). By Invariant (4) we have $d', e' \in Q'_u$, where $d' = \langle d.x, u.y \rangle$, and $e' = \langle e.x, u.y \rangle$, i.e. the projections of the endpoints of $I(b)$ to $I(u)$. One of the points d', e' is contained in every $\square_{pq}$, making it satisfied.

34

As there are no other cases, Invariant (2) holds.

For Invariant (3), as only a becomes a new parent of b, we only need to show that $\{\langle b.x, i\rangle \mid a.y < i < b.y\} \cap Q' = \emptyset$. As we have $\{\langle b.x, i\rangle \mid u.y < i < b.y\} \cap Q = \emptyset$ and $d, e \notin \{\langle b.x, i\rangle \mid u.y < i < b.y\}$, we are done.

For Invariant (4), we need to argue that $\langle I(b).\min, a.y\rangle$, $\langle I(b).\max, a.y\rangle$, $\langle I(a).\max, a.y\rangle$, $\langle I(a).\max, u.y\rangle \in Q'$. The first three are true by construction (these are exactly d, e, and e, respectively). The fourth is true, because $\langle I(a).\max, u.y\rangle = \langle I(b).\max, u.y\rangle \in Q \subseteq Q'$ by Invariant (4) for T and Q. ∎

We conclude with the proof of Lemma 7.2.

Proof of Lemma 7.2. Given $\mathcal{A}_{\texttt{path}}$ and P, we let $\mathcal{A}_{\texttt{sat}}$ be simply the algorithm that returns Q as described above. By Lemma 7.5, for each link of $\mathcal{A}_{\texttt{path}}$, one or two new points are added to Q. So $|Q| = \Theta(|\mathcal{A}_{\texttt{path}}(P)|)$. By Proposition 7.3 and the fact that $Q \supseteq (P \cup \texttt{base})$, Q is a satisfied superset of P. Lastly, it is easy to see from Figure 17 that all points added to Q are "below" P, i.e. for every $q \in Q_{x=i}$ we have $q.y \leq (p_{x=i}).y$, where $p_{x=i}$ is the unique point in $P_{y=i}$. (To see this, consider for contradiction the first time a point would be added above a point of P.) In other words, Q is an insertion-compatible superset of P^r. ∎

7.3.1 The general transformation to a family of BST algorithms

In this subsection we sketch the idea of generating multiple (different) BST executions.

The argument goes in a similar way as in §7.3, except that Invariant (3) is no longer maintained. Every time $\mathcal{A}_{\texttt{path}}$ performs a link operation and Lemma 7.5 is invoked, there is an alternative way to add points to Q (see Figure 18) such that Invariants (1), (2) and (4) are maintained, by a similar argument. However, with this alternative, it is possible that the points d and e are already in the point set Q. It follows that $|Q| = O(|\mathcal{A}_{\texttt{path}}(P)|)$ (instead of $|Q| = \Theta(|\mathcal{A}_{\texttt{path}}(P)|)$ as before). Note also that it is no longer the case that all points added to Q are below P.
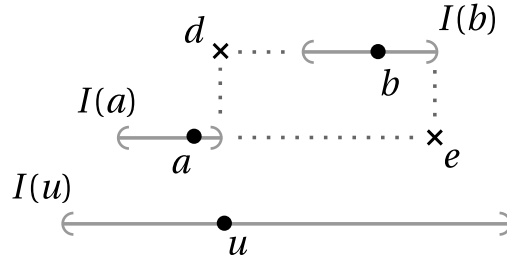

Figure 18: An alternative step in the general transformation. After linking b to a, the new points d and e (marked as crosses) are added. Dotted lines show horizontal or vertical alignment.

The important observation is that, we can choose, independently for each link operation, whether to add points as shown in Figure 17 or as shown in Figure 18. Thus, we obtain infinitely many ways of constructing Q (as n goes to infinity).

Applied to Smooth, we obtain various (non-trivial) satisfied supersets, whose costs are at most (up to constant factors) the conjectured optimal cost of Greedy. The surprising aspect of this result

is that small local changes tend to propagate and affect the future behavior of algorithms in ways that are otherwise difficult to analyse.

For instance, it appears difficult even to show that Greedy (in the geometric view) is competitive with itself, executed on the same input point set, but with *one additional point* added at time zero. Understanding the effect of different initial/intermediate states is a bottleneck in our current understanding of Greedy, and indeed, the BST model. (See also [CGK+15b].)

7.4   Structure of smooth heap after linking all two-neighbor nodes

Before proving Theorem 5.1 in §7.5, we make some observations about the tree that Smooth$_{\texttt{path}}$ maintains.

Recall the non-deterministic view of the smooth heap described in §4. In sorting mode, the smooth heap performs n *rounds*, where each round corresponds to one extract-min operation. In each round, as long as the current root has more than one child, we repeatedly find an *arbitrary* local maximum x among the children of the root, and link x to the neighbor with a larger key (if there are more than one neighbors).

To facilitate the proof, we assume that, whenever possible, a local maximum with two neighbors is chosen. If such a node does not exist, then we choose a local maximum with one neighbor.

We translate this process to describe how Smooth$_{\texttt{path}}$ works as follows. Given a permutation P, Smooth$_{\texttt{path}}$ works in n *rounds*. In the beginning of the r-th round $(u_0, \ldots, u_{r-1})$ form a path in the current monotone tree. Let $v_1, \ldots, v_k$ be the current children of u_{r-1} ordered from left to right. We call a node v_i a *local maximum* (among the children of u_{i-1}) if $(v_i).y > (v_{i-1}).y$ (if v_{i-1} exists) and $(v_i).y > (v_{i+1}).y$ (if v_{i+1} exists). As long as $k > 1$, we choose an arbitrary *local maximum* v_i, then link v_i to its higher neighbor. Again, we assume that, if possible, a local maximum with two neighbors is chosen, before choosing a local maximum with only one neighbor. The following observations are immediate.

Proposition 7.7. *Let $v_1, \ldots, v_k$ be the current children of u_{r-1} and suppose that there is no local maximum node with two neighbors. Then:*

- *$v_1, \ldots, v_k$ form a V-shape, i.e. for some i:*

$$v_1.y > \cdots > v_{i-1}.y > v_i.y < v_{i+1}.y < \cdots < v_k.y.$$

- *Only v_1 and v_k are local maxima. Both have one neighbor.*

- *After linking v_1 or v_k, there is still no local maximum with two neighbors.*

- *At the end of the round, v_j is linked to v_{j+1} for all $j < i$, and v_j is linked to v_{j-1} for all $j > i$.*

By this observation, it follows that in each round, once we have linked local maxima with two neighbors until this is no longer possible, all the remaining links will be of one-neighbor local maxima.

Proposition 7.8. *For each round, the sequence of link operations can be divided into two consecutive phases: a* two-neighbor phase *with links involving a local maximum with two neighbors, and a* one-neighbor phase *consisting of the remaining link operations.*

7.5 The Smooth-Greedy transformation

In this subsection, we finally prove Theorem 5.1. The approach is similar to the one in §7.3. The differences are in (1) the definition of Q^{init} and I, and (2) in the method of updating Q and I for each link performed by $\mathsf{Smooth}_{\texttt{path}}$.

Let $\texttt{box} = \{\langle i, 0 \rangle \mid i \in [n]\} \cup \{\langle 0, i \rangle \mid 0 \le i \le n\} \cup \{\langle n+1, i \rangle \mid 0 \le i \le n\}$. Let $Q^{init} = P \cup \texttt{box}$. Unlike in §7.3, we do not have a closed form formula for the interval function I at every time. We instead describe how it is updated after each link. Initially, we set $I(u_0).\min = 0$, $I(u_0).\max = n+1$, and $I(u_i).\min = I(u_i).\max = i$, for $i > 0$.

We keep invariants (1),(2), and (4) from §7.2 unchanged. We strengthen Invariant (3) to Invariant (3')[22]. We also introduce one new invariant.

- **Invariant 3' (No points added below nodes):** For every point $p \in Q \setminus \texttt{box}$ and node v, if $p.x = v.x$, then $p.y \ge v.y$.

- **Invariant 5 (Intervals include all non-box points):** For every node v, $I(v) \supseteq (\min_{q \in (Q_v \setminus \texttt{box})} q.x, \max_{q \in (Q_v \setminus \texttt{box})} q.x)$.

Observe that earlier, Invariant (5), although not stated explicitly, was true *with equality*, by the definition of the intervals. Here we have a relaxed statement, to allow for box points being included in the intervals.

Lemma 7.9. *The invariants hold initially.*

Proof. Observe that the initial intervals form a laminar family whose structure is exactly $star(P)$, so Invariant (1) holds. Next, for every $i \in [n]$, u_i has only u_0 as an ancestor. For every point $p \in Q_{y=i}^{init}$ and $q \in Q_{y=i}^{init}$, we have that $\square_{pq}$ is satisfied as $Q_{y=0}^{init} \supseteq \{\langle i, 0 \rangle \mid 0 \le i \le n+1\}$. So Invariant (2) holds. Invariants (3'), (4), and (5) hold trivially. ∎

Next, we specify how to add points to Q and update the interval function I. There are two cases: whether we are in the two-neighbor phase (Lemma 7.10) or in the one-neighbor phase (Lemma 7.14). We argue these cases separately in the following subsections.

7.5.1 Adding points in two-neighbor phases

During the the two-neighbor phase, let T, Q and I be the current monotone tree, point set, and interval function respectively. Suppose that the invariants hold for T, Q and I. We argue that the invariants hold after just one link. Let T' be obtained from T by a stable link operation performed by $\mathsf{Smooth}_{\texttt{path}}$. We first describe how we update Q and I to obtain Q' and I'. Then, we show that the invariants hold. Finally, we show that the number of new points in Q is between 1 and 3. By applying this argument repeatedly for each link until the phase ends, we conclude:

Lemma 7.10. *For the two-neighbor phase of every round, the invariants are maintained, and the number of points added into Q is proportional to the number of links performed in the phase.*

---

[22]Invariant (3) states that for every node v with parent u, $\{\langle v.x, i \rangle \mid u.y < i < v.y\} \cap Q = \emptyset$. Invariant (3') states more strongly that for every node v, $\{\langle v.x, i \rangle \mid 0 < i < v.y\} \cap Q = \emptyset$.
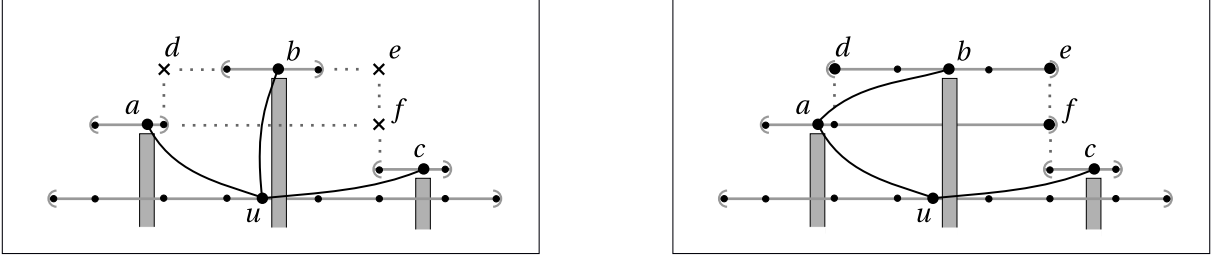
Figure 19: A two-neighbor link and the maintenance of invariants. **(left)** Before the transformation: a, b, and c are children of u in T. Crosses mark locations of newly added points d, e, and f. **(right)** After the transformation: b is a child of a in T'. Intervals shown as horizontal segments, respecting tree-structure (Invariant (1) and Invariant (5)); labels omitted. Shaded rectangles indicate empty areas (Invariant (3')). Smaller dots indicate interval endpoints and their projections (Invariant (4)). Dotted lines indicate horizontal and vertical alignment, some lines omitted for clarity. Non-emptiness of newly created rectangles (Invariant (2)) is guaranteed by newly added points and the points of Invariant (4).

Updating Q and I after one link. Suppose that we are in the r-th round. Let b be the local maximum child of u_{r-1}, with two neighbors. Let a and c be the left and right neighbors of b respectively. By symmetry, assume $a.y > c.y$. So b is linked to a. To update Q and I, we set $Q' = Q \cup \{d, e, f\}$ where

$$d = \langle I(a).\max, b.y \rangle,$$
$$e = \langle I(c).\min, b.y \rangle,$$
$$f = \langle I(c).\min, a.y \rangle.$$

We set

$$I'(a) = (I(a).\min, I(c).\min),$$
$$I'(b) = (I(a).\max, I(c).\min)$$

and $I'(x) = I(x)$ for all nodes $x \neq a, b$. See Figure 19.

Invariants maintained after one link. For Invariant (1), observe that $I'(b) \subseteq I'(a)$, and for every child a' of a, $I'(b) \cap I'(a') = \emptyset$ because $I'(a').\max = I(a').\max \leq I(a).\max = I'(b).\min$. Similarly, $I'(a) \subseteq I'(u)$ and $I'(b) \subseteq I'(u)$. As there are no other changes in the monotone tree T (other than a gaining b as a child), Invariant (1) is maintained.

For Invariant (2), after linking, a becomes a new ancestor of b and of every descendant of b. This introduces several new pairs of (p, q) where we need to check if $\square_{pq}$ is satisfied. It is clear that this holds if either $p \in \texttt{box}$ or $q \in \texttt{box}$. The other cases are as follows:

1. If $p = d$ and q is below p (i.e. $q.y < p.y$), then $\langle I(a).\max, a.y \rangle \in \square_{pq}$. If $p = d$ and q is above p, then $\langle I(b).\min, b.y \rangle \in \square_{pq}$. Note that $\langle I(a).\max, a.y \rangle, \langle I(b).\min, b.y \rangle \in Q$ by Invariant (4).

2. If $p = e$ and q is below p, then $f \in \square_{pq}$. If $p = e$ and q is above p, then $\langle I(b).\max, b.y \rangle \in \square_{pq}$. Note that $\langle I(b).\max, b.y \rangle \in Q$ by Invariant (4).

38

3 If $p = f$ and q is below p, then $\langle f.x, u.y \rangle \in \square_{pq}$. If $p = f$ and q is above p, then $e \in \square_{pq}$. Note that $\langle I(b). \max, b.y \rangle \in Q$ by Invariant (4).

4 If $p \in Q_a \setminus \texttt{box}$ is an old point, then $\langle I(a). \max, a.y \rangle \in \square_{pq}$ where $q \in Q_b$.

The aboves cases exhaust all the cases because only d, e, f are new points, and only a becomes a new ancestor of b. This implies that Invariant (2) holds.

For Invariant (3'), we show that the points d, e, f are not below any node of the monotone tree. Let v_d be a node such that $v_d.x = I(a). \max = d.x$. Now, by Invariant (4), we have $\langle I(a). \max, a.y \rangle \in Q$ and actually we have $\langle I(a). \max, a.y \rangle \in Q \setminus \texttt{box}$. By Invariant (3'), $v_d.y \leq a.y < b.y = d.y$. So d does not violate Invariant (3'). Next, let v_e be a node such that $v_e.x = I(c). \min = e.x = f.x$. Again, $\langle I(c). \min, c.y \rangle \in Q \setminus \texttt{box}$ by Invariant (4). Then, Invariant (3') implies $v_e.y \leq c.y < f.y < e.y$. So e and f do not violate Invariant (3') either.

For Invariant (4), as $I'(u) = I(u)$ for $u \neq a, b$ we only need to argue about the endpoints of $I'(a)$ and $I'(b)$. As $I'(a). \min = I(a). \min$, we have $\langle I'(a). \min, a.y \rangle \in Q$ and $\langle I'(a). \min, u.y \rangle \in Q$ by Invariant (4) before adding points. As $I'(a). \max = I(c). \min$, we have $\langle I'(a). \max, a.y \rangle = f \in Q'$ and $\langle I'(a). \max, u.y \rangle = Q$ by Invariant (4) before adding points. Next, we have $\langle I'(b). \min, b.y \rangle = d \in Q'$ and $\langle I'(b). \min, a.y \rangle = \langle I(a). \max, a.y \rangle \in Q$ by Invariant (4) before adding points. Finally, we have $\langle I'(b). \max, b.y \rangle = e \in Q'$ and $\langle I'(b). \max, a.y \rangle = f \in Q'$.

For Invariant (5), only $Q_a \setminus \texttt{box}$ and $Q_b \setminus \texttt{box}$ change. For $Q_a \setminus \texttt{box}$, we have set $I'(a). \max$ to be $f.x$. For $Q_b \setminus \texttt{box}$, $I'(b) = (d.x, e.x)$. So Invariant (5) is maintained as well. This concludes that the invariants hold during the two-neighbor phase.

Counting points added after one link. We first observe the following:

Claim 7.11. If $I(b). \min = I(b). \max = b.x$, then $I(a). \max < b.x < I(c). \min$.

Proof. Recall that $b.y > a.y > c.y$. Suppose that $I(a). \max = b.x$. Then, $\langle I(a). \max, a.y \rangle \in Q$ by Invariant (4) and actually $\langle I(a). \max, a.y \rangle \in Q \setminus \texttt{box}$. As $\langle I(a). \max, a.y \rangle = \langle b.x, a.y \rangle$, Invariant (3') implies that $a.y \geq b.y$ which is a contradiction. So we have $I(a). \max < I(b). \min$. Symmetrically, suppose that $I(c). \min = b.x$. Then $\langle I(c). \min, c.y \rangle \in Q \setminus \texttt{box}$ by Invariant (4). As $\langle I(c). \min, c.y \rangle = \langle b.x, c.y \rangle$, Invariant (3') implies that $c.y \geq b.y$ which is a contradiction. ∎

Now, we get:

Claim 7.12. $I(a). \max < I(c). \min$.

Proof. There are two cases: if $I(b). \min < I(b). \max$, then this is easy; by Invariant (1), $I(a). \max \leq I(b). \min < I(b). \max \leq I(c). \min$. Else, Claim 7.11 directly implies the current claim. ∎

We claim that we add at least one and at most three new point into Q:

Claim 7.13. $f \notin Q$. Hence, $1 \leq |Q' \setminus Q| \leq 3$.

Proof. As $I(a). \max < I(c). \min$ and $f.y = a.y$, by Invariant (5), we have that $f \notin Q_a \setminus \texttt{box}$, and so $f \notin Q$. ∎
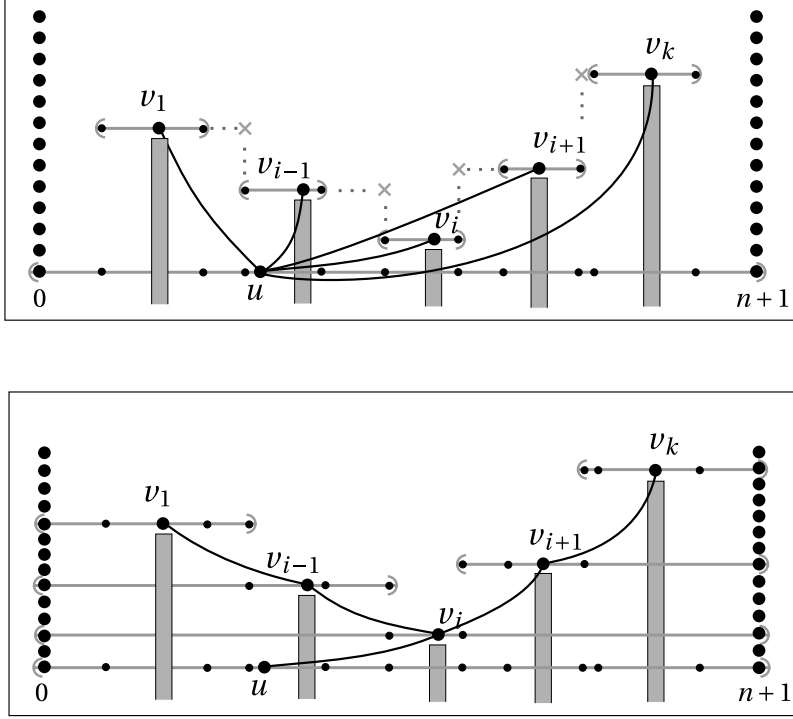
Figure 20: Links of a one-neighbor phase and the maintenance of invariants. **(above)** Before the transformation: $v_1, \ldots, v_k$ are children of u in T. Crosses mark locations of newly added points. **(below)** After the transformation: $v_1, \ldots, v_{i-1}$ and $v_{i+1}, \ldots, v_k$ form two paths rooted at v_i in T'. Intervals shown as horizontal segments, respecting tree-structure (Invariant (1) and Invariant (5)); labels omitted. Observe that intervals may extend to include box points. Shaded rectangles indicate empty areas (Invariant (3')). Smaller dots indicate interval endpoints and their projections (Invariant (4)). Dotted lines indicate horizontal and vertical alignment, some lines omitted for clarity. Non-emptiness of newly created rectangles (Invariant (2)) is guaranteed by newly added points and the points of Invariant (4). Dots on positions $x = 0$ and $x = n+1$ are points from `box`.

7.5.2 Adding points in one-neighbor phases

Before the first link in the one-neighbor phase, let T, Q, I be the current tree, point set and interval function respectively, and assume that the invariants hold for T, Q, and I. Let T' be obtained from T at the end of the one-neighbor phase.

After the entire phase, we show how to update Q and I to obtain Q' and I', and we argue that the invariants hold. Finally, we bound the total number of points added and the links performed over all one-neighbor phases. This subsection is for proving the following.

Lemma 7.14. *For each one-neighbor phase of every round, the invariants are maintained. The total number of link operations and the total number of points newly added to Q in all one-neighbor phases over n rounds are both at most $2n$.*

Update Q and I after one phase. Suppose that we are at the beginning of the one-neighbor phase of the r-th round. Let $v_1, \ldots, v_{k+1}$ be the current children of u_{r-1} ordered from left to

right. By Proposition 7.7, we obtain T' by performing k links to the current local maximum with one-neighbor (i.e. the current rightmost or leftmost child). There is some i such that, at the end of the round, v_j is linked to v_{j+1} for all $j < i$, and v_j is linked to v_{j-1} for all $j > i$.

For $j < i$, set

$$Q' = Q \cup \langle I(v_{j+1}). \min, v_j.y \rangle \text{ and } I'(v_j) = (0, I(v_{j+1}). \min).$$

For $j > i$, set

$$Q' = Q \cup \langle I(v_{j-1}). \max, v_j.y \rangle \text{ and } I'(v_j) = (I(v_{j-1}). \max, n + 1).$$

For $j = i$, we do not add points, and we set $I'(v_j) = (0, n + 1)$. See Figure 20. Note that we do not claim that all points that we include are new. That is, possibly, $\langle I(v_{j+1}). \min, v_j.y \rangle \in Q$ for $j < i$ and $\langle I(v_{j-1}). \max, v_j.y \rangle \in Q$ for $j > i$. But we do have that the number of new points is bounded.

Proposition 7.15. *In each one-neighbor phase, the number of* new *points added is at most the number of link operations in the phase.*

Invariants maintained after one phase. For each of the invariants, we only argue about j where $j < i$ or $j = i$. The other case is symmetric.

For Invariant (1), it is enough to show that, for $j < i$, $I'(v_j) \subseteq I'(v_{j+1})$ and $I'(v_j) \cap I'(x) = \emptyset$ for any other child x of v_{j+1}. First, $I'(v_j) \subseteq I'(v_{j+1})$ because if $j+1 = i$, then $I'(v_{j+1}) = (1, n)$ (i.e. the entire range), and otherwise $j + 1 < i$, then $I'(v_j) = (1, I(v_{j+1}). \min) \subseteq (1, I(v_{j+2}). \min) = I'(v_{j+1})$. Next, we have $I'(v_j). \max = I(v_{j+1}). \min \leq I(x). \min = I'(x). \min$ for any other child x of v_{j+1} where the inequality holds by Invariant (1).

For Invariant (2), we only argue about points outside box, as for every $p \in$ box and every q, $\square_{pq}$ is satisfied. Consider, for every $j < i$, $p \in Q'_{v_j} = Q_{v_j} \cup \{\langle I(v_{j+1}). \min, v_j.y \rangle\}$ and $q \in Q'_{v_{j'}}$. If $j' < j$, then there is another point $\langle I(v_j). \min, v_{j-1}.y \rangle \in \square_{pq} \cap Q'$. Else, $j' > j$, and then we have that $\langle I(v_{j+1}). \min, v_j.y \rangle \in \square_{pq} \cap Q'$. For $j = i$, for every $p \in Q'_{v_i} = Q_{v_i}$ and $q \in Q'_{v_{j'}}$ where $j' \neq i$, there is $\langle I(v_i). \min, v_{i-1}.y \rangle \in \square_{pq} \cap Q'$ if $j' < i$ and $\langle I(v_i). \max, v_{i+1}.y \rangle \in \square_{pq} \cap Q'$ if $j' > i$. This implies that Invariant (2) holds.

For Invariant (3'), for every $j < i$, let v' be a node where $v'.x = I(v_{j+1}). \min$. As $\langle I(v_{j+1}). \min, v_{j+1}.y \rangle \in Q$ by Invariant (4) and actually $\langle I(v_{j+1}). \min, v_{j+1}.y \rangle \in Q \backslash$ box, Invariant (3') implies $v'.y \leq v_{j+1}.y < v_j.y$. So the new point $\langle I(v_{j+1}). \min, v_j.y \rangle$ does not violate Invariant (3').

For Invariant (4), for every $j < i$, observe that $\langle I'(v_j). \max, v_j.y \rangle = \langle I(v_{j+1}). \min, v_j.y \rangle \in Q'$ which is newly added, and $\langle I'(v_j). \max, v_{j+1}.y \rangle = \langle I(v_{j+1}). \min, v_{j+1}.y \rangle \in Q$ by Invariant (4). $\langle I'(v_j). \min, v_j.y \rangle, \langle I'(v_j). \min, v_{j+1}.y \rangle \in$ box because $I'(v_j) = 0$. For $j = i$, as $I(v_i). \min = 0$ and $I(v_i). \max = n + 1$, so $\langle I'(v_i). \min, v_i.y \rangle, \langle I'(v_i). \max, v_i.y \rangle \in$ box.

Invariant (5) clearly holds. We conclude that the invariants hold after the one-neighbor phase.

Counting links and points added over all rounds. We show that over all n rounds, the number of link operations performed in all one-neighbor phases is at most $2n$. By Proposition 7.15, it follows that the total number of added points in one-neighbor phases is $2n$ as well.

Let w be a local maximum with one-neighbor v where $w.y > v.y$, and w is linked to v. If $w.x < v.x$, then we say thet v gets a *one-neighbor link from the left*. Else $v.x < w.x$, then we say v gets a *one-neighbor link from the right*.

Claim 7.16. Every node v gets a one-neighbor link from the left at most once, and a one-neighbor link from the right at most once.

Proof. Suppose v gets a one-neighbor link from the left. It must be the case that $I(v).\min > 0$ before linking. Because v has a left neighbor w and $0 < w.x \leq I(w).\max \leq I(v).\min$. However, after linking, we have $I(v).\min = 0$. So v can never get a one-neighbor link from the left again. The case from the right is argued symmetrically. ∎

It follows that there are at most $2n$ links in one-neighbor phases over all n rounds. This concludes the proof of Lemma 7.14.

7.5.3 Smooth$_{\texttt{path}}$ fills $\texttt{ADD}$ gadgets

In order to draw a connection to Greedy , we need the following lemma. Recall the $\texttt{ADD}$ gadget from Definition 6.1. We claim that each added point *fills* some $\texttt{ADD}$ gadget in Q.

Lemma 7.17. *For each new point x added to Q by either Lemma 7.10 or Lemma 7.14, there is an $\texttt{ADD}$ gadget in Q which is filled by x.*

To prove the above, we introduce some notation. For every node v, let $v_{\min} = \langle I(v).\min, v.y \rangle$ and $v_{\max} = \langle I(v).\max, v.y \rangle$ where I is the current interval function before adding points. For every point p, let $v_p = \langle p.x, v.y \rangle$ be the point p "projected" to the row $v.y$. Now, there are two cases.

Two-neighbor phases. Let Q be the current point set during a two-neighbor phase. Suppose that there are three nodes a, b, c, where b is a local maximum and a and c are its left and right neighbors respectively. Assume by symmetry that $a.y > c.y$ and so b is linked to a. Then $Q' = Q \cup \{d, e, f\}$ where $d = \langle I(a).\max, b.y \rangle$, $e = \langle I(c).\min, b.y \rangle$ and $f = \langle I(c).\min, a.y \rangle$. Let u be the parent of a, b, c. We define the following as depicted in Figure 21(a):

$$G_d = (b_{\min}, a_{\max}, u_{b_{\min}}, u_{a_{\max}}, u_{c_{\min}}),$$
$$G_e = (b_{\max}, c_{\min}, u_{b_{\max}}, u_{c_{\min}}, u_{a_{\max}}), \text{ and}$$
$$G_f = (a_{\max}, c_{\min}, u_{a_{\max}}, u_{c_{\min}}, u_{\min}).$$

Claim 7.18. If $x \in \{d, e, f\}$ is newly added, i.e., $x \notin Q$, then G_x is an $\texttt{ADD}$ gadget in Q and x fills G_x.

Proof. Observe that Invariant (4) implies that

$$\{a_{\max}, b_{\min}, b_{\max}, c_{\min}, u_{a_{\max}}, u_{b_{\min}}, u_{b_{\max}}, u_{c_{\min}}, u_{\min}\} \subseteq Q.$$
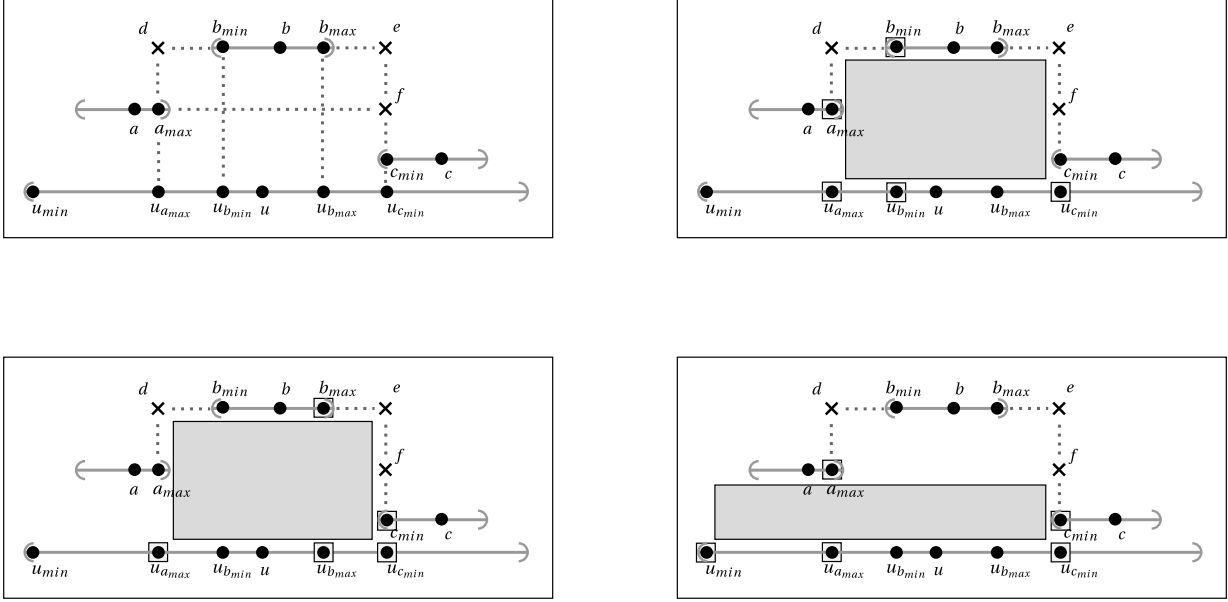
(These are the points of G_d, G_e, G_f.)

Figure 21: (a) Points in the proof of Claim 7.18. **(b)** ADD gadget G_d. **(c)** ADD gadget G_e. **(d)** ADD gadget G_f. Shaded rectangles are empty.

We first prove the case $x = d$. See Figure 21(b). We argue that the five points $b_{\min}$, $a_{\max}$, $u_{b_{\min}}$, $u_{a_{\max}}$, $u_{c_{\min}}$ are distinct by showing that $I(a).\max < I(b).\min < I(c).\min$. To see this, as $d \notin Q$, we get $I(a).\max < I(b).\min$ (otherwise, by Invariant (4), $d \in Q$). Also $I(b).\min < I(c).\min$, because if $I(b).\min = I(b).\max$, then $I(b).\min = b.x < I(c).\min$ by Claim 7.11. Else, $I(b).\min < I(b).\max \le I(c).\min$. Next, it is easy to see that the five points are in the correct relative position for ADD gadgets. It remains to show that $\square_{G_d} \cap Q = \emptyset$. Suppose there is a point $r \in \square_{G_d} \cap Q$. Then, let v_r be a node such that $v_r.y = r.y$. By Invariant (5), we have $r.x \in I(v_r)$. By Invariant (1), we obtain a contradiction in one of the following ways: 1) if v_r is a descendant of a or c, then $I(v_r) \not\subseteq I(a)$ or $I(v_r) \not\subseteq I(c)$ respectively, contradicting Invariant (1), or 2) if v_r is a descendant of b, then $r.y \ge b.y$ and so $r \notin \square_{G_d}$, or else 3) v_r is such that $I(v_r)$ is disjoint from $I(a), I(b), I(c)$, and so $r \notin \square_{G_d}$.

Next we prove the case $x = e$ in a similar way. See Figure 21(c). First, we argue that the five points $b_{\max}, c_{\min}, u_{b_{\max}}, u_{c_{\min}}, u_{a_{\max}}$ are distinct by showing that $I(a).\max < I(b).\max < I(c).\min$. To see this, as $e \notin Q$, we get $I(b).\max < I(c).\min$ (otherwise, by Invariant (4), $e \in Q$). Also $I(a).\max < I(b).\max$, because if $I(b).\min = I(b).\max$, then $I(a).\max < b.x = I(b).\max$ by Claim 7.11. Else, $I(a).\max \le I(b).\min < I(b).\max$. Next, it is easy to see that the five points are in the correct relative position for ADD gadgets. It is left to show that $\square_{G_e} \cap Q = \emptyset$. Suppose there is a point $r \in \square_{G_e} \cap Q$. Then, let v_r be a node such that $v_r.y = r.y$. By Invariant (5), we have $r.x \in I(v_r)$. By Invariant (1), we obtain a contradiction in one of the following ways: 1) if v_r is a descendant of a or c, then $I(v_r) \not\subseteq I(a)$ or $I(v_r) \not\subseteq I(c)$ respectively, or 2) if v_r is a descendant of b, then $r.y \ge b.y$ and so $r \notin \square_{G_e}$, or else 3) v_r is such that $I(v_r)$ is disjoint with $I(a), I(b), I(c)$, and so $r \notin \square_{G_e}$.

Remains the case $x = f$. See Figure 21(d). First, we argue that the five points $a_{\max}, c_{\min}, u_{a_{\max}}, u_{c_{\min}}, u_{\min}$ are distinct by showing that $I(u).\min < I(a).\max < I(c).\min$. To see this, $I(a).\max < I(c).\min$
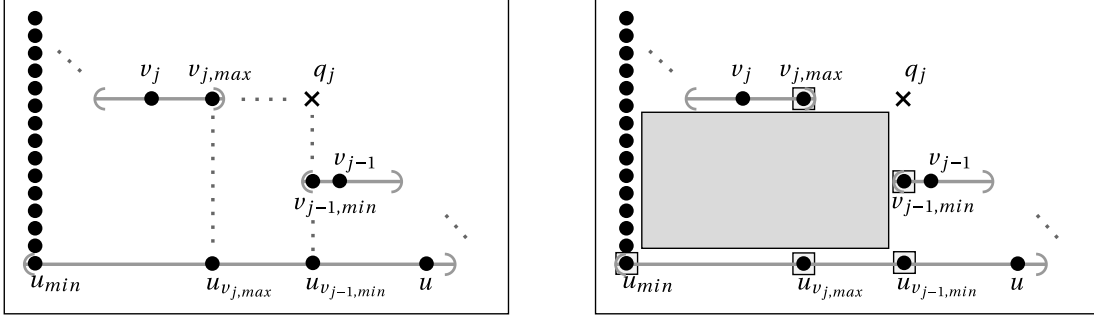
43

Figure 22: (a) Points in the proof of Claim 7.19. **(b)** `ADD` gadget G_{q_j}. Shaded rectangle is empty.

by Claim 7.12. Also $I(u).\min < I(a).\max$. There are two cases: if $I(a).\min < I(a).\max$ and we are done because $I(u).\min \leq I(a).\min$. Else, we have $I(a).\min = I(a).\max$, then, by Invariant (3'), $u_{a_{\max}} \notin Q \setminus \text{box}$. As $u_{a_{\max}} \in Q$, it must be that $u_{a_{\max}} \in \text{box}$, i.e. $u.y = 0$. Hence, $I(u).\min = 0$. But $I(a).\max > 0$. Next, we can see that the five points are in the correct relative position for `ADD` gadgets. It is left to show that $\square_{G_f} \cap Q = \emptyset$. Suppose there is a point $r \in \square_{G_f} \cap Q$. Then, let v_r be a node such that $v_r.y = r.y$. By Invariant (5), we have $r.x \in I(v_r)$. By Invariant (1), we obtain a contradiction in one of the following ways: 1) if v_r is a descendant of c, then $I(v_r) \nsubseteq I(c)$, or 2) if v_r is a descendant of a, then $r.y \geq a.y$ and so $r \notin \square_{G_f}$, or else 3) v_r is such that $I(a).\max < I(v_r).\min \leq I(v_r).\max < I(c).\min$. ∎

One-neighbor phases. Suppose that we are at the beginning of the one-neighbor phase of the r-th round. Let $v_1, \ldots, v_{k+1}$ be the current children of $u = u_{r-1}$ ordered from left to right. Let i be the index such that, at the end of the round, v_j is linked to v_{j+1} for all $j < i$, and v_j is linked to v_{j-1} for all $j > i$. By symmetry, we only consider j where $j < i$ from now. A point $q_j = \langle I(v_{j-1}).\max, v_j.y \rangle$ is included into Q' (if it is not already in Q). Let $G_{q_j} = (v_{j-1,\max}, v_{j,\min}, u_{v_{j-1,\max}}, u_{v_{j,\min}}, u_{\min})$. See Figure 22.

Claim 7.19. If q_j is newly added, i.e. $q_j \notin Q$, then G_{q_j} is an `ADD` gadget in Q and q_j fills G_{q_j}.

Proof. As $q_j \notin Q$, then $I(v_{j-1}).\max < I(v_j).\min$ (otherwise $q_j \in Q$ by Invariant (4)). So the five points $v_{j-1,\max}, v_{j,\min}, u_{v_{j-1,\max}}, u_{v_{j,\min}}, u_{\min}$ are distinct. Also, they are all in Q by Invariant (4), and are in the correct relative position for an `ADD` gadget. It is left to argue that $\square_{G_{q_j}} \cap Q = \emptyset$. Suppose there is a point $r \in \square_{G_{p_j}} \cap Q$. Then, let v' be a node such that $v'.y = r.y$. By Invariant (5), we have $r.x \in I(v')$. By Invariant (1), we obtain a contradiction in one of the following ways: 1) if v' is a descendant of v_j, then $I(v') \nsubseteq I(v_j)$, or 2) if v' is a descendant of v_{j-1}, then $v'.y \geq v_{j-1}.y$ and so $r \notin \square_{G_{q_j}}$, or else 3) v_r is such that $I(v_{j-1}).\max < I(v').\min$ and so $r \notin \square_{G_{q_j}}$. ∎

7.5.4 Finishing the proof

Finally, we conclude with the proof of Lemma 7.1.

Proof of Lemma 7.1. First, observe that $|Q| = \Theta(|\mathsf{Smooth}_{path}(P)|)$ by Lemma 7.10 and Lemma 7.14. We partition $Q = Q_2 \cup Q_1 \cup Q^{init}$ where Q_2 and Q_1 are points newly added to Q in two-neighbor and one-neighbor phases respectively, and $Q^{init} = P \cup \text{box}$ is the initial point set. Similarly, we

write $S = \mathsf{Smooth}_{path}(P)$ and partition $S = S_2 \cup S_1$ where S_2 and S_1 correspond to the links in two-neighbor and one-neighbor phases respectively. By Lemma 7.10, we have $|S_2| \leq |Q_2| \leq 3|S_2|$, and, by Lemma 7.14, we have $|Q_1|, |S_1| \leq 2n$, and so $|Q_1 \cup Q^{init}| \leq 6n$. There are two cases: if $|Q_2| \leq 20n$, then $|Q| = \Theta(n)$ and $|S| = \Theta(n)$. Else, we have $|Q| = \Theta(|Q_2|) = \Theta(|S_2|) = \Theta(|S|)$. Therefore, $|Q| = \Theta(|\mathsf{Smooth}_{path}(P)|)$.

Next, we claim that $\mathsf{Greedy}(P) = Q \setminus \mathtt{box}$ exactly. By Lemma 7.17, Q is constructed by repeatedly filling some $\mathtt{ADD}$ gadget in Q. We claim that this continues until there is no $\mathtt{ADD}$ gadget in Q. This is because, by Lemma 7.9, Lemma 7.10 and Lemma 7.14, the invariants hold. So, by Proposition 7.3, at the end Q is a satisfied superset of $P \cup \mathtt{box}$. Hence, by Lemma 6.6, there is no $\mathtt{ADD}$ gadget left in Q. Recall Definition 6.9. We can see that Q is produced exactly by the non-deterministic Greedy. Therefore $Q \setminus \mathtt{box} = \mathsf{Greedy}_{nondet}(P) = \mathsf{Greedy}(P)$, by Theorem 6.10. So we can conclude $|\mathsf{Greedy}(P)| = \Theta(|\mathsf{Smooth}_{path}(P)|)$. ∎

8 Discussion and open questions

The work presented in this paper raises several new questions. We discuss those that we find the most interesting.

Further study of the stable heap model. The fact that a simple modification in the definition of the link operation implies a general connection between heaps and BSTs is intriguing. We believe that the stable model is interesting in its own right.

- **Stable vs. non-stable heaps:** How restrictive is stable linking? Are there examples showing separations between the various stable and non-stable algorithms?

- **Instance-specific lower bounds for general sequences:** We have transferred instance-specific lower bounds from BSTs to stable heaps when only extract-min operations are used (§5.2). Can one extend this study to general sequences including extract-min, insert, and decrease-key operations?

- **Algorithms (besides smooth heap) in the stable heap model:** What is the complexity of the stable variants of pairing heaps described in Figure 6? How good are their BST counterparts?

Further connection between heaps and BSTs. By Theorem 5.5, we can obtain an offline BST algorithm from any stable heap algorithm. The following directions seem promising.

- **Beyond stable heaps:** Can one do the same for (some class of) non-stable heap algorithms?

- **Online BSTs:** Our connections yield offline BST algorithms, except for GreedyFuture, which can be made online [DHI$^+$09]. Can one derive other online BST algorithms from heaps?

- **Characterization:** Is it true that *every* BST algorithm can be obtained from some stable heap algorithm? If not, what is the subclass of BST algorithms for which this is possible? Does it include algorithms such as Splay, Move-to-root, Tango, a static BST, etc.? If yes, what stable heap algorithms generate these?

Further analysis of smooth heaps. In §5, we gave an upper bound on the amortized cost of extract-min for smooth heaps, and two instance-specific upper bounds. There are several further directions.

- What is the amortized cost of **insert** and **decrease-key**? The techniques developed in [CGK$^+$15a] for Greedy may be useful. As discussed in §1, matching the bounds of Fibonacci heaps by a simpler algorithm is a long-standing problem. Can smooth heaps achieve this?

- Does smooth heap satisfy **working-set-like bounds** similar to those studied in [IL05, Elm06, EFI12]?

- Are smooth heaps (with a careful implementation) efficient **in practice**?

46

Status of the conjectured optimality of GreedyFuture. In §5.1 we present results conditional on the the conjectured optimality of GreedyFuture (i.e. Conjecture 2.9). Settling the following questions would give more insight towards the proof or refutation of the conjecture, a significant open problem of the field.

- **Optimality of smooth heap:** Is smooth heap (in sorting-mode) an instance-optimal stable heap algorithm? By our results, the competitive ratio of the smooth heap is not worse than the competitive ratio of Greedy. Is it better?

- **Inversion and reversion:** Is it true that for every permutation X,

$$|\mathsf{GreedyFuture}\,(X)| = \Theta(|\mathsf{GreedyFuture}\,(X')|) = \Theta(|\mathsf{GreedyFuture}\,(X^r)|)?$$

  How about $\mathsf{Smooth}\,(\cdot)$, $\mathsf{OPT_{stable}}(\cdot)$, and $\mathsf{OPT_{ibst}}(\cdot)$? If any of these are false, then so is Conjecture 2.9.

- **Approximation algorithms:** Are there polynomial-time algorithms for $O(1)$-approximating $\mathsf{OPT_{bst}}$ and $\mathsf{OPT_{stable}}$? Both of these are captured by clean geometric problems (i.e. the satisfied superset and the star-path problem).

Acknowledgement

References

[AM78]   Brian Allen and Ian Munro. Self-organizing binary search trees. *J. ACM*, 25(4):526–535, October 1978. 2

[Art07]   David Arthur. Fast sorting and pattern-avoiding permutations. In *ANALCO 2007*, pages 169–174, 2007. 8

[AVL62]   G. M. Adelson-Velskiĭ and E. M. Landis. An algorithm for organization of information. *Dokl. Akad. Nauk SSSR*, 146:263–266, 1962. 1

[Bay72]   Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, Dec 1972. 1

[BLT12]   Gerth Stølting Brodal, George Lagogiannis, and Robert Endre Tarjan. Strict fibonacci heaps. In *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA, May 19 - 22, 2012*, pages 1177–1184, 2012. 2

[BN13]   Jérémy Barbay and Gonzalo Navarro. On compressing permutations and adaptive sorting. *Theoretical Computer Science*, 513:109 – 123, 2013. 8

[Bro13]   Gerth Stølting Brodal. A survey on priority queues. In *Space-Efficient Data Structures, Streams, and Algorithms - Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, pages 150–163, 2013. 2

[CGK+15a]   P. Chalermsook, M. Goswami, L. Kozma, K. Mehlhorn, and T. Saranurak. Greedy is an almost optimal deque. *WADS*, 2015. 46

[CGK+15b]   Parinya Chalermsook, Mayank Goswami, László Kozma, Kurt Mehlhorn, and Thatchaphol Saranurak. Pattern-avoiding access in binary search trees. In *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 410–423, 2015. 2, 8, 9, 24, 27, 36, 53, 54

[CGK+16]   Parinya Chalermsook, Mayank Goswami, László Kozma, Kurt Mehlhorn, and Thatchaphol Saranurak. The landscape of bounds for binary search trees. *CoRR*, abs/1603.04892, 2016. 24, 54, 55

[Cha13]   Timothy M. Chan. *Quake Heaps: A Simple Alternative to Fibonacci Heaps*, pages 27–32. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. 2

[CMSS00]   Richard Cole, Bud Mishra, Jeanette Schmidt, and Alan Siegel. On the dynamic finger conjecture for splay trees. part i: Splay sorting log n-block sequences. *SIAM J. Comput.*, 30(1):1–43, April 2000. 2, 24, 53

[Col00]   R. Cole. On the dynamic finger conjecture for splay trees. part ii: The proof. *SIAM Journal on Computing*, 30(1):44–85, 2000. 2, 24, 53

[DGST88]   James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert Endre Tarjan. Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation. *Commun. ACM*, 31(11):1343–1354, 1988. 2

[DHI+09]   Erik D. Demaine, Dion Harmon, John Iacono, Daniel M. Kane, and Mihai Pătraşcu. The geometry of binary search trees. In *SODA 2009*, pages 496–505, 2009. 2, 8, 9, 10, 11, 12, 17, 25, 26, 46, 55

[DHIP07]   Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Pătraşcu. Dynamic optimality - almost. *SIAM J. Comput.*, 37(1):240–251, 2007. 2, 9, 26

[DKK+18]   Dani Dorfman, Haim Kaplan, László Kozma, Seth Pettie, and Uri Zwick. Improved bounds for multipass pairing heaps and path-balanced binary search trees. In *26th Annual European Symposium on Algorithms, ESA 2018, August 20-22, 2018, Helsinki, Finland*, pages 24:1–24:13, 2018. 3, 15, 16

[DKKZ18]   Dani Dorfman, Haim Kaplan, László Kozma, and Uri Zwick. Pairing heaps: the forward variant. In *43rd International Symposium on Mathematical Foundations of Computer Science, MFCS 2018, August 27-31, 2018, Liverpool, UK*, pages 13:1–13:14, 2018. 3, 15, 16

[ECW92]   Vladmir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Comput. Surv.*, 24(4):441–476, December 1992. 8

[EFI12]   Amr Elmasry, Arash Farzan, and John Iacono. A priority queue with the time-finger property. *J. Discrete Algorithms*, 16:206–212, 2012. 4, 46

[Elm04]   Amr Elmasry. On the sequential access theorem and deque conjecture for splay trees. *Theoretical Computer Science*, 314(3):459 – 466, 2004. 2

[Elm06]   Amr Elmasry. A priority queue with the working-set property. *Int. J. Found. Comput. Sci.*, 17(6):1455–1466, 2006. 4, 46

[Elm10]   Amr Elmasry. The violation heap: a relaxed fibonacci-like heap. *Discrete Math., Alg. and Appl.*, 2(4):493–504, 2010. 2

[Epp]   David Eppstein. Splaysort versus cartesian tree sort. `https://11011110.github.io/blog/2014/01/21/splaysort-versus-cartesian.html`. Accessed: 2017-08-25. 9, 52, 53

[Fox11]   Kyle Fox. Upper bounds for maximally greedy binary search trees. In *WADS 2011*, pages 411–422, 2011. 24

[Fre99a]   Michael L. Fredman. On the efficiency of pairing heaps and related data structures. *J. ACM*, 46(4):473–501, 1999. 3, 5, 16, 26

[Fre99b]   Michael L. Fredman. A priority queue transform. In *Algorithm Engineering, 3rd International Workshop, WAE '99, London, UK, July 19-21, 1999, Proceedings*, pages 244–258, 1999. 13

[FSST86]   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. 3, 4, 8, 13, 15, 16

[FT87]   Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987. 2

[GBT84]   Harold N. Gabow, Jon Louis Bentley, and Robert Endre Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1984, Washington, DC, USA*, pages 135–143, 1984. 21

[Geo08]   George F. Georgakopoulos. Chain-splay trees, or, how to achieve and prove loglogn-competitiveness by splaying. *Inf. Process. Lett.*, 106(1):37–43, 2008. 26

[GKKS07]   Joachim Gudmundsson, Oliver Klein, Christian Knauer, and Michiel Smid. Small manhattan networks and algorithms for the earth mover's distance. In *In Proc. 23rd European Workshop Comput. Geom. (EWCG'07*, pages 174–177, 2007. 26

[GM14]   Sylvain Guillemot and Dániel Marx. Finding small patterns in permutations in linear time. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 82–101, 2014. 8, 24

[GS78]   Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978*, pages 8–21, 1978. 1

[Har06]   Dion Harmon. *New Bounds on Optimal Binary Search Trees*. PhD thesis, Massachusetts Institute of Technology, 2006. 26

[HKTZ15] Thomas Dueholm Hansen, Haim Kaplan, Robert Endre Tarjan, and Uri Zwick. Hollow heaps. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part I*, pages 689–700, 2015. 2

[HST11] Bernhard Haeupler, Siddhartha Sen, and Robert Endre Tarjan. Rank-pairing heaps. *SIAM J. Comput.*, 40(6):1463–1485, 2011. 2

[Iac00] 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. 3

[Iac13] 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. 26

[IL05] John Iacono and Stefan Langerman. Queaps. *Algorithmica*, 42(1):49–56, 2005. 4, 46

[IL16] John Iacono and Stefan Langerman. *Weighted dynamic finger in binary search trees*, chapter 49, pages 672–691. 2016. 2, 24, 53, 54

[IÖ14a] John Iacono and Özgür Özkan. A tight lower bound for decrease-key in the pure heap model. *CoRR*, abs/1407.6665, 2014. 3, 4, 5, 16, 26

[IÖ14b] John Iacono and Özgür Özkan. Why some heaps support constant-amortized-time decrease-key operations, and others do not. In *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I*, pages 637–649, 2014. 3, 4, 5, 16, 26

[Kit11] S. Kitaev. *Patterns in Permutations and Words*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2011. 8, 24

[Knu68] Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 1968. 8, 24

[Knu97] 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. 4, 13

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. 1998. 2, 5, 8

[KT08] Haim Kaplan and Robert Endre Tarjan. Thin heaps, thick heaps. *ACM Trans. Algorithms*, 4(1):3:1–3:14, 2008. 2

[LP89] Christos Levcopoulos and Ola Petersson. *Heapsort—Adapted for presorted files*, pages 499–509. Springer Berlin Heidelberg, Berlin, Heidelberg, 1989. 8, 9, 52, 54

[LP94] Christos Levcopoulos and Ola Petersson. Sorting shuffled monotone sequences. *Inf. Comput.*, 112(1):37–50, 1994. 8, 52

[LST14] 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. 2, 3

[Luc88] Joan M. Lucas. Canonical forms for competitive binary search tree algorithms. *Tech. Rep. DCS-TR-250, Rutgers University*, 1988. 2, 9, 10, 12, 25

[Man84] Heikki Mannila. Measures of presortedness and optimal sorting algorithms (extended abstract). In *ICALP 1984*, pages 324–336, 1984. 8, 52

[Meh79] Kurt Mehlhorn. Sorting presorted files. In *Theoretical Computer Science, 4th GI-Conference, Aachen, Germany, March 26-28, 1979, Proceedings*, pages 199–212, 1979.

8, 52

[MP92] Alistair Moffat and Ola Petersson. An overview of adaptive sorting. *Australian Computer Journal*, 24(2):70–77, 1992. 8

[MS76] J. Ian Munro and Philip M. Spira. Sorting and searching in multisets. *SIAM J. Comput.*, 5(1):1–8, 1976. 8

[Mun00] J.Ian Munro. On the competitiveness of linear search. In Mike S. Paterson, editor, *Algorithms - ESA 2000*, volume 1879 of *Lecture Notes in Computer Science*, pages 338–345. Springer Berlin Heidelberg, 2000. 2, 12, 25

[NRRS17] Ilan Newman, Yuri Rabinovich, Deepak Rajendraprasad, and Christian Sohler. Testing for forbidden order patterns in an array. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 1582–1597, 2017. 8, 24

[Pet05] 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. 1, 3, 8

[Pet08] Seth Pettie. Splay trees, Davenport-Schinzel sequences, and the deque conjecture. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '08, pages 1115–1124, Philadelphia, PA, USA, 2008. Society for Industrial and Applied Mathematics. 1, 2, 8

[Pra73] Vaughan R. Pratt. Computing permutations with double-ended queues, parallel stacks and parallel queues. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, STOC '73, pages 268–277, New York, NY, USA, 1973. ACM. 8, 24

[SA96] Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996. 2, 9

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985. 2, 4, 12, 24

[Sun92] Rajamani Sundar. On the deque conjecture for the splay algorithm. *Combinatorica*, 12(1):95–124, 1992. 2

[SV87] John T. Stasko and Jeffrey Scott Vitter. Pairing heaps: Experiments and analysis. *Commun. ACM*, 30(3):234–249, March 1987. 2

[Tak03] Tadao Takaoka. Theory of 2-3 heaps. *Discrete Applied Mathematics*, 126(1):115–128, 2003. 2

[Tar72] Robert Endre Tarjan. Sorting using networks of queues and stacks. *J. ACM*, 19(2):341–346, April 1972. 8, 24

[Tar85] Robert Endre Tarjan. Sequential access in splay trees takes linear time. *Combinatorica*, 5(4):367–378, 1985. 2

[Vui80] Jean Vuillemin. A unifying look at data structures. *Commun. ACM*, 23(4):229–239, April 1980. 9

[WDS06] Chengwen C. Wang, Jonathan C. Derryberry, and Daniel D. Sleator. $O(\log \log n)$-competitive dynamic binary search trees. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*, SODA '06, pages 374–383, Philadelphia, PA, USA, 2006. Society for Industrial and Applied Mathematics. 26

[Wil89] R. Wilber. Lower bounds for accessing binary search trees with rotations. *SIAM Journal on Computing*, 18(1):56–67, 1989. 8, 9, 10, 17, 26

A   Adaptive sorting

Several measures of pre-sortedness have been considered in the literature on adaptive sorting. Most, but not all, of these are known to be subsumed by properties of the BST optimum. A full survey of such measures is out of our scope. We briefly discuss four examples. In all cases, we consider an input permutation $X = (x_1, \ldots, x_n) \in [n]^n$, and we express the number of comparisons a sorting algorithm performs as a function of X.

Number of inversions. Perhaps the most natural measure of pre-sortedness of X is the number of inversions $Inv(X)$. It is equal to the number of pairs i, j, such that $1 \leq i < j \leq n$, and $x_i > x_j$. Observe that for increasing X we have $Inv(X) = 0$, and in general, $Inv(X) \leq \binom{n}{2}$, attained by the decreasing sequence.

There are sorting algorithms that achieve a running time of $O\big(n \cdot \log\big(\frac{Inv(X)}{n} + 1\big)\big)$, and this is, in general, well-known to be optimal (see e.g. [Meh79]).

Number of runs. The number of runs, denoted $Run(X)$ is the number of pairs x_i, x_{i+1}, such that $x_i > x_{i+1}$. This value is between 0 (for the increasing sequence) and $n - 1$ for the decreasing sequence. A running time of $O\big(n \cdot \log\big(Run(X) + 1\big)\big)$ is achievable and, in general, optimal [Man84].

Number of oscillations. The number of oscillations of X is defined in [LP89] as follows:

$$Osc(X) = \sum_{i=1}^{n} \big|\{j : \min\{x_i, x_{i+1}\} < x_j < \max\{x_i, x_{i+1}\}\}\big|.$$

In words, $Osc(X)$ measures the number of oscillations, i.e. the number of times that a consecutive pair "brackets" some item in the sequence. Clearly, $0 \leq Osc(X) \leq n^2/2$.

Levcopoulos and Petersson [LP89] show that their sorting algorithm based on the Cartesian tree achieves $O\big(n \cdot \log\big(\frac{Osc(X)}{n} + 1\big)\big)$, and that this is, in general, optimal. Furthermore, they show that $Osc(X) \leq 4 \cdot Inv(X)$, and $Osc(X) \leq 2n \cdot Run(X) + n$. In consequence, an algorithm that is optimal with respect to the number of oscillations is also optimal (up to a constant factor) with respect to the number of inversions and the number of runs.

Shuffled monotone sequences. The quantity $SMS(X)$, introduced in [LP94], is the minimum number of monotone subsequences (not necessarily contiguous) that cover the sequence X. Levcopoulos and Petersson [LP94] show that their *Slabsort* algorithm achieves the running time $O\big(n \cdot \log\big(SMS(X) + 1\big)\big)$, which is, in general, optimal. As $SMS(X) \leq Run(X) + 1$ clearly holds, the "shuffled monotone sequences" measure subsumes the "number of runs" measure.

In the following we discuss the relationship between the four measures and the BST optimum.

We first show that the BST optimum subsumes the "number of oscillations" measure. The argument is based on an unpublished note by Eppstein [Epp]. Then we show that the BST optimum *almost* subsumes the "shuffled monotone sequences" measure.

Let $DF(X)$ denote the *dynamic finger* bound, i.e.

$$DF(X) = \sum_{i=2}^{n} \log\left(|x_{i+1} - x_i| + 1\right).$$

It is known that $\mathsf{OPT_{bst}}(X) \leq O(DF(X))$, furthermore, the Splay and Greedy algorithms also match this bound [CMSS00, Col00, IL16].

Lemma A.1 (Eppstein [Epp]).

$$DF(X) \leq O\left(n \cdot \log\left(\frac{Osc(X)}{n} + 1\right)\right).$$

Proof. We denote $M_i = \max\{x_i, x_{i+1}\}$ and $m_i = \min\{x_i, x_{i+1}\}$. We thus have $DF(X) = \sum_{i=1}^{n-1} \log\left(M_i - m_i + 1\right)$. We say that i *brackets* j if $m_i < x_j < M_i$.

Consider the quantity:

$$Q = \sum_{i=1}^{n-1} \sum_{j:\ i \text{ brackets } j} \frac{1}{M_i - x_j}.$$

Let H_n denote the n-th harmonic number, i.e. $H_n = \sum_{i=1}^{n} \frac{1}{n}$. Since all items bracketed by i have distinct ranks between m_i and M_i, we have:

$$Q = \sum_{i=1}^{n-1} \sum_{j=m_i+1}^{M_i-1} \frac{1}{M_i - j} = \sum_{i=1}^{n-1} H_{(M_i - m_i - 1)} = \Theta(DF(X)).$$

On the other hand we have:

$$Q = \sum_{j=1}^{n} \sum_{i:\ i \text{ brackets } j} \frac{1}{M_i - x_j}.$$

Let $Osc_j(X)$ denote the number of items that bracket j. We have $Osc(X) = \sum_{j=1}^{n} Osc_j(X)$.

Since for all i that bracket j, at most two have the same value M_i, we have

$$Q \leq \sum_{j=1}^{n} 2H_{Osc_j(X)}.$$

By convexity, this quantity is maximized if all $Osc_j(X)$ are equal, and thus

$$Q \leq O\left(n \cdot \log\left(\frac{Osc(X)}{n} + 1\right)\right). \quad \blacksquare$$

We now discuss the connection between the BST optimum and the "shuffled monotone sequences" measure.

Let k denote the quantity $SMS(X)$. In [CGK$^+$15b] we study the restricted variant of $SMS(X)$ in which all monotone subsequences are increasing. We show (from more general results for pattern-avoiding sequences) that $\mathsf{OPT_{bst}}(X) \leq n \cdot 2^{O(k^2)}$. Furthermore, this bound is achieved by Greedy.

In [CGK+16] we show, building on [IL16], the stronger $\mathsf{OPT}_{\mathsf{bst}}(X) \leq O(n \cdot k \log k)$, which is at this time not known to be matched by online BST algorithms. (In [CGK+16] the bound is stated for shuffled *increasing* sequences, but the argument easily extends to shuffled *monotone* sequences.)

Thus, we observe that the BST optimum is known to match the $O(n \log k)$ bound from the adaptive sorting literature only for small values of k. Whether, in fact, it holds that $\mathsf{OPT}_{\mathsf{bst}}(X) \leq O(n \cdot \log k)$, i.e. whether the BST optimum subsumes the SMS measure is an interesting open question. We note that, in general, we cannot expect the BST optimum to subsume *every* structural measure for adaptive sorting; for every input sequence there is *some* sorting algorithm that can sort that particular sequence with a linear number of comparisons, whereas $\mathsf{OPT}_{\mathsf{bst}} = \Omega(n \log n)$ for most sequences.

B   Cartesian tree sorting

Levcopoulos and Petersson [LP89] show that the quantity $n \cdot \log\left(\frac{Osc(X)}{n} + 1\right)$ is not just an upper bound, but (up to a small constant factor) a closed-form expression for the exact running time of the Cartesian sort algorithm.

We show an example in which sorting with the smooth heap is significantly faster than sorting with the Cartesian tree algorithm. The example is the "tilted grid" permutation, shown in Figure 23. A simple inductive argument shows that the cost of Greedy on both this permutation and its inverse is linear [CGK+15b, CGK+16]. As a consequence, by our results in this paper, the cost of sorting the permutation by smooth heap is $O(n)$. On the other hand, we observe that at least half of the items are "bracketed" by $\Theta(\sqrt{n})$ other items, yielding $Osc(X) = \Omega(n\sqrt{n})$. The running time of Cartesian sort on this sequence is thus $\Omega(n \log n)$.
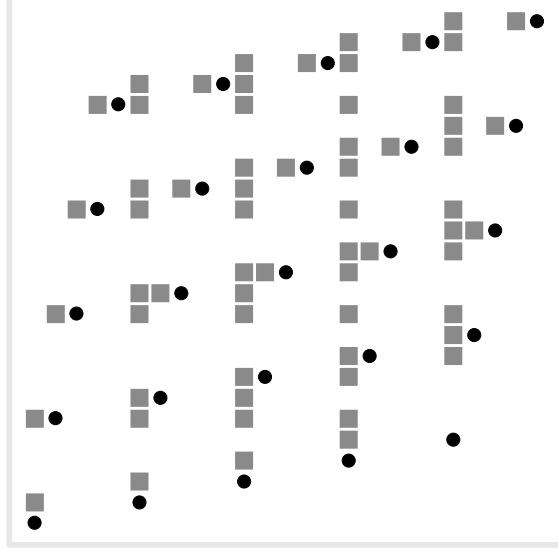
Figure 23: Tilted grid sequence. Points denote input permutation point set, squares denote Greedy output. The sequence is defined as follows. Let $n = t^2$ and consider the point set $\mathcal{P} = \{(it + (j-1), jt + i - 1) \mid i, j \in [t]\}$. It is not hard to verify that no two points align on either x or y coordinates, therefore $\mathcal{P}$ defines a permutation of $[n]$. The example is a special case of a more general family of "perturbed grid" sequences, amenable to similar analysis [CGK+16].

As the running time of Cartesian sort is subsumed by the dynamic finger bound, an example on which Cartesian tree sorting is asymptotically faster than smooth heap sorting would contradict the dynamic optimality of Greedy (Conjecture 2.9). In fact, it would contradict even the weaker conjecture that Greedy (X) and Greedy (X') are within a constant factor of each other.

C Geometry of insert-only execution

Theorem C.1 (Geometry of BSTs, insertion-compatible (Restatement of Theorem 2.3)). *Let $X \in [n]^n$ be an arbitrary permutation, and let $Q \subset [n]^2$. Q is a insertion-compatible satisfied superset of P^X iff there is an offline BST algorithm $\mathcal{A}_{\mathtt{bst}}$ in insert-only mode such that $\mathcal{A}_{\mathtt{bst}}(X) = Q$.*

Proof. ($\Rightarrow$) This direction is similar to [DHI+09, Lemma 2.2]. In their proof, each node is assigned with a priority which is the next touch time, and then the min-treap is maintained. By the property of this treap, at time i, the keys from $K_i = \{p.x \mid p \in Q_{y=i} \setminus \{p_{y=i}\}\}$ form a connected subtree containing the current root of the treap. So at time i, we just touch exactly K_i which contains both the current predecessor and successor of $p_{y=i}.x$. Then, we insert $p_{y=i}.x$ into the subtree formed by K_i and rearrange the subtree with next touch time as priority again. This implies that each time i, we touch exactly K_i. Hence, the execution trace is exactly Q.

($\Leftarrow$) This is exactly [DHI+09, Lemma 2.1] but with an additional observation that, for any key x, x is never touched before it is inserted. So Q is insertion-compatible for P^X. ∎

D Detailed pseudocode of smooth heap

Input: A list of nodes, with x initially pointing to the leftmost node

(**LTR**) **while** x.next $\neq$ null **do**

> **if** $(x$.key $< x$.next.key$)$ **then** $x \leftarrow x$.next
>
> **else**   ▷ Local maximum found
>
> > **while** $(x$.prev $\neq$ null$)$ **do**
> >
> > > **if** $(x$.prev.key $> x$.next.key$)$ **then**
> > >
> > > > $x \leftarrow x$.prev
> > > >
> > > > link(x)
> > >
> > > **else**
> > >
> > > > $x \leftarrow x$.next
> > > >
> > > > link$(x$.prev$)$
> > > >
> > > > **continue** (**LTR**)
> >
> > link(x)

(**RTL**) **while** x.prev $\neq$ null **do**

> $x \leftarrow x$.prev
>
> link(x)

Figure 24: Pseudocode of smooth heap re-structuring

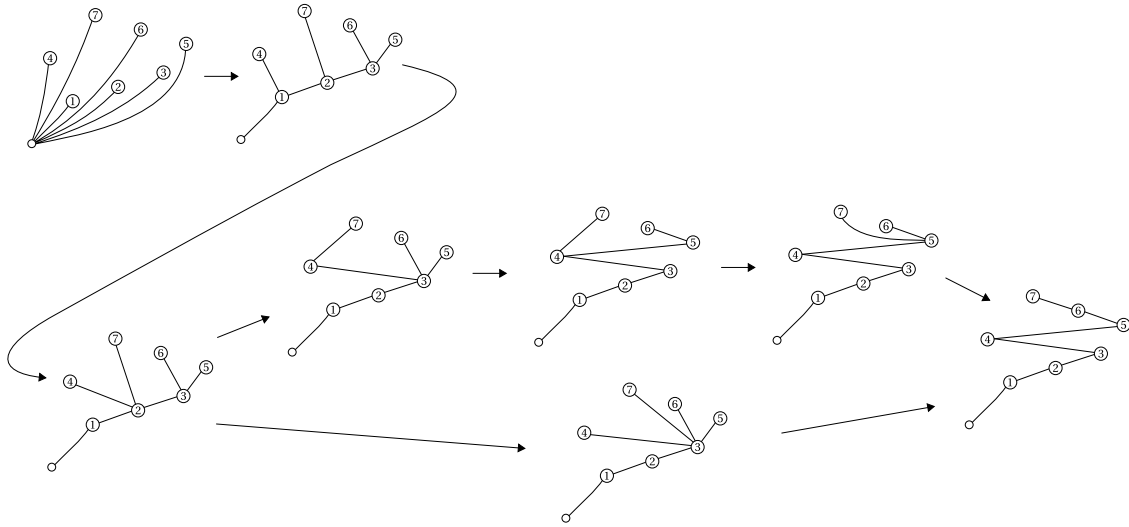56

E   Example in which smooth heap is not optimal

Figure 25: Sequence of operations illustrating that smooth heap is not globally optimal. Top branch: smooth heap execution in sorting-mode for $X = (4, 1, 7, 2, 6, 3, 5)$. Total link-cost is 13. Bottom branch: an alternative stable heap execution for the same input. Total link-cost is 12.