SAARLAND UNIVERSITY

Binary Search Trees, Rectangles and Patterns

László KOZMA

Dissertation for obtaining the title of

Doctor rerum naturalium (Dr. rer. nat.)

of the Faculty of Natural Sciences and Technology
of Saarland University

Saarbrücken, Germany
July 2016

Colloquium Information

Date and place:	Friday 16th September, 2016 Saarbrücken, Germany
Dean:	Prof. Dr. Frank-Olaf Schreyer *Saarland University*
Chair:	Prof. Dr. Dr. h.c. Wolfgang Paul *Saarland University*
Reviewers:	Prof. Dr. Raimund Seidel *Saarland University* Prof. Dr. Dr. h.c. mult. Kurt Mehlhorn *Max Planck Institute for Informatics* Prof. Robert E. Tarjan, Ph.D. *Princeton University*
Scientific assistant:	Dr. Tobias Mömke *Saarland University*

Abstract

The topic of this thesis is the classical problem of searching for a sequence of keys in a binary search tree (BST), allowing the re-arrangement of the tree after every search. Our current understanding of the power and limitations of this model is incomplete, despite decades of research. The proven guarantees for the best known algorithms are far from the conjectured ones. We cannot efficiently compute an optimal sequence of rotations for serving a sequence of queries (even approximately and even with advance knowledge of the input), but we also cannot show this problem to be difficult. Sleator and Tarjan conjectured in 1983 that a simple *online* strategy for tree re-arrangement is as good, up to a constant factor, as the theoretical optimum, for every input. This is the famous *dynamic optimality conjecture*. In this thesis we make the following contributions to the topic.

- We define in various ways the computational models in which BST algorithms are described and analyzed. We clarify some of the assumptions that are made in the literature (often implicitly), and survey known results about the BST model. (§ 2)
- We generalize Splay, a popular BST algorithm that has several proven efficiency-properties, and define a set of sufficient (and, in a limited sense, necessary) criteria that guarantee the efficient behavior of a BST algorithm. The results give new insights into the behavior and efficiency of Splay (a topic that is generally considered intriguing). (§ 3)
- We study query sequences in terms of their avoided patterns, a natural and general structural property from combinatorics. We show that pattern-avoiding sequences can be served much faster than what the logarithmic worst-case guarantees would suggest. The result complements classical structural bounds such as dynamic finger and working set. The study of pattern-avoiding inputs generalizes known examples of easy sequences, introduces new barriers towards dynamic optimality, and addresses open questions in the BST model. (§ 4)
- We introduce a novel interpretation of searching in BSTs in terms of rectangulations, a well-studied combinatorial structure also known as mosaic floorplan. The connection to rectangulations gives a new perspective on the BST model. Furthermore, it answers open questions from the literature about rectangulations and gives simplified proofs for known results. The relation of BSTs and rectangulations to other structures such as Manhattan networks is also explored. We see the main value of the presented connections in the fact that they bring new techniques to the study of dynamic optimality. (§ 5)

Throughout the thesis we state a number of open problems (some well-known, some new). The purpose of this is to collect in one place information that is scattered throughout the literature. Furthermore, we attempt to identify intermediate questions (easier than the dynamic optimality conjecture). The list of problems may help an interested reader in starting research on this family of problems.

Zusammenfassung

Das Thema dieser Dissertation ist das klassisches Problem, Schlüssel in einem binären Suchbaum (BS) zu suchen. Zurzeit ist unser Verständnis dieses Problems, trotz jahrzehntelanger Forschung, begrenzt. Die beweisbaren Garantien für die bekanntesten Algorithmen sind weit von den vermuteten Garantien. Wir können keine optimale Sequenz von Rotationen effizient berechnen, um eine Folge von Abfragen zu bedienen. (Auch dann nicht wenn wir das Optimum nur approximieren wollen und wenn wir die Angabe im Voraus kennen.) Wir können auch nicht beweisen dass dieses Problem schwer ist. Sleator und Tarjan haben 1983 vermutet, dass eine einfache online-Strategie für die Reorganisierung eines BS, für alle Eingaben, bis auf einen konstanten Faktor, optimal ist. Dies ist die berühmte *Dynamic Optimality Vermutung*. In dieser Dissertation leisten wir zu diesem Thema die Folgende Beiträge.

- Wir definieren auf verschiedene Weise die Rechenmodelle in denen BS-Algorithmen beschrieben und analysiert werden. Wir versuchen, manche Annahmen aus der Literatur explizit zu machen, und wir geben einen Überblick über bekannte Ergebnisse über das BS-Modell. (§ 2)

- Wir verallgemeinern Splay, einen berühmten BS Algorithmus, der viele beweisbare Effizienz-eigenschaften hat, und wir definieren eine Menge von hinreichenden (in einem gewissen Sinn auch notwendigen) Kriterien, die die Effizienz eines BS-Algorithmus garantieren. Die Ergebnisse gewähren ein neuen Einblick in das Verhalten des Splay Algorithmus, ein Thema das oft als verblüffend angesehen wird. (§ 3)

- Wir studieren Abfragefolgen bezüglich ihrer Muster-Vermeidung Eigenschaften, eine generelle und natürliche Familie von Eigenschaften aus der Kombinatorik. Wir zeigen, dass Abfragefolgen die Muster-vermeidend sind, viel schneller bedient werden können als von den logarithmischen Worstcase-Garantien zu erwarten wäre. Diese Ergebnisse verhalten sich komplementär zu den klassischen strukturalen Grenzen wie “dynamic finger” oder “working set”. Unser Untersuchung der Muster-vermeidenden Eingaben generalisiert bekannte Beispielen von einfachen Abfragefolgen, führt neue Barrieren zur Dynamic Optimality Vermutung ein und wirft neue Fragen im BS Modell auf. (§ 4)

- Wir führen neue Interpretationen des BS-Modells ein, die in Beziehung zu Rectangulierungen stehen. Rectangulierungen sind gut untersuchte, auch als Mosaik Tesselierung bekannte Kombinatorische Strukturen. Der Zusammenhang zwischen binären Suchbäumen und Rectangulierungen gibt einen neuen Einblick ins BS-Modell. Weiterhin beantwortet er offene Fragen aus der Literatur über Rectangulierungen, und gibt vereinfachte Beweise für bekannte Ergebnisse. Die Beziehungen des BS-Modells und der Rectangulierungen mit anderen Strukturen wie Manhattan-Netzwerke ist auch erforscht. Der wichtigsten Beitrag der dargelegten Beziehungen liegt darin, dass sie neue Methoden zum Studium der Dynamic Optimality Vermutung bringen. (§ 5)

In der gesamten Dissertation stellen wir zahlreiche offenen Fragen (manche neue, manche wohlbekannt). Unser Ziel ist es damit, Informationen an einem Ort zu sammeln, die zurzeit in der Literatur verstreut sind. Außerdem ermitteln wir dazwischenliegende Fragen, die einfacher als Dynamic Optimality sind. Die Liste der Fragen kann ein Hilfsmittel für die interresierte Leserin sein, die Forschung über diese Familie von Problemen beginnen möchtet.

Note on collaboration

In § 1 and § 2 mostly known results from the literature are described. The chapters also contain definitions, examples, and observations that are new to the thesis.

Most of the results in § 3 and § 4 were obtained in collaboration with Parinya Chalermsook, Mayank Goswami, Kurt Mehlhorn, and Thatchaphol Saranurak. These were published in 2015 in conference proceedings [25] and [24]. The presentation was adapted for the thesis and some strengthened results and new observations were added.

The results in § 5 were obtained in joint work with Thatchaphol Saranurak and were published in manuscript form in 2016 [62].

Acknowledgements

I am grateful to my advisor, Raimund Seidel, for the trust and freedom he afforded me throughout my doctoral studies. I especially thank him for patiently answering every random question I asked him over the years (whether or not related to the thesis), and for not insisting that I should be more focused. He is a role model in both research and teaching, through his intuition for important problems and elegant solutions, as well as his kind and friendly attitude.

I am as well deeply thankful to Kurt Mehlhorn for starting a project on binary search trees in which I was happy to participate. Besides his many insights and ideas on the topic, the deepest influence on me were his efficient and systematic approach to research, his precision and his limitless enthusiasm.

Likewise, I express my gratitude to Robert Tarjan for being on the thesis committee, for reviewing the thesis, and for thoughtful feedback.

The work presented here benefits from many ideas of Parinya Chalermsook, from whom I also learnt a lot about research in general and efficient presentation of results in particular.

Perhaps the best part of my PhD was working with Thatchaphol Saranurak. Due to our different and complementary ways of thinking, our collaboration was always intensive, productive, and enjoyable. The contributions of Thatchaphol can be found throughout the thesis. Besides his concrete ideas, he also contributed indirectly, via his BST simulator that was indispensable for quickly testing (and usually discarding) conjectures.

I found Saarbrücken to be a vibrant center of computer science research, and I was fortunate to discuss various interesting topics, whether or not research-related, with a diverse group of people. Besides those already mentioned, I would especially like to name Victor Alvarez, Karl Bringmann, Radu Curticapean, Lavinia Dinu, Christian Engels, Gorav Jindal, Shay Moran, Tobias Mömke, Sascha Parduhn, Alexey Pospelov, Raghavendra Rao, Karteek Srinivasajan. I also thank Meena Mahajan, Nikhil Bansal, and Saurabh Ray, for hosting me for stimulating research visits.

I gratefully acknowledge the generous financial support from the Department of Computer Science at Saarland University, and at an earlier time from the Graduate School of Computer Science, and for a connecting three months from the Max-Planck Research School. I thank Michelle Carnell for wide-ranging advice – especially at the beginning of my stay in Saarbrücken. I also thank Sandra Neumann for her help with all kinds of practical and administrative matters.

My thanks are due to Rodica Potolea for first showing me the beauty of algorithms and data structures during my undergraduate studies, and to Eva Cotar for teaching me programming and thereby steering me towards this field more than two decades ago.

I thank my mother, Zsuzsa Szabó for raising me and my sister Ágnes, and for showing us the value of learning. I thank my wife Judit for her love, encouragement, and for putting up with my absence – as well as with my presence – all this time. I am grateful to our children Sámuel and Mirjam for every moment we spend together, even if they haven't helped with the thesis. (To be fair, Samu did offer to help.)

In memory of my grandparents

Kozma Mihály and Dr. Szabó Lenke

Contents

Chapter 1

Introduction

This thesis is about the *binary search tree* (in short: BST), a fundamental and ubiquitous data structure for storing and retrieving data with the help of computers. The statement that the BST is one of the simplest concepts in computer science is essentially true, but misleading in at least two ways.

First, the idea of a BST is implicitly contained in the problem of searching in sorted lists, a problem that predates computers by centuries if not millenia (Knuth traces the origins of searching in sorted lists to the ancient Babilonians [60, § 6.2.1]). The idea of *balanced binary search* is itself very old, appearing in disguise in the bisection method for root-finding in mathematical analysis (see e.g. the 19th century work of Bolzano [36, p. 308]). The combinatorial study of structures equivalent to BSTs goes back at least to Euler [79].

Second, behind the apparent simplicity of the humble BST lies a deep well of mathematical complexity. There is a rich literature on various combinatorial, statistical, and algorithmic aspects of BSTs. Nevertheless, several basic questions are still open, in spite of the intensive effort of the research community. Perhaps the most intriguing of these questions is the *dynamic optimality conjecture* of Sleator and Tarjan from 1983. This question and some of its (easier) relatives are the main motivation for the present work.

1.1 Binary search

> While the first binary search was published in 1946, the first binary search that works correctly for all values of n did not appear until 1962.
>
> — JON BENTLEY, *Programming Pearls* (1986)

> I was shocked to learn that the binary search program that Bentley proved correct and subsequently tested in *Programming Pearls* contains a bug.
>
> — JOSHUA BLOCH (2006)

> The [...] proposed fix for C and C++ (Line 6) is not guaranteed to work by the relevant C99 standard.
>
> — ANTOINE TRUX (2008)

Suppose we need to store a set S of n elements from some ordered universe $\mathcal{U}$ and we need to answer queries of the form: Is a certain element $x \in \mathcal{U}$ contained in S? In practice, in case the answer is YES, we would also want to retrieve some additional data associated with x that has been previously stored. However, in this thesis we focus only on answering *membership* queries of the above kind, which we also call *searching* for x.

We refer to elements of $\mathcal{U}$ as *keys*. The fact that they can be ordered is the only assumption we make on keys – therefore, the only operation we may perform with them are *pairwise comparisons* that yield one of the three outcomes ($<$, $>$, or $=$). The search for x consists of a

sequence of comparisons between x and elements in S. (All comparisons among elements of S can be performed in advance, before the query.) Clearly, as soon as we find out that $t < x$ for some key t in S, it becomes pointless to compare x with elements smaller than t, as the outcomes of such comparisons yield no new information. A symmetric argument holds if $t > x$. Therefore, in this model, any *reasonable* strategy of searching for x in S can be described as follows.

Search(S, x)
If S is empty, answer NO. Otherwise select some element $t \in S$ and compare x with t. If $x = t$, answer YES. If $x < t$, remove from consideration all keys greater or equal with t and repeat. If $x > t$, remove from consideration all keys smaller or equal with t and repeat.

This process is called *binary search*. The only remaining detail is the order in which elements of S are selected for comparison with the searched key x.

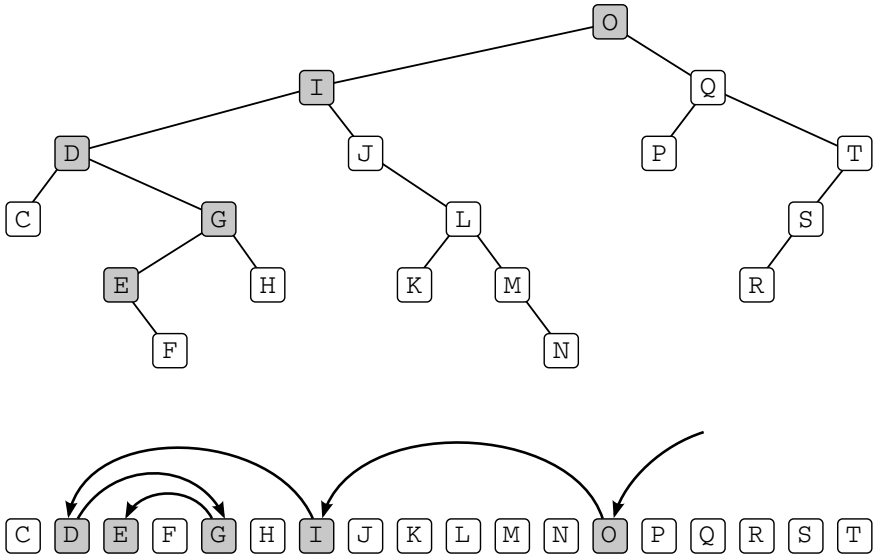

Figure 1.1: Example of a binary search. We search for the letter E in an alphabetically sorted list of letters. Above is the binary search tree corresponding to the search strategy.

The order in which elements are selected for comparison (depending on the outcomes of previous comparisons) can be visualized as a binary search tree (Figure 1.1). More than a visualization, the BST can be thought of as the internal representation of the search strategy, i.e. how it is stored inside a computer.

In informal language, *binary search* often means *balanced binary search*, referring to the strategy in which we compare the search key with (roughly) the median of the remaining keys, thereby (roughly) halving the number of candidate keys after each comparison.

When searching for a key, we would like to perform as few comparisons as possible. Typically, we need to perform more than one search, i.e. we want to serve a sequence of search queries for the same set S. For now we assume that for each search we make a "fresh start", i.e. we search from the root of the BST. Without making any assumption on the search queries (or alternatively, assuming that they are adversarially selected), our best choice is a balanced BST, corresponding to the balanced binary search strategy mentioned before. The number of comparisons in this case is at most logarithmic in terms of the size of S for each query, and simple arguments show that this cannot be improved. On the other hand, given

some information about the distribution of queries, we may be able to perform searches with significantly fewer comparisons. As an extreme example, if a certain key is queried all the time, we should compare with that key first (placing that key in the root of the BST), always obtaining the YES answer with one comparison.

So far, we have assumed that the strategy for searching keys is *fixed* – in other words, that the BST underlying the search strategy is a *static* tree. This is not necessarily the case. In fact, for some query sequences we can reduce the total number of comparisons by changing the search strategy (i.e. the underlying BST) after each search. This is true even if we account for the cost of changing the tree. The model in which the BST is adapted in response to queries is called the *dynamic BST model*, to contrast it with the *static BST model* in which the BST is kept unchanged. The "rules" of these models are defined more precisely in § 2.

Finding the best static tree for a given sequence of queries is a well-studied and well-understood problem. By contrast, our current understanding of the dynamic BST model is incomplete, despite decades of effort.

A note on terminology. The choice of the term "dynamic" in the above sense is due to historical reasons, and we adopt it for compatibility with the literature, although "adaptive" is perhaps a better term. Unfortunately, "dynamic" has another well-established meaning, one that is also relevant in the study of BSTs. Informally, a data structure is dynamic if it supports insertions, deletions, and possibly other operations. This type of "dynamism" raises a range of interesting questions that have been the main driving force behind the early development of the BST field. For instance, the question of whether logarithmic query time can be maintained while allowing efficient insertions and deletions has lead to the invention (or discovery) of AVL trees, red-black trees and other interesting data structures [30]. In this thesis we use the term "dynamic" only in the first sense and we ignore insertions, deletions, and other operations, focusing only on search queries. This means that the set S is unchanged throughout the process. As a matter of fact, we only consider *successful* search queries, which we also call an *access*. While this may seem overly reductive, there are reasons to believe that the restricted problem already captures most of the complexity of the BST model. In any case, the restricted problem is sufficiently difficult to be worthy of study on its own.

1.2 Dynamic optimality

What is the best way to re-arrange a BST in response to search queries? Intuitively, if an element is queried again and again, it should be brought closer to the root. The Move-to-root heuristic of Allen and Munro [7] from 1978 achieves this in a straightforward way. However, as simple examples show, Move-to-root can be very inefficient, even when compared to a static BST. The Splay tree algorithm of Sleator and Tarjan [91] from 1983 is a more sophisticated (but still very simple) strategy for BST re-arrangement.

Splay is known to be very efficient and able to adapt to various kinds of useful structure in sequences of queries. Among its attractive properties, the following stands out: for all sufficiently long access sequences, the total cost of Splay for serving the sequence (including both the search and re-arrangement costs) is at most a small constant factor greater than the cost of the best *static* tree for serving the same sequence. This is called the *static optimality* property. The famous *dynamic optimality conjecture* of Sleator and Tarjan speculates that for all sufficiently long access sequences Splay is as good as any *dynamic* algorithm (again, allowing the loss of a constant factor). Since 1983, other algorithms have also been proposed as candidates for dynamic optimality, but it is still not known whether any algorithm meets this criterion.

Let us observe that the dynamic optimality conjecture consists of (at least) three, more or less distinct questions. The first question is that of computing the best sequence of BST re-arrangements for serving a given sequence of accesses. Even with advance knowledge of the access sequence, we currently have no efficient method for solving this problem. If the dynamic optimality of Splay were true, then we would have on our hands a particularly simple and efficient method for computing an optimal (up to a constant factor) sequence of re-arrangements – a task for which we currently have only exponential-time algorithms.

The second difficult question hidden in the dynamic optimality conjecture is whether there is a significant gap between online and offline algorithms in the BST model. We call algorithms such as Splay *online* to stress that they react to search queries one by one, without advance knowledge of future queries. An offline algorithm is necessarily more powerful, since it can prepare in advance for the entire sequence of queries. How much advantage does such a knowledge of the future confer to an offline algorithm? The dynamic optimality conjecture suggests that the answer is: (essentially) none.

The existence of an online algorithm as good as the best offline algorithm is not a priori guaranteed. Such an algorithm would be optimal not only in the usual *worst-case* sense, but on every single input (up to a constant factor). Even if we restrict ourselves to online algorithms, it is possible that for every input, some different heuristic works well. A third component of the dynamic optimality conjecture is the suggestion that this is not the case, and that some online algorithm – in fact, one with a very concise description – behaves well *uniformly* (i.e. on every single access sequence). The fact that such an optimal online algorithm (if indeed there exists one) adapts to every possible structure (in the BST model) can also be seen as a powerful form of *learning*.

Arguments from information theory show that for *most* access sequences, no dynamic BST algorithm can achieve better than logarithmic cost per access, a performance that is matched even by a *static* balanced BST. The quest for dynamic optimality can thus be seen as a hunt for "exotic" sequences, i.e. those that exhibit some kind of structure that can be exploited in the BST model. Several different kinds of structure have been described in the literature, but a full characterization remains elusive.

A further reason why the dynamic optimality conjecture is interesting (besides the possible practical relevance) is that the BST model appears to be in the sweet spot where optimality might be possible, but not obviously so. In computational models more flexible than the BST model, dynamic optimality is usually impossible: this is because an offline algorithm can optimally prepare for the future, for instance, by "memorizing" the answer for every possible input – something an online algorithm can hardly compete with. In models with less flexibility (e.g. the list update problem [90]), dynamic optimality is possible, but much easier to achieve. If even offline algorithms are weak, then it may be easier for online algorithms to be competitive.

For an overview of dynamic optimality and the state of our knowledge (as of 2013) we refer to the short survey of Iacono [53]. Needless to say, partly due to its composite nature, the dynamic optimality conjecture seems formidably difficult, and we do not settle it in any of its forms. Instead, we study certain specific aspects and related questions. We also attempt to untangle some of the questions comprising dynamic optimality, with the goal of identifying specific easier questions that might inspire further research on the problem. The broader topics addressed in this thesis are the following:

- What is the reason that an algorithm like Splay is efficient and others like Move-to-root are inefficient? What other algorithms may have the proven attractive characteristics of Splay? These questions are discussed in § 3. We identify simple and easily verifiable

conditions that are sufficient for a BST algorithm to match several of the properties of Splay, in other words, to be efficient in a broad sense. We also study to what extent are these conditions necessary. Our purpose with this study is (i) to better understand the connection between the relatively simple operations that Splay performs and its adaptability to sophisticated structure – a connection that is generally considered mysterious (see e.g. [31]), and (ii) to generalize Splay to a broader "design space" of efficient algorithms.

- Efficiency of a BST algorithm is understood as good performance on input sequences with a certain structure. What are the structures that facilitate fast access in the BST model? Earlier work has mostly focused (apart from some ad-hoc examples) on a broadly understood "locality of reference". This means that a query sequence should be easy if the queries have some temporal or spatial (i.e. in key-space) coherence. We identify a different kind of structural criterion that captures a more global property of access sequences. The new criterion is defined in a negative way: an access sequence is easy, if it avoids a certain *arbitrary* finite pattern as subsequence. This criterion generalizes some previously studied classes of sequences, which can be defined as avoiding a *particular* pattern. Some of these special cases are the subject of long-standing open questions, which our results partially address. We also show that there are online algorithms from the literature that perform well on sequences with such pattern-avoiding properties. This topic is explored in § 4. Besides the new insight on what makes access sequences efficient, the purpose of studying such structural properties is that they serve as concrete intermediate steps towards dynamic optimality. For instance, it is not currently known whether Splay is efficient on pattern-avoiding inputs, but we show that it must be, if it is to satisfy dynamic optimality.

- Recent progress on the BST problem has come from a surprising geometric interpretation in which many of the BST concepts and questions can be cleanly reformulated. This geometric view is due to Demaine, Harmon, Iacono, Kane, and Pătraşcu [31]. (A somewhat similar model was described earlier by Derryberry, Sleator, and Wang [34].) The results of § 4 also rely on this view. In § 5 we introduce a novel geometric view of the BST problem. This view connects binary search tree executions with rectangulations, a well-studied combinatorial structure with many uses in computer science, for example in the context of planar point location. Connections of BSTs and rectangulations with other structures, such as Manhattan networks are also presented. These connections are explored in § 5. The presented findings are still somewhat preliminary. Besides the general interest in observing hidden connections, the motivation behind these results is the hope that they lead to new angles of attack on the dynamic optimality problem.

In the remainder of this chapter we review some mathematical terminology, as well as the basic definitions and facts about binary search trees. In § 2 we define more carefully the different BST models and the algorithmic problem of serving access sequences with a BST (static or dynamic, online or offline). We review known results in this model, including BST algorithms and known lower and upper bounds on the cost of algorithms. After discussing the BST model in the classical tree-view, we also describe the geometric view of Demaine et al. and related results.

Throughout the thesis we include short pointers to relevant literature and we mention open questions, many of them long-standing. However, the work is not intended as a comprehensive survey of even this admittedly restricted topic – rather, it is a biased selection of problems and results.

1.3 Preliminaries

We review some rather standard mathematical notation used throughout the thesis. Further notation is described when needed.

- Given two *almost everywhere positive* functions $f, g\colon \mathbb{N} \to \mathbb{R}$, we use the usual asymptotic notation: both $f(n) = O(g(n))$ and $g(n) = \Omega(f(n))$ denote the fact that there exists a constant $c > 0$, such that $f(n) \le c \cdot g(n)$ *almost everywhere*. We write $f(n) = \Theta(g(n))$ if both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ hold. Both $f(n) = o(g(n))$ and $g(n) = \omega(f(n))$ denote the fact that for all constants $c > 0$, we have $f(n) \le c \cdot g(n)$ *almost everywhere*.
- For a positive integer n, we denote $[n] = \{1, \dots, n\}$.
- Given $x \in \mathbb{R}$, we denote by $\lfloor x \rfloor$ the greatest integer not greater than x, and by $\lceil x \rceil$ the smallest integer not less than x.
- By $\log n$ we denote the logarithm of n to base 2. Whenever another base is used, it is explicitly written, e.g. $\log 3 \cdot \log_3 2 = 1$. By H_n we denote the nth *harmonic number*, i.e. $H_n = \sum_{k=1}^{n} \frac{1}{k}$. The approximation $H_n = \log_e(n) + \gamma + O(1/n)$ is well-known, where $e \approx 2.7183$ and $\gamma \approx 0.5772$.
- We let $[n]^m$ denote the set of all sequences of length m with entries from $[n]$. S_n is the set of all permutations of length n. We often write permutations and other sequences in inline notation, i.e. we refer to $\pi \in S_n$ as $(\pi_1, \pi_2, \dots, \pi_n)$, where π_i denotes $\pi(i)$. When the entries are small and there is no chance of confusion, we omit the commas, and sometimes the brackets as well, e.g. $\{231, 213\} \subset S_3$.
- A *subsequence* is not necessarily contiguous, i.e. A is a subsequence of B, if it can be obtained from B by deleting an arbitrary number of entries. Given a sequence A and a value x, we denote by $A_{<x}$, $A_{\ge x}$, $A_{=x}$, etc. the subsequences consisting of all elements smaller, not smaller, or equal to x respectively.
- Two sequences $A = (a_1, \dots, a_n)$ and $B = (b_1, \dots, b_n)$ of the same length are *order-isomorphic*, if their entries have the same relative order, i.e. $a_i < a_j \iff b_i < b_j$ for all i and j. For example, (581) is order-isomorphic with (231).
- A sequence A is *π-avoiding*, if it has no subsequence that is order-isomorphic with π, otherwise we say that A *contains* π. For example, the sequence $(4, 7, 5, 2)$ contains 231 and is 123-avoiding. It is easy to show that if A is π-avoiding, then all subsequences of A are π-avoiding, and that A is σ-avoiding, for all permutations σ that contain π.

Next, we define the main concepts related to binary search trees and review some basic results. Results given without reference can be considered folklore.

A *binary tree* T is a tree with a designated root node denoted $\mathsf{root}(T)$, in which every node has zero, one, or two child nodes. The children of an arbitrary node x are denoted $\mathsf{left}(x)$ and $\mathsf{right}(x)$ (referred to as *left child* or *right child*). If the left, respectively, right child of x is missing, we set $\mathsf{left}(x) = \mathsf{null}$, resp. $\mathsf{right}(x) = \mathsf{null}$. A node whose both children are missing is a *leaf*. For an arbitrary node x in T we denote by $\mathsf{parent}(x)$ the unique node y in T, such that x is a child of y. By definition, $\mathsf{parent}(\mathsf{root}(T)) = \mathsf{null}$.

Occasionally we also use T to denote the set of nodes in T. The *size* of a tree T, denoted $|T|$, is the number of nodes in T. Observe that in a binary tree of size n, exactly $n + 1$ of all

the left(·) and right(·) pointers are null. This can be seen as saying that there are $n+1$ "slots" where subtrees can be attached to the tree.

The *depth* of a node x in a binary tree T, denoted $d_T(x)$ or simply $d(x)$ is the number of edges on the (unique) simple path from x to root(T). Nodes on this path, including x and root(T), are called *ancestors* of x. A node x is a *descendant* of y exactly if y is an ancestor of x. We call descendants and ancestors of x other than x itself *proper*. The subtree consisting of all descendants of x is called the subtree *rooted* at x, denoted subtree(x). The subtree rooted at left(x) is the *left subtree* of x, and the subtree rooted at right(x) is the *right subtree* of x. The *depth* of a binary tree T is the maximum depth of a node in T. A binary tree of size n is *balanced* if its depth is at most $c \cdot \log n$, for some constant c (we may assume $c = 2$).

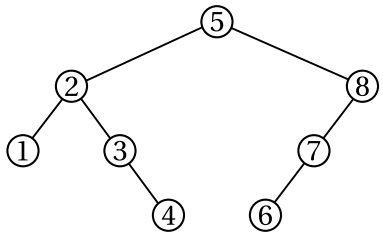

Figure 1.2: Example BST over [8] with depth 3. Node 5 is the root, nodes 1, 4, 6 are leafs, $d(3) = 2$, $d(6) = 3$, and $\mathrm{lca}(1,4) = 2$.

With each node we associate a key, usually assuming that for a tree of size n, the set of keys is $[n]$. We call such a tree a *tree over* $[n]$ and we refer interchangeably to a node and its associated key. A *binary search tree* (BST) additionally satisfies the following *ordering condition*: for every node x all nodes in the left subtree of x have keys smaller than x and all nodes in the right subtree of x have keys greater than x.

The *lowest common ancestor* of nodes x and y, denoted $\mathrm{lca}(x,y)$ is the unique common ancestor of both x and y that has maximum depth. Observe that $x < y$ implies $x \le \mathrm{lca}(x,y) \le y$. See Figure 1.2 for illustration.

The following simple observations express relations between the size and depth of a BST.

Lemma 1.1. For every $n \in \mathbb{N}$ there is a BST over $[n]$ with depth at most $\lfloor \log n \rfloor$.

Proof. We prove the following equivalent statement. Let k be such that $2^k \le n < 2^{k+1}$. Then there is a BST T of size n, with depth k. Indeed, let 2^k be the root of T, and let a BST T_L over $L = [2^k - 1]$ be the left subtree of the root, and a BST T_R over $R = \{2^k + 1, \ldots, n\}$ be the right subtree of the root. By induction, both T_L and T_R can be built such as to have depth at most $k-1$, and thus the depth of T is at most k. (At the base of the induction we have that from a single element we can build a BST of depth 0.) ∎

Lemma 1.2. In a BST of size n, the depth of more than half of the nodes is at least $\lfloor \log n \rfloor - 1$.

Proof. For each node x the path from the root to x can be encoded as a unique binary string of length $d(x)$, where a 0 corresponds to a step to a left child and a 1 corresponds to a step to a right child. Since for all $k \in \mathbb{N}$ there are $2^k - 1$ unique binary strings of length at most $k-1$ (including the empty string), the depth of more than $n - 2^k$ nodes is at least k. Setting $k = \lfloor \log n \rfloor - 1$ finishes the proof. ∎

Lemma 1.3. The average depth of a node is one less than the average size of a subtree.

Proof. Let T be an arbitrary BST, and let I be the set of proper ancestor-descendant pairs in T, i.e. $I = \{(x,y) : x, y \in T \text{ and } x \text{ is a proper ancestor of } y\}$. The indicator function $\mathbb{1}_I(x)$ equals 1 if $x \in I$ and 0 otherwise. We count I in two ways.

$$\sum_{y \in T} d_T(y) = \sum_{y \in T} \sum_{x \in T} \mathbb{1}_I\big((x,y)\big) = \sum_{x \in T} \sum_{y \in T} \mathbb{1}_I\big((x,y)\big) = \sum_{x \in T} (|\mathsf{subtree}(x)| - 1). \qquad ∎$$

Let C_n denote the number of different BSTs over $[n]$. The numbers C_n are the famous Catalan numbers [93, 98, 96, 79], given by the formula $C_n = \frac{1}{n+1}\binom{2n}{n}$. Using Stirling's approximation, one obtains the well-known estimate $C_n = \frac{4^n}{n^{3/2}\sqrt{\pi}}\left(1 + O\left(\frac{1}{n}\right)\right)$.

Catalan numbers are ubiquitous in mathematics. Stanley [96] lists over 200 different combinatorial interpretations. Besides BSTs, we mention one other Catalan structure that is used in the thesis.

Lemma 1.4. The number of different 231-avoiding permutations of length n is C_n.

Proof. We define a bijection that maps an arbitrary 231-avoiding permutation $\pi \in S_n$ to a BST T over $[n]$. Let $L = (\pi_2, \ldots, \pi_k)$ be the longest contiguous subsequence of π after π_1, consisting of elements smaller than π_1. Observe that all elements in $R = (\pi_{k+1}, \ldots, \pi_n)$ are greater than π_1, for otherwise π would have a subsequence order-isomorphic to 231. (One or both of L and R may be empty.) Furthermore, the sequences L and R are also 231-avoiding. By induction, we build a BST with π_1 as root and the BSTs recursively built of L and R as the left, respectively right subtree of the root. (The BST built of an empty sequence is empty.)

The reverse mapping is as follows: given a BST T, build a permutation π, with $\pi_1 = \mathsf{root}(T)$, followed by the sequences π_L and π_R built recursively from the left, respectively, right subtree of $\mathsf{root}(T)$. (The sequence built of an empty BST is empty.) The resulting permutation π is known as the *preorder sequence* of T.

We argue that π is 231-avoiding. By induction, π_L and π_R are 231-avoiding. Suppose for contradiction that π contains a subsequence (x, y, z), such that $z < x < y$. If $x = \pi_1$, then y cannot be in π_L, since entries of π_L are smaller than x. But then z is in π_R, contradicting $z < x$. If x is in π_L, then z must be in π_R (since otherwise π_L would contain 231), again contradicting $z < x$. Thus, x, y, and z are in π_R, contradicting that π_R is 231-avoiding. ∎

Next, we describe the elementary *rotation* operation that effects a small local change in the structure of a BST, while preserving the ordering condition. Rotations in BSTs were perhaps first used by Adelson-Velskiĭ and Landis [4] in 1962.

Suppose x and y are neighboring nodes in a BST and $x < y$. Then either $x = \mathsf{left}(y)$ or $y = \mathsf{right}(x)$ holds, i.e. the direction of the edge xy is either ／ or ＼. A rotation at the edge xy transforms the BST from one case to the other, re-attaching the children of x and y in the unique way determined by the ordering condition, leaving the remainder of the tree unchanged. A rotation requires a constant number of pointer-changes, can therefore be executed in constant time. Rotating twice at the same edge restores the original state of the tree. See Figure 1.3 for illustration. We choose this definition instead of the common way of refering to *left* and *right* rotations to avoid confusion about whether clockwise and counter-clockwise correspond to left and right or the other way around.

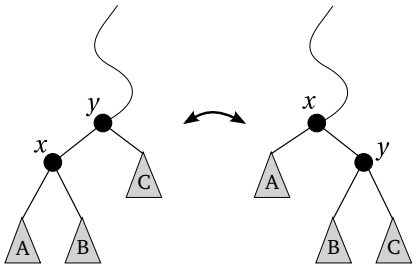

Figure 1.3: Rotation in a BST at edge xy. A, B, C denote subtrees that are unaffected by the rotation.

The following observation (Culik and Wood [55], 1982) shows that with at most a linear number of rotations it is always possible to re-arrange one BST into another.

Lemma 1.5 ([55])**.** Given two BSTs over $[n]$, there is a sequence of at most $2n-2$ rotations that transforms one into the other. Such a sequence can be found in time $O(n)$.

Proof. We show that with a sequence of at most $n-1$ rotations any BST can be transformed into the *right-leaning path*, i.e. a tree in which no node has a left child. Start with an arbitrary BST and consider the *right spine* of the tree, i.e. the longest path starting from the root, consisting only of ＼ edges. (As a special case, the right spine may consist of only the root.) Repeatedly take x, the node of smallest depth on the right spine with the property that $\mathsf{left}(x) \neq \mathsf{null}$. (If there is no such x, then we are done.) Rotate the edge $(x, \mathsf{left}(x))$. Each such

rotation increases the length of the right spine by one. Therefore, after at most $n-1$ rotations we reach the canonical tree.

Observe that once a node on the right spine has no left child, it never gains a new one. Therefore, we can find the rotation sequence by starting a cursor at the root, in each step performing either a rotation, or moving the cursor one step down on the right spine. The time for the process is thus $O(n)$. Clearly, the obtained rotation sequence is reversible. ∎

We call the minimum number of rotations necessary to transform one BST into another the *rotation distance* between the two BSTs. The following stronger result was shown by Sleator, Tarjan, and Thurston [92] in 1986.

Lemma 1.6 ([92])**.** The rotation distance between two BSTs of size n is at most $2n-6$. This is tight for sufficiently large n.

The proof of the second part of Lemma 1.6 is difficult. Recently, Pournin [84] gave a combinatorial proof and showed that the bound is in fact tight for $n \geq 11$. The following is an outstanding open question already raised implicitly in 1982 by Culik and Wood.

Problem 1. Can the rotation-distance between two BSTs be computed in polynomial time?

Questions about rotation-distance in BSTs are often studied in an equivalent setting that involves flips in triangulated convex polygons [92, 97]. Certain geometric generalizations of this problem have been shown to be NP-hard [5, 83, 67], but Problem 1 remains open.

At the end of this chapter we review a useful method for building binary search trees. Besides the BST ordering condition described earlier, we define the *heap-order* condition of a binary tree as follows. A binary tree is in heap-order if for every non-root node x we have $\mathsf{parent}(x) \leq x$. Consider a sequence of distinct integers π, and a sequence of integers τ, both of length n. Let T be a binary tree of size n, where node i is associated with the pair (π_i, τ_i), for all $i \in [n]$. We say that T is a *treap* over (π, τ), denoted as $T = \mathsf{treap}(\pi, \tau)$, if T satisfies the BST ordering condition with respect to the first entry of each node, and the heap-order condition with respect to the second entry of each node. To simplify notation, if $\pi = (1, \ldots, n)$, we denote $\mathsf{treap}(\tau) = \mathsf{treap}(\pi, \tau)$. Given π and τ, constructing $\mathsf{treap}(\pi, \tau)$ is straightforward. Treaps were studied by Seidel and Aragon [88] in 1996 and shown to be an efficient data structure if the sequence τ is chosen randomly. Essentially the same structure (sans randomness) was called Cartesian tree and described in 1980 by Vuillemin [104].

It is easy to see that if all elements of τ are distinct (e.g. if τ is a permutation), then $\mathsf{treap}(\pi, \tau)$ is unique: by the heap-order condition, the node with the minimum τ-entry must be the root, which splits the remaining nodes into left and right subtrees, according to the ordering condition on the π-entries. We can continue the process recursively. On the other hand, fixing the π-entries, the permutation τ that generates a given BST as a treap is not unique. This can be seen by comparing the number of different permutations ($n!$) with the number of different BSTs (C_n). An example treap is shown in Figure 1.4.

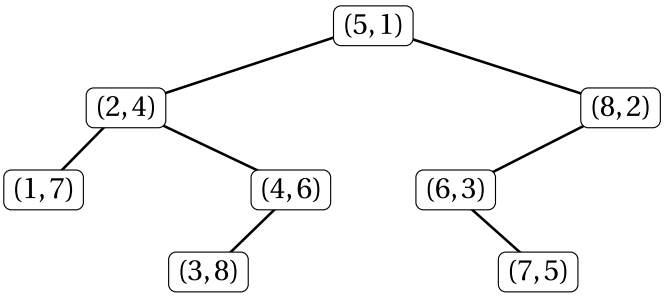

Figure 1.4: Treap constructed from the sequences $\pi = (1,2,3,4,5,6,7,8)$ and $\tau = (7,4,8,6,1,3,5,2)$.

Chapter 2

The binary search tree model

> The reason for this *geometry* is so that every militant will know only three members in the entire organization: his commander who has chosen him, and the two members that he himself has chosen.
>
> — COL. MATHIEU, *The Battle of Algiers* (1966)

In this chapter we define the BST access problem and we describe various known algorithms for this problem. A BST algorithm can be *static* or *dynamic*, and it can be *offline* or *online*. We define suitable computational models in which such algorithms can be studied, and we survey various results and questions about these models. The definitions are relied upon in all subsequent chapters.

In all cases, the problem we want to solve is to execute a sequence of accesses in a BST, each access starting from the root. In the static model the BST is fixed throughout the process, whereas in the dynamic model, the BST can be restructured between accesses, paying a certain extra cost in the present, in order to reduce the cost of future accesses. Dynamic BST algorithms can be online or offline. Both online and offline algorithms execute the sequence of accesses in the given order. The difference between the two is that online algorithms process an access sequence one element at a time, without advance knowledge of future queries, whereas offline algorithms read the entire access sequence at once and compute the strategy for the entire sequence.

In the static and the dynamic offline cases the definition of the underlying cost model is straightforward. In the case of dynamic online algorithms the situation is murkier: currently, certain statements about online algorithms can be shown only if the underlying model is sufficiently flexible. Ideally, our model should be simple and mathematically clean (to allow us to argue concisely about it), as well as realistic (i.e. close to capturing how an algorithm is plausibly implemented in a real programming language, on a real computer). As it often happens, it is difficult to satisfy both goals at the same time.

2.1 The static BST problem

Let T be a BST over $[n]$. Accessing a key $x \in [n]$ in T means starting at $\mathsf{root}(T)$ and following $\mathsf{left}(\cdot)$ and $\mathsf{right}(\cdot)$ pointers until node x is reached.

Access(T, x)
1. Let $r = \mathsf{root}(T)$.
2. If $r = x$, return YES. If $r > x$, let $r = \mathsf{left}(r)$, otherwise let $r = \mathsf{right}(r)$. Repeat 2.

Recall that we are making the (non-trivial) assumption, that x is eventually found in T. For this reason, we can assume that the assignments in step 2 always succeed. The correctness of the algorithm follows easily from the ordering condition of the BST.

We call the path between root(T) and the accessed node x the *search path* of x. We take the cost of accessing x in T, denoted $\text{cost}_T(x)$ to be the number of times step 2 is executed, which equals the number of nodes on the search path, or equivalently, one more than the depth of x in T. We thus define $\text{cost}_T(x) = d_T(x) + 1$.

Let $X = (x_1, \ldots, x_m) \in [n]^m$ be a sequence of accesses. In the standard *static* BST problem we assume that every access starts from the root (a.k.a. *root-access model*). The total cost of accessing X in T, denoted $\text{cost}_T(X)$ is then simply

$$\text{cost}_T(X) = \sum_{i=1}^{m} \text{cost}_T(x_i) = m + \sum_{i=1}^{m} d_T(x_i).$$

If T is a balanced BST, then for all $x \in [n]$ we have $\text{cost}_T(x) = O(\log n)$, and therefore, $\text{cost}_T(X) = m \cdot O(\log n)$. For certain sequences X a significantly better performance can be achieved with a well-chosen tree T.

Let us denote $\mathsf{OPT}^{\text{stat}}(X) = \min_T \text{cost}_T(X)$. In words, $\mathsf{OPT}^{\text{stat}}$ is the cost of accessing sequence X with the *best static BST* for this sequence. The term *static* refers to the fact that the tree T does not change during the execution. The quantity $\mathsf{OPT}^{\text{stat}}(X)$ is called the *static optimum* for X. Observe that $\mathsf{OPT}^{\text{stat}}(X)$ depends only on the frequencies of elements in X, and not on the exact ordering of entries. In fact, $\mathsf{OPT}^{\text{stat}}$ is known to be asymptotically equal to the *Shannon entropy* of the frequency-distribution. (Intuitively, this quantity captures how "uniform" is the distribution – the quantity is maximized if all frequencies are equal.)

The best static BST for a given sequence can be found efficiently. The dynamic programming algorithm given by Knuth [58] in 1971 achieves this in time $O(n^2)$. Mehlhorn [72] gave in 1975 a linear time algorithm that computes a solution with cost at most a small constant factor away from the optimum. Let us summarize these results.

Theorem 2.1. For an arbitrary access sequence $X \in [n]^m$ let n_i be given for every $i \in [n]$, denoting the number of times key i appears in X.

(i) [58] There is an algorithm that finds in $O(n^2)$ time a BST T with $\text{cost}_T(X) = \mathsf{OPT}^{\text{stat}}(X)$.

(ii) [72] There is an algorithm that finds in $O(n)$ time a BST T with $\text{cost}_T(X) = O\big(\mathsf{OPT}^{\text{stat}}(X)\big)$.

(iii) [72] $\mathsf{OPT}^{\text{stat}}(X) = O\left(\sum_{i=1}^{n} n_i \log \frac{m}{n_i+1}\right)$.

As far as we know, it is open whether the running time of Knuth's algorithm can be improved. Note that the Hu-Tucker and Garsia-Wachs algorithms have $O(n \log n)$ running time, but solve a variant of the problem where all keys need to be placed in the leaves of the tree [60, § 6.2.2].

Problem 2. Given the access frequencies of n elements, can the optimum static BST be computed in time $o(n^2)$?

2.2 The dynamic BST access problem

Suppose again, that we want to access a sequence of keys $X = (x_1, \ldots, x_m) \in [n]^m$ in a BST. Let us always require $m \geq n$. Again, we assume that every access starts from the root of the tree. This time, however, we allow the tree to be re-arranged during the process by performing the rotation described in § 1.3 in various parts of the tree. The purpose of the re-arrangement is to adapt the BST to the access sequence, in order to reduce the total access cost. (For instance, elements that are accessed very frequently should be brought closer to the root.) The question of how to do this efficiently is the *dynamic* BST problem.

In this case, we can no longer describe a *canonical* algorithm for accessing keys as we did in the static model. Instead, we define in a more generic way the input and output of a dynamic BST algorithm. We call such a description a *BST model*. A model can be thought of as a family of algorithms. There are different ways to formalize the various BST models, and the fact that we afford to lose small constant factors in the cost makes the task easier. For example, we need not worry about the exact cost of a rotation compared to the cost of following a pointer in the tree, as long as we can assume that the ratio between the two costs is constant.

2.2.1 Offline BST algorithms

First, we define the *offline BST model*. From several possible alternative definitions we present two, loosely following Wilber [106] and Demaine et al. [32, 31]. The first definition is perhaps more intuitive and closer to how a BST algorithm might be implemented in practice. The second definition is mathematically cleaner, and illustrates the point that the particular way in which rotations are performed does not significantly affect the cost model. Later we prove that the two models are equivalent, up to a small constant factor.

Offline BST algorithm $\mathcal{A}$ for serving X **(first model)**
1. Read the sequence $X = (x_1, \ldots, x_m) \in [n]^m$.
2. Output an initial BST T_0 over $[n]$.
3. Output a *valid* sequence $\mathcal{S} = (S_1, \ldots, S_{|\mathcal{S}|})$ of operations, consisting of elements from $\{\mathsf{moveleft}, \mathsf{moveright}, \mathsf{moveup}, \mathsf{rotate}, \mathsf{find}\}$.

We say that $\mathcal{S}$ is *valid*, if the operations $S_1, \ldots, S_{|\mathcal{S}|}$ can be executed sequentially, starting with the BST T_0, as follows. Let a cursor initially point to the root of T_0. If at step i we have $S_i \in \{\mathsf{moveleft}, \mathsf{moveright}, \mathsf{moveup}\}$, then move the cursor from its current node to the left child, right child, respectively parent node. If $S_i = \mathsf{rotate}$, then perform a rotation on the edge between the node of the cursor and its parent. If $S_i = \mathsf{find}$, then move the cursor to the root of the current tree. In addition, we require that find appears exactly m times in $\mathcal{S}$ and when it is issued the jth time, for all j, the cursor must point to the node x_j.

The cost of accessing X by algorithm $\mathcal{A}$, denoted $\mathrm{cost}_{\mathcal{A}}(X)$ is defined to be the total number of operations output by $\mathcal{A}$, i.e. $\mathrm{cost}_{\mathcal{A}}(X) = |\mathcal{S}|$.

Offline BST algorithm $\mathcal{A}$ for serving X **(second model)**
1. Read the sequence $X = (x_1, \ldots, x_m) \in [n]^m$.
2. Output an initial BST T_0 over $[n]$.
3. Output a *valid* sequence $\mathcal{Q} = (Q_1, \ldots, Q_m)$ of BSTs, each over some subset of $[n]$.

We say that $\mathcal{Q}$ is *valid*, if the trees $Q_1, \ldots, Q_m$ can be interpreted as a sequence of transformations applied to the initial BST T_0, with the following rules. For all $i \in [m]$, the keys in Q_i form a subtree of T_{i-1}, and we have $x_i \in Q_i$, and $\mathsf{root}(T_{i-1}) \in Q_i$.

The BST T_i is obtained by replacing the subtree of T_{i-1} consisting of the keys of Q_i with the tree Q_i. Observe that Q_i contains the search path of x_i in T_{i-1} (and possibly other nodes). As the exchanged subtrees are over the same set of keys, the subtrees hanging from them can be re-attached in the unique way determined by the ordering condition. We say that the nodes of Q_i are *touched at time* i.

The cost of accessing X by algorithm $\mathcal{A}$, denoted $\mathrm{cost}_{\mathcal{A}}(X)$ is defined to be the sum of the sizes of all trees in the sequence output by $\mathcal{A}$, i.e. $\mathrm{cost}_{\mathcal{A}}(X) = \sum_{i=1}^{m} |Q_i|$.

The term *offline* in the previous two definitions refers to the fact that the algorithms read the entire access sequence at once. We now define the most important quantity of this thesis.

Definition 2.2. The *offline optimum* for a sequence X, denoted $\mathsf{OPT}(X)$ is defined as $\mathsf{OPT}(X) = \min_{\mathcal{A}} \mathrm{cost}_{\mathcal{A}}(X)$, where $\mathcal{A}$ is an offline BST algorithm.

The definition of OPT depends on whether we use the first or the second model. Let us denote the respective quantities by OPT^1 and OPT^2, and let $\mathsf{OPT}(X) = \min\{\mathsf{OPT}^1(X), \mathsf{OPT}^2(X)\}$. This distinction is not essential, as we will show that the two quantities differ by a small constant factor only. Let us first observe the following fact.

Theorem 2.3. For all X, we have $\mathsf{OPT}^1(X) \le \mathsf{OPT}^{\mathrm{stat}}(X)$ and $\mathsf{OPT}^2(X) \le \mathsf{OPT}^{\mathrm{stat}}(X)$.

Proof. In both dynamic offline models we can simulate static access by restricting ourselves: In the first model, by not using rotate operations, and in the second model, by letting Q_i be the search path to x_i in T_0, for all i. The claim follows. ∎

As mentioned before, the quantity OPT is rather mysterious. It is not known, for instance, if it is possible to efficiently compute it, or to approximate it by a constant (or slightly superconstant) factor.

Problem 3. Given $X \in [n]^m$ is it possible to compute $\mathsf{OPT}(X)$ in polynomial time? What is the best approximation computable in polynomial time?

Currently the best upper bound for $\mathsf{OPT}(X)$ is the cost of the Tango tree algorithm of Demaine, Harmon, Iacono, and Pătraşcu [32] from 2004, which overestimates $\mathsf{OPT}(X)$ by a factor of $O(\log\log n)$. (See § 2.6.3.)

Computing OPT exactly is possible, if we can afford exponential-time. Observe that even the existence of an exponential-time algorithm is not obvious. As the length of the optimal sequence of operations may be $m \cdot \Omega(\log n)$, the naïve way of trying every possible sequence of pointer-moves and rotations (according to the first model) would take, for some inputs, time $c^{m \cdot \Omega(\log n)}$, for some constant c. The following algorithm was suggested to the author by Raimund Seidel.

Theorem 2.4. Given $X = (x_1, \ldots, x_m) \in [n]^m$, we can compute $\mathsf{OPT}^1(X)$ in time $O^*(4^n)$, where $O^*(\cdot)$ hides a factor polynomial in m and n.

Proof. Build a directed graph whose vertices are all C_n possible BSTs over $[n]$, annotated by a value from $[n]$ indicating the location of the cursor, and a value from $[m] \cup \{0\}$ indicating which of the m accesses have already been performed (if any). From each vertex we add edges to other vertices according to the operations moveleft, moveright, moveup, rotate, find from the first offline BST model. Recall that moveleft, moveright, moveup, and find change the location of the cursor, and rotate changes the structure of the tree. The edges corresponding to these operations will thus point to vertices where the cursor location or the tree are updated accordingly. Edges for find are used only if the cursor in the current vertex points to the next key to be accessed. A find-edge points to a vertex in which the second indicator value is incremented and the cursor is moved to the root. We also add a start vertex with edges to all vertices that have the cursor at the root and the second indicator value 0, and an end vertex with edges from all vertices that have second indicator value m. The execution of an offline algorithm with a certain cost on X is mapped now to a path from start to end of the same length, and vice versa. Finding the optimal sequence is thus reduced to finding a shortest path in an unweighted directed graph with $m \cdot O(n \cdot C_n)$ vertices and edges (every vertex has constant outdegree). Using breadth-first search, we get a total running time of $O^*(4^n)$. ∎

Theorem 2.5.

(i) Given an offline BST algorithm $\mathcal{A}$ in the first model, we can simulate it with a BST algorithm $\mathcal{B}$ in the second model, such that $\text{cost}_{\mathcal{B}}(X) \leq \text{cost}_{\mathcal{A}}(X)$, for all X.

(ii) Given an offline BST algorithm $\mathcal{B}$ in the second model, we can simulate it with a BST algorithm $\mathcal{A}$ in the first model, such that $\text{cost}_{\mathcal{A}}(X) = O\big(\text{cost}_{\mathcal{B}}(X)\big)$, for all X.

Proof.

(i) Let algorithm $\mathcal{B}$ output the same initial tree that $\mathcal{A}$ outputs. Let f_i denote the index in $\mathcal{S}$ of the find operation corresponding to the access x_i. For all $i \in [m]$ consider the contiguous subsequences $\mathcal{S}_i = (S_{f(i-1)+1}, \ldots, S_{f(i)})$, with the convention that $f(0) = 0$. Observe that $\mathcal{A}$ executes $\mathcal{S}_i$ starting with the tree T_{i-1}, and ending with the tree T_i. Let R_i denote all the nodes to which the cursor points at least once while $\mathcal{A}$ executes $\mathcal{S}_i$. The nodes R_i form a subtree of T_i, which we denote by Q_i. (This can be seen by induction, and easily verified for all five types of operations in $\mathcal{S}_i$.) Furthermore, R_i contains the root of T_{i-1} (since the cursor is there at the start of $\mathcal{S}_i$), and R_i contains the accessed key x_i (since the cursor is there before executing the find operation $S_{f(i)}$). We can now define a sequence of trees $\mathcal{Q} = (Q_1, \ldots, Q_m)$ that are a valid sequence of transformations by the rules of the second model, and which produce exactly the behavior of algorithm $\mathcal{A}$. We conclude $\mathcal{B}$ by outputting $\mathcal{Q}$.

The cost of $\mathcal{B}$ is $\sum_{i=1}^{m} |Q_i|$, which is exactly the number of nodes visited by the cursor during the execution of $\mathcal{A}$. Since only moveleft and moveright operations can make the cursor visit a new node, and at most one new node is visited for either of these operations, we have $\text{cost}_{\mathcal{B}}(X) \leq \text{cost}_{\mathcal{A}}(X)$.

(ii) Let algorithm $\mathcal{A}$ output the same initial tree that $\mathcal{B}$ outputs. In $\mathcal{A}$ we have to simulate the transformation from the subtree of T_{i-1} with the nodes of Q_i to Q_i. First we rotate the subtree with nodes of Q_i to a right-leaning path. Starting with the cursor from the root, we can achieve this with at most $|Q_i| - 1$ rotations and at most $|Q_i| - 1$ moveright operations (following the process described in the proof of Lemma 1.5). Since the rotate operation is defined between a node and its parent, and we need to rotate between a node and its left child, we need an additional moveleft operation before each rotate. Then we execute the reverse process, transforming the right-leaning path over the nodes

of Q_i into the tree Q_i. Again, the number of operations is proportional to $|Q_i|$. After the transformation is finished, we need to move the cursor from the root to the element x_i, and issue the find operation.

By Lemma 1.5, the total number of operations we issue to simulate the transformation and the access is $O\big(|Q_i|\big)$. The claim follows. ∎

We finish our discussion of the offline model by mentioning two natural restrictions on offline BST algorithms. The first is to require that for all i, the accessed key x_i is moved to the root during the re-arrangement, before x_{i+1} is accessed. We call this the *access-to-root* restriction. (In the context of the second model, this means that for all i, node x_i is the root of Q_i.) Denote by $\mathsf{OPT}^{\mathrm{root}}(X)$ the cost of the best algorithm for X that conforms to this restriction.

The second restriction is to perform re-arrangements in the BST only on the search path of the current access. We call this the *search-path-only* restriction. (In the context of the second model, this means that Q_i contains only nodes of the search path for x_i in T_{i-1}.) Denote by $\mathsf{OPT}^{\mathrm{sp}}(X)$ the cost of the best algorithm for X that conforms to this restriction. Let $\mathsf{OPT}^{\mathrm{sp,root}}(X)$ denote the cost of the best algorithm for X that conforms to both the access-to-root and the search-path-only restrictions. The following observation is due to Wilber [106].

Lemma 2.6 ([106])**.** $\mathsf{OPT}^{\mathrm{root}}(X) = \Theta\big(\mathsf{OPT}(X)\big)$, for all X.

Proof. Let $\mathcal{A}$ be an offline algorithm in the first model that *does not* necessarily move the accessed element to the root. For all i, before issuing the ith find operation, if the cursor (pointing to x_i) is not at the root already, we rotate it to the root with a number of rotations equal to one less than the depth of x_i. Observe that this is at most the number of operations that were used for moving the cursor from the root to x_i. We start the next access x_{i+1} by undoing the rotations we performed to bring x_i to the root, and moving the cursor back to the root. We then proceed normally to access x_i. We increased the total cost only by a small constant factor. ∎

The relations $\mathsf{OPT}^{\mathrm{sp,root}}(X) \geq \mathsf{OPT}^{\mathrm{sp}}(X) \geq \mathsf{OPT}(X)$ hold by definition, since the minimum cost can not decrease if we compute it over a more restricted set of algorithms. With the search-path-only restriction, the argument of Lemma 2.6 does not seem to work, therefore the following basic question is open.

Problem 4. Does it hold for all X that $\mathsf{OPT}^{\mathrm{sp,root}}(X) = O\big(\mathsf{OPT}^{\mathrm{sp}}(X)\big)$?

Lucas [68] describes an ever stronger restriction on BST algorithms: allow only those rotations that cause the depth of the current access x_i to *strictly decrease*. She conjectures that the optimum can be assumed to take this kind of canonical form. We remark that the Splay algorithm (described in § 2.5.2) meets all the restrictions described, therefore, if the cost of Splay matches OPT (as conjectured by Sleator and Tarjan), then the answer to Problem 4 (as well as to the conjecture of Lucas) is YES.

2.2.2 Online BST algorithms

Next, we describe the *online BST model*. We present a number of alternative definitions that differ in the amount of restrictions they place on an online BST algorithm. In contrast to the offline models, in the online case it is not clear whether there is a significant difference in power between the various models.

Our first definition is a *strict model*, inspired by the Splay algorithm, which is seen in § 2.5.2 and § 3.2 to conform to this model.

Online BST algorithm $\mathcal{A}$ for serving X **(strict model)**
1. Output an initial BST T_0 over $[n]$.
2. For each $i = 1, \ldots, m$:
   2.1. Read key $x_i \in [n]$.
   2.2. Let P_i be the *encoding* of the search path to x_i in T_{i-1}.
   2.3. Let $Q_i = \Gamma(P_i)$, and compute T_i by *replacing* the search path to x_i in T_{i-1} by Q_i.

In step 2.2, by an encoding P_i of the search path to x_i we mean a binary string of the same length as the search path, whose jth entry is 0 if the jth edge on the path from the root to x_i is to the left (i.e. $\diagup$), and 1 if the edge is to the right (i.e. $\diagdown$). The binary string P_i fully describes the ordering of the keys on the search path, but ignores the actual key values. The search path P_i is mapped in step 2.3 to a BST $Q_i = \Gamma(P_i)$ of size $|P_i| + 1$, according to a fixed function Γ. We call Γ the *transition function* of algorithm $\mathcal{A}$ and require Γ to be defined for all binary strings of length at most $n-1$. The BST $Q_i = \Gamma(P_i)$ is over $[|P_i| + 1]$, i.e. it may contain different keys than those of the search path. The keys from the search path can, however, be mapped in a unique way to the nodes of Q_i, governed by the ordering condition. After this relabeling, the replacement of the search path with the new tree is done similarly as in the second model of offline algorithms.

The cost of accessing X by algorithm $\mathcal{A}$, denoted $\mathrm{cost}_{\mathcal{A}}(X)$ is defined as the total number of nodes in all search paths, or equivalently as $\mathrm{cost}_{\mathcal{A}}(X) = \sum_{i=1}^{m} |Q_i|$.

The definition of the strict model is not standard, let us therefore make some remarks on the particular choices. First, observe that an algorithm in the strict model (called a *strict online algorithm*) is rather mechanistic: it consists of an initial BST T_0 and a fixed transition function Γ from search paths to BSTs. Such an algorithm lives in an "eternal present" – as it carries no internal state from one access to another, it has no "knowledge" of previous or future accesses, or even of its position within the access sequence.

The reader may wonder why we restrict the algorithm to replace (re-arrange) the search path only, instead of a possibly larger subtree, as in the case of offline algorithms. The main reason is to stay fully within a comparison-only model: to wander away from the search path would mean that the algorithm must base its decisions on administrative details such as whether a pointer is null or not. The second reason is that we want strict algorithms to operate only on the keys, forbidding them to the extent possible from "taking notes", i.e. of carrying state from one access to the other. If we allow an algorithm to inspect and modify arbitrary parts of the tree, then it might encode some information in the particular re-arrangements it makes in certain "out-of-view" parts of the tree. We feel such cleverness to be out of place in the strict model, since we want the algorithm to behave "uniformly", i.e. to react the same way under the same circumstances. Finally, re-arranging only the search path is natural because a cost proportional to its length is already incurred by the access, so we can conveniently forget about the cost of re-arrangement, which is (asymptotically) subsumed by the cost of

access. (Recall that a BST re-arrangement can be done with a linear number of operations, by Lemma 1.5.)

For an access sequence X we define $\mathsf{OPT}^{\mathrm{str}}(X)$ to be the cost of the best strict online BST algorithm for X, i.e. $\mathsf{OPT}^{\mathrm{str}}(X) = \min_{\mathcal{A}} \mathrm{cost}_{\mathcal{A}}(X)$, where $\mathcal{A}$ conforms to the strict online model. The following observation holds.

Theorem 2.7. For all X, we have $\mathsf{OPT}^{\mathrm{str}}(X) \leq \mathsf{OPT}^{\mathrm{stat}}(X)$.

Proof. In the strict online model we can simulate static access by restricting ourselves, letting Q_i be the unchanged search path to x_i in T_0, for all i. ∎

Remark. For a strict online algorithm to be practical, we should require it to compute $\Gamma(P_i)$ using only polynomial time and space. This may seem contradictory, since the size of Γ is clearly exponential. (There are 2^n binary strings of length n.) Nonetheless, Γ may have a concise implicit description, for instance, as a set of rules, e.g. "if there are three consecutive left-edges, then rotate the one in the middle", and so on. While a polynomial bound on the description of Γ seems reasonable, we can not rule out the possibility that the most efficient algorithm in this model may have a large and incompressible transition function. Such an algorithm may be interesting theoretically, although possibly unwieldy to argue about. In this thesis we mostly ignore this aspect of the problem.

Let us also define a few variants of the strict model. First, we can further restrict a strict online algorithm to require that the accessed key is moved to the root during the re-arrangement. (This means that x_i is the root of Q_i, for all i.) We denote by $\mathsf{OPT}^{\mathrm{str,root}}(X)$ the cost of the best strict online algorithm for X that conforms to this restriction.

The following inequalities result simply from the fact that the optimum over a more restricted family of algorithms cannot be smaller.

Theorem 2.8. For arbitrary X, we have

$$\mathsf{OPT}^{\mathrm{str,root}}(X) \geq \mathsf{OPT}^{\mathrm{str}}(X) \geq \mathsf{OPT}^{\mathrm{sp}}(X) \geq \mathsf{OPT}(X).$$

We remark that Splay (§ 2.5.2) conforms to the strict online model even with the restriction that the accessed key is moved to the root. Therefore, if Splay is dynamically optimal, then the inequalities in Theorem 2.8 collapse to equalities (up to a constant factor). However, each inequality poses a meaningful (easier) question in itself, and settling any of them would yield insight such as whether it is ever worth *not* bringing the accessed element to the root, whether there is an online-offline gap in the BST model, and whether it is ever worth wandering off the search path.

Problem 5. Do any of the following statements hold for all X?

- $\mathsf{OPT}^{\mathrm{str,root}}(X) = O\big(\mathsf{OPT}^{\mathrm{str}}(X)\big)$,
- $\mathsf{OPT}^{\mathrm{str}}(X) = O\big(\mathsf{OPT}^{\mathrm{sp}}(X)\big)$,
- $\mathsf{OPT}^{\mathrm{sp}}(X) = O\big(\mathsf{OPT}(X)\big)$.

Theorem 2.8 and Problem 5 concern the cost of the optimum algorithm *for a given sequence*. We do not know however, if there are concrete algorithms that match these optima *for all sequences*. The original dynamic optimality conjecture asks essentially the following.

Problem 6. Is there a strict online algorithm $\mathcal{A}$ such that for all X we have $\mathrm{cost}_{\mathcal{A}}(X) = O\big(\mathsf{OPT}(X)\big)$?

Based on the previous discussion, we can formulate different questions that are similarly open, which could however be significantly easier than dynamic optimality. In particular, the following question seems interesting.

Problem 7. Is there a strict online algorithm $\mathcal{A}$ such that for all X we have $\mathrm{cost}_{\mathcal{A}}(X) = O\big(\mathsf{OPT}^{\mathrm{str,root}}(X)\big)$?

We now define an online model that is less restricted than the previous one. We call it the *lenient model*. It is in some sense closer to the capabilities of a real implementation and it also captures some of the known algorithms from the literature that fall outside the strict model. The main differences of the lenient model compared to the strict model are that in the lenient model an algorithm may store a small number of bits of annotation at each node and it may also perform operations outside the search path. Algorithms with similar capabilities are sometimes called *real-world* BST algorithms in the literature [53].

Online BST algorithm $\mathcal{A}$ for serving X **(lenient model)**

1. Output an initial BST T_0 over $[n]$.
2. Output initial annotations $f\colon [n] \to \{0,1\}^L$, where $L = o(\log n)$.
3. For each $i = 1, \ldots, m$:
   3.1. Read key $x_i \in [n]$.
   3.2. Output a *valid* sequence $\mathcal{S}_i$ of operations, consisting of elements from $\big\{\mathsf{moveleft}, \mathsf{moveright}, \mathsf{moveup}, \mathsf{rotate}, \mathsf{read}, \mathsf{write}(\cdot), \mathsf{compare}\big\}$.

We say that $\mathcal{S}_i = (S_1, \ldots, S_{|\mathcal{S}_i|})$ is *valid*, if the operations in $\mathcal{S}_i$ can be executed sequentially, starting with the BST T_{i-1} and resulting in the BST T_i as follows. For all i, just after reading key x_i (a.k.a. "at time i"), let a cursor point to the root of T_{i-1}. If at step j of executing $\mathcal{S}_i$, we have $S_j \in \{\mathsf{moveleft}, \mathsf{moveright}, \mathsf{moveup}\}$, then move the cursor from its current node to the left child, right child, respectively parent node. If $S_j = \mathsf{rotate}$, then perform a rotation on the edge between the node of the cursor and its parent.

If $S_j = \mathsf{write}(s)$, for some bit string s of length at most L, then set the value of the annotation $f(\cdot)$ at the cursor to s. If $S_j = \mathsf{read}$, then read out the annotation at the cursor into a variable. If $S_j = \mathsf{compare}$, then compare the current key value x_i with the key value at the cursor and save the result in a variable.

The algorithm can maintain internal variables, but it is *not* allowed to carry any state over from one iteration to the next (i.e. its internal variables are reset at each time i, in step 3.1), except via the annotations on the nodes. The annotations read from the nodes and the outcomes of the comparisons can influence the future operations generated in the sequence $\mathcal{S}_i$. In addition, we require that in each sequence $\mathcal{S}_i$ there is a $\mathsf{compare}$ operation issued when the cursor points to the node x_i. (This plays the same role as the find operation in the first offline model.)

The cost of accessing X by algorithm $\mathcal{A}$, denoted $\mathrm{cost}_{\mathcal{A}}(X)$ is defined to be the total number of operations output by $\mathcal{A}$, i.e. $\mathrm{cost}_{\mathcal{A}}(X) = \sum_{i=1}^{m} |\mathcal{S}_i|$.

Again, we ignore the *computational* cost of producing the outputs $\mathcal{S}_i$, although we should expect any reasonable algorithm to perform its computations in polynomial time and space.

The intention in allowing annotations is to let the algorithm maintain simple balancing and other bookkeeping information. Indeed, some BST algorithms from the literature such

as red-black trees or Tango do exactly this. By using annotations, we also avoid the issue of comparing with null pointers, since we can keep track (using a few bits and a small extra cost) of which neighbors of a node are present. We can thus assume that the issued moveleft, moveright, moveup, and rotate operations are always valid.

We limit the number of bits to $o(\log n)$, in order to disallow the storage of global depths, counts of nodes, and other complex information that would require a logarithmic number of bits. Another motivation for this choice is that in this way, the cost of comparing keys (which may take $\Omega(\log n)$ time in realistic models) still dominates other bookkeeping and annotation costs.

The reason for resetting the internal state before every access is to forbid the algorithm from remembering the structure of the tree. In this way, except for the limited help given by the annotations, the structure of the tree can be inferred only by exploring it with pointer moves starting from the root (and by paying a proportional cost).

All in all, we tried to define the lenient model such as to be as limited possible, while still capturing the best *known* online algorithms.

We define the optimum over algorithms conforming to the lenient model (called lenient online algorithms) in a similar way as before: for a given X, let $\mathsf{OPT}^{\text{len}}(X) = \min_{\mathcal{A}} \text{cost}_{\mathcal{A}}(X)$, where $\mathcal{A}$ is a lenient online algorithm. A lenient algorithm can simulate a strict algorithm restricting itself to re-arranging only the search path, and not using annotations. Thus we have the following.

Theorem 2.9. For arbitrary X, we have

$$\mathsf{OPT}^{\text{str}}(X) \geq \mathsf{OPT}^{\text{len}}(X) \geq \mathsf{OPT}(X).$$

Intuitively, $\mathsf{OPT}^{\text{len}}$ seems closer to OPT than to $\mathsf{OPT}^{\text{str}}$. This is because a lenient online algorithm is able to perform the same re-arrangements that an offline algorithm does, apart from the fact that an offline algorithm might choose to do different re-arrangements at different times, even if the current tree and the accessed key are the same (such as to prepare for future queries). To some extent, a lenient algorithm that is lucky enough to "guess what it should do" can also simulate this with the use of annotations. However, for arbitrarily long access sequences (i.e. if m is very large), an online algorithm cannot encode all sequences of possible re-arrangements even if we allow space and time resources superpolynomial in n. Thus the following are open.

Problem 8. Do any of the following statements hold for all X?

- $\mathsf{OPT}^{\text{str}}(X) = O\big(\mathsf{OPT}^{\text{len}}(X)\big)$,
- $\mathsf{OPT}^{\text{len}}(X) = O\big(\mathsf{OPT}(X)\big)$.

If the second statement of Problem 8 holds, then the lenient model is rich enough to contain a good algorithm for every access sequence. This statement seems quite plausible. However, even in this case, it does not immediately follow that there is a single lenient online algorithm that is good for all sequences.

The following question asks exactly this (again, this is simply the dynamic optimality question for lenient algorithms).

Problem 9. Is there a lenient online algorithm $\mathcal{A}$ such that for all X we have $\text{cost}_{\mathcal{A}}(X) = O\big(\mathsf{OPT}(X)\big)$?

Finally, we describe an online model with even more capabilities than the lenient model. We call this the *unlimited* online model, and we define it such as to differ as little as possible from the *offline model*.

Recall that an algorithm conforming to the second offline model outputs an initial tree T_0, and a sequence of BSTs $Q_1, \ldots, Q_m$ encoding the re-arrangement of the tree, as a function of the query sequence $(x_1, \ldots, x_m)$. Let us restrict an algorithm only in the fact that $T_0, Q_1, \ldots, Q_m$ must not depend on the future.

Online BST algorithm $\mathcal{A}$ for serving X **(unlimited model)**
1. Output an initial BST T_0 over $[n]$.
2. Read the sequence $X = (x_1, \ldots, x_m) \in [n]^m$.
3. Output a *valid* sequence $\mathcal{Q} = (Q_1, \ldots, Q_m)$ of BSTs, where $Q_i = \Lambda_i\big((x_1, \ldots, x_i)\big)$.

The sequence $\mathcal{Q}$ encodes tree-rearrangements in the exact same way as in the offline case. The cost of an algorithm is also computed the same way. An algorithm conforming to the unlimited online model is uniquely determined by the initial tree T_0 and by the mappings Λ_i from access sequences to trees.

We may define the optimum in this model as $\min_{\mathcal{A}} \mathrm{cost}_{\mathcal{A}}(X)$, for a given X, where $\mathcal{A}$ conforms to the unlimited online model. Observe, however, that this quantity is exactly the same as the offline optimum $\mathsf{OPT}(X)$. This is because for every particular sequence X, there is an unlimited online algorithm that "happens to do the same" for X as the offline optimum. (Such an algorithm may perform arbitrarily badly on inputs other than X.) This does not imply, however, that there is an unlimited online algorithm that matches the optimum *for every sequence*. This is, in fact, another variant of the dynamic optimality conjecture.

Problem 10. Is there an unlimited online algorithm $\mathcal{A}$ such that for all X we have $\mathrm{cost}_{\mathcal{A}}(X) = O\big(\mathsf{OPT}(X)\big)$?

We may ask the easier question of whether there exists an unlimited online algorithm $\mathcal{A}$ competitive with the *online optimum*, e.g. with $\mathsf{OPT}^{\mathrm{str}}$. Such a result is implied by work of Iacono [53], which we describe in § 2.3.

The unlimited online model is intended only as a theoretical construction. We find it interesting that even in such a strong model, the dynamic optimality question is unresolved. Otherwise, the model is highly unrealistic. First, we have not placed any restriction on how the functions Λ_i are stored or computed. To explicitly store $\Lambda_1, \Lambda_2, \ldots$, for an arbitrarily long sequence of accesses would require unbounded memory. It would be reasonable, therefore, to restrict the functions Λ_i to depend only on the current tree T_i and access x_i, instead of the entire history of accesses. For an optimal online algorithm to behave differently in two identical states (depending on the history of accesses) seems to amount to a *gambler's fallacy*. More importantly, even if the action of the algorithm depends only on the current tree, it is unreasonable that the algorithm should "know" the entire structure of the tree without paying for its exploration. The lenient model described earlier was designed to address exactly these issues of the unlimited model.

Initial trees. A somewhat subtle issue for online algorithms (and one that is often swept under the rug) is that of the initial tree T_0. Observe that all models defined so far leave it to the discretion of the algorithm to choose an initial tree before serving an access sequence. The initial tree can therefore be thought of as being a part of the description of the algorithm (i.e. independent of the input). We can think of an algorithm $\mathcal{A}$ as a family of algorithms $\{\mathcal{A}_T\}$, one algorithm for each choice of the initial tree T. (Note however, that for certain dynamic

algorithms not every BST T over $[n]$ may be a valid state, and thus, a valid initial tree. This can not happen in the strict model, where every tree is valid.)

Whenever we say that algorithm $\mathcal{A}$ has property $\mathcal{P}$, we should specify which version $\mathcal{A}_T$ of the algorithm we are referring to, i.e. with which initial tree the property holds. In some cases we would like to make the stronger statement that $\mathcal{P}$ holds for $\mathcal{A}_T$ for all choices of valid initial trees T, i.e. we may say that $\mathcal{A}$ has property $\mathcal{P}$ even with the worst initial tree. Indeed, most known results for Splay hold in this form, and dynamic optimality is also conjectured to hold regardless of initial tree. (Intuitively, the effect of the initial tree should wane for sufficiently long access sequences, although a precise characterization of such a behavior is still missing.)

The question of initial trees is largely irrelevant for offline algorithms, since an offline algorithm can always start by rotating the initial tree into a tree of its choosing. By Lemma 1.5, this contributes only a linear additive term to the total cost. With unlimited and lenient online algorithms, it may seem that we can similarly sidestep the issue of the initial tree, since a certain agreed-upon initial pattern of annotations can alert the algorithm that it has just begun accessing the sequence, and it can thus transform the tree into an arbitrary initial tree, again, only with a linear additive term in the cost. However, even if we may choose an arbitrary initial tree, we may not choose one depending on the (future) access sequence. What we may nevertheless assume about unlimited and lenient online algorithms is that they have no particularly "bad" initial trees – if there were such a tree, then the algorithm would start by changing it.

For strict online algorithms the issue of initial trees is crucial, since these algorithms do not know at what stage they are within the access sequence. The choice of initial tree can significantly affect the total cost of the algorithm, especially if the access sequence is short. Therefore, in the strict online model we do not find satisfactory a statement that algorithm $\mathcal{A}$ has property $\mathcal{P}$ with an initial tree that depends on $\mathcal{P}$, i.e. on the sequence to be read. (Even if in some cases the best results we currently have are of this type.)

We define what it means for an online algorithm to be competitive (this is merely a different name for approximation-ratio).

Definition 2.10. For some c (possibly depending on n), we say that an online algorithm (strict, lenient, or unlimited) $\mathcal{A}$ is *c-competitive*, if for every access sequence $X \in [n]^m$ with $m \geq n$, we have $\mathrm{cost}_{\mathcal{A}}(X) \leq c \cdot \mathsf{OPT}(X)$.

We require $m \geq n$, only to avoid the uninteresting case of short sequences where OPT can be very small, but an otherwise good online algorithm may have high cost (due to the choice of a bad initial tree). Needless to say, a competitiveness-result is still highly interesting if it holds only for $m \geq f(n)$, for some $f(n) = \omega(n)$.

The following are perhaps the most important open questions in the online BST model.

Problem 11. Is there an $o(\log n)$-competitive strict online algorithm? Is there an $o(\log\log n)$-competitive lenient or unlimited online algorithm?

From the previous discussion, it is clear that for every algorithm $\mathcal{A}$ in any of the defined models, with every initial tree T_0, and for every access sequence X, it holds that $\mathsf{OPT}(X) \leq \mathrm{cost}_{\mathcal{A}}(X)$. We repeat that the original form of the dynamic optimality conjecture states that Splay, a particularly simple algorithm in the strict online model with the access-to-root restriction has cost proportional to OPT for every access sequence. Proving this conjecture would be highly desirable, as this would settle Problems 3–11, and would let us avoid further questions about various models with particular subsets of restrictions and capabilities.

2.2.3 A simple algorithm: Rotate-once

To make the discussion more concrete, we describe a simple online BST algorithm called Rotate-once, introduced by Allen and Munro [7] in 1978. The algorithm is well-defined for any initial BST, and it works in the strict online model (a proof of this fact is left as exercise). We only specify the operation when accessing a key x_i, assuming that the state of the BST before the current access is T.

Rotate-once(T, x_i)
1. Access(T, x_i)
2. If $\mathsf{parent}(x_i) \neq \mathsf{null}$, then rotate the edge $\big(x_i, \mathsf{parent}(x_i)\big)$.

Step 1 is an access as in a static tree, following the search path from $\mathsf{root}(T)$ to x_i. Step 2 is a single rotation that brings x_i closer to the root (unless x_i is already the root). The rotation can be implemented with only constant additional cost, since after finishing step 1 we have a cursor pointing to x_i. In any case, since we are in the strict model, we only re-arrange the search path, so we can ignore the cost of rotation, as it is subsumed by the cost of access.

We show an example where Rotate-once works very well. Suppose we start with the *right-leaning path* over $[n]$ as initial tree, and we access the sequence $S = (1, 2, \ldots, n)$. The cost is easily seen to be constant for each access. (At the time of its access, each key is at depth at most 1.) Since we need at least a constant cost per access, the obtained cost is asymptotically optimal, i.e. $\mathrm{cost}(S) = \mathsf{OPT}^{\mathrm{str}}(S) = O(n)$. From Lemma 1.2 it follows that $\mathsf{OPT}^{\mathrm{stat}}(S) = \Omega(n \log n)$, in fact the same would hold for any permutation sequence S. We have thus shown a separation between $\mathsf{OPT}^{\mathrm{str}}$ and $\mathsf{OPT}^{\mathrm{stat}}$, illustrating that a dynamic algorithm can be significantly more efficient than even the best static tree.

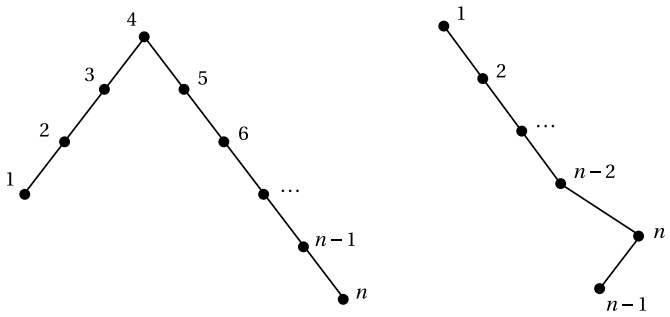

Figure 2.1: Rotate-once execution.
(left) Good example after accessing $(1, 2, 3, 4)$, before accessing $(5, \ldots, n)$.
(right) Bad example after accessing $(n, n-1, n)$, before accessing $(n-1, n, \ldots)$.

Rotate-once is in general, however, not very efficient. Consider the previous example of accessing S, with the *left-leaning path* over $[n]$ as initial tree. The total cost can be seen to be $\Omega(n^2)$. As a different example, consider the access sequence S' alternating between accessing n and $n-1$ for a sufficiently long time, i.e. $S' = (n, n-1, n, n-1, \ldots)$, again with the right-leaning path over $[n]$ as initial tree. Now each access takes time $\Omega(n)$, whereas a simple static tree with n and $n-1$ placed close to the root would achieve $O(1)$ cost per access. Thus we have shown that Rotate-once is not c-competitive for any $c = o(n)$. See Figure 2.1 for illustration. We remark that the choice of a bad initial tree is not essential in either of the two bad examples. It is easy to think of a sequence of accesses that forces Rotate-once into the state of a right-leaning path or left-leaning path (or any other state), regardless of the initial tree.

2.3 Other models

We describe further ideas and techniques from the literature and we speculate about their possible relevance to the dynamic optimality question. Some of the described results stretch the limits of the BST model by requiring an "unreasonable" amount of memory or running time (in the spirit of the unlimited online model described in § 2.2.2), or they augment the BST model with further capabilities. Studying such extended models is interesting theoretically, but ultimately we would like to be able to "simulate" them in a more realistic BST model.

2.3.1 Meta-algorithms

Multiple authors have suggested the application of general techniques from machine learning that can "simulate" and "combine" multiple algorithms from a broad family.

The best online algorithm. Iacono [53] describes an online BST algorithm that essentially "simulates" all possible online BST algorithms. This "meta-algorithm" is based on the "multiplicative weights update" (MWU) technique.

Informally, MWU solves the following problem: a sequence of events are revealed one by one (e.g. by nature or by the stock market). In every round, *before the event is revealed*, we are to choose an appropriate response for it. Depending on the suitability of our response, we receive a certain payoff (or penalty). Before every choice, we can consult a number of experts, each of whom reveal their own responses (likewise, before the event), and we can decide which one to follow. After each event, we can evaluate how each of the experts performed. The MWU technique allows us to make our choices such that our total payoff over the long term is not far from that of the best expert (with essentially no further assumptions). This is achieved by maintaining "scores" (a.k.a. "weights") for the experts based on their past performance, suitably updated after every event (this is when "multiplication of weights" takes place). Experts with higher score are more likely followed in the future. For a comprehensive treatment of the MWU technique, we refer to the survey of Arora, Hazan, and Kale [8].

Let us sketch the idea of Iacono's approach for an online BST meta-algorithm. We consider the input access sequence in epochs (i.e. contiguous subsequences) and for each epoch we choose one of the possible online algorithms to execute, using the MWU technique. (Here the access sequence within the epoch is the "event", and the various online algorithms are the "experts".) We also evaluate how the other online algorithms would have done on the epoch, and update their scores according to the MWU rules.

To correctly simulate an online algorithm $\mathcal{A}$ within an epoch, the meta-algorithm has to bring the tree at the beginning of the epoch into the state in which $\mathcal{A}$ would be, if it had been running from the beginning. (Our description differs here from the one in [53].) Assuming that the epochs are sufficiently long, i.e. of length $\Omega(n)$, the cost of this transformation at the beginning of the epoch does not affect the total cost by more than a small constant factor.

The known bounds for the MWU technique tell us that over a sufficiently long sequence, the meta-algorithm does not perform much worse than "the best expert in hindsight", i.e. it is competitive with the best online algorithm in the simulated class of algorithms. In this way, we can obtain an unlimited online algorithm whose cost matches $\mathsf{OPT}^{\mathrm{str}}$ or even $\mathsf{OPT}^{\mathrm{len}}$. We stress that the cost of the meta-algorithm includes a huge additive term, so the result holds only for sufficiently long access sequences.

The result implies that if we had a non-constructive argument for the *existence* of a dynamically optimal online algorithm, then we could turn it into a constructive result, i.e. into an actual online algorithm, as impractical as it may be. (Problem 7 asks whether a similar result can be shown in a more restricted online model.)

We have not specified the exact online BST model assumed for this result. From the above description it is clear that the meta-algorithm needs to reside in a model with vastly more resources than the "simulated" model. The unlimited model is clearly sufficient to simulate e.g. the strict or the lenient model. In fact, we do not need the full power of the unlimited model: as the number of possible strict or lenient online algorithms is a function of n only, the meta-algorithm can also be implemented with time and space requirement depending on n only (i.e. not on m).

Dynamic optimality? We observe that if the meta-algorithm outlined above could simulate unlimited online algorithms, then it would be dynamically optimal (since the optimum over unlimited online algorithms matches OPT, see § 2.2.2). We cannot directly do this, however, since unlimited online algorithms require unbounded memory.

Let us instead consider a more restricted family of online algorithms and see under what conditions would the meta-algorithm achieve dynamic optimality. Consider again the access sequence in epochs of length n, and let $x_1, \ldots, x_n$ be the accesses in the current epoch. Consider the ith access x_i, and assume that the online algorithm transforms the current tree T_{i-1} into the tree T_i. The transformation $T_{i-1} \to T_i$ is governed by a function Λ, i.e. $\Lambda(T_{i-1}) = T_i$. We let Λ depend on the current access x_i, as well as on the past accesses in the current epoch, i.e. on $x_1, \ldots, x_{i-1}$. Let $\mathcal{F}$ denote the family of online algorithms thus defined.

Observe that algorithms in $\mathcal{F}$ are more restricted than unlimited online algorithms (their space requirement is a function of n only). The number of online algorithms in $\mathcal{F}$ is a function of n, and therefore, the meta-algorithm outlined above can simulate this family (and match the optimum over it). Dynamic optimality boils down to whether $\mathcal{F}$ is rich enough to contain an algorithm matching OPT. We don't know whether this is the case; let us, however, make some simple related observations.

Clearly, the optimal *offline* algorithm may also be assumed to perform actions depending on the current epoch only. (At the epoch boundaries we can transform the tree into a canonical state, increasing the cost by a small factor only.) Consider again the ith access x_i, and assume that the offline algorithm transforms the current tree T_{i-1} into the tree T_i. Whatever effect the past accesses $x_1, \ldots, x_{i-1}$ may play on our current transformation is already captured by the current tree T_{i-1}. Thus, we may assume that the transformation $T_{i-1} \to T_i$ is governed by a function Λ' that depends on the current access x_i, as well as on the *future* accesses in the current epoch, i.e. on $x_{i+1}, \ldots, x_n$, but not on the past accesses.

An obvious first observation is that if Λ' depends on x_i only (and not on the future accesses), then $\mathcal{F}$ does in fact contain Λ' and the meta-algorithm over $\mathcal{F}$ is dynamically optimal. A more subtle point is that if the optimal offline algorithm Λ' depends on the future, but is *invertible*, then the inverse of Λ' is an online algorithm from $\mathcal{F}$, when *ran backwards in time*. Since the offline optimum is invariant to reversing the access sequence, this would also lead to a dynamically optimal online algorithm.

The requirement for the offline optimum to be invertible seems too strong. (It would mean that every tree is reachable as the next state, including those trees in which soon-to-be-accessed keys are very far from the root). It remains to be seen whether a relaxed form of this property has some relevance for the dynamic optimality question.

Strong static optimality. While the simulation-technique does not seem to solve dynamic optimality, it can be used to construct an online algorithm whose cost matches the *static optimum*. (This is a special case of Iacono's result described in the beginning of the section, since we can restrict attention to static trees, as a special family of "online" algorithms.) Such an application of the MWU technique is also mentioned by Blum, Chawla, and Kalai [16].

Kalai and Vempala [49] derive a similar result using a related, but more practical technique, also resulting in an online algorithm that achieves strong static optimality. (The term "strong" refers to the fact that $\mathsf{OPT}^{\text{stat}}$ is matched with a constant factor arbitrarily close to 1, in contrast to Splay, which matches $\mathsf{OPT}^{\text{stat}}$ with a larger constant factor.)

Free rotations. This model, studied by Blum, Chawla, and Kalai [16] is similar to the unlimited online model (§ 2.2.2), with the important difference that the cost of an algorithm is only the cost of access, i.e. the length of the search path, and re-arrangements can be performed for free in arbitrary parts of the tree.

Blum et al. show that in this (arguably very strong, but still online) model there is an algorithm, again using a variant of the MWU technique, whose cost in this model (i.e. not counting the re-arrangement of the tree) asymptotically matches the offline BST optimum OPT on every sequence. It remains an interesting question whether this result can be strengthened to apply in more restricted online models. In particular, it is not clear how many rotations the algorithm of Blum et al. typically performs.

2.3.2 Other models

Randomization. In both the strict and lenient online models we required algorithms to behave deterministically. It is straightforward to extend the models such that the algorithms can use a certain number of random bits for each access. (In case of the strict model, this would mean that the function Γ can produce different re-arrangements for the same search path, depending on a parameter which we set randomly.)

It is not clear whether such an extension can significantly improve the cost of an algorithm (in expectation or with high probability). Randomized variants of Splay were studied by Fürer [46] and by Albers and Karpinski [6]. In both works the random strategies are relatively simple, e.g. deciding randomly after every access between doing the Splay re-arrangement or doing nothing, and the improvements are only by small constant factors.

Non-root access. Instead of starting every access from the root, we may allow an algorithm to start from the location of the previous access. In this model, x_{i+1} is accessed by first going up from x_i to $\mathrm{lca}(x_i, x_{i+1})$, then going down to x_{i+1}, as in a normal access. Even in a static tree, this access model leads to an interesting and non-trivial optimum, studied recently by Bose, Douïeb, Iacono, and Langerman [20]. The optimum in this model is called the *lazy finger bound*, and Bose et al. show that an approximation to this quantity can be expressed in closed form. We mention this quantity again in § 2.4.

It is easy to see that the lazy finger bound is at most twice the cost of normal root-access in the same static tree (since going to the lca is less costly than going to the root, and the cost of going to the root is paid for anyway by the access in the root-access model). More difficult to show is that the lazy finger bound is asymptotically not smaller than the usual offline BST optimum OPT. This is implied by the result of Demaine, Iacono, Langerman, and Özkan [33]. More strongly, the recent result of Iacono and Langerman [54] shows that the cost of Greedy, an online algorithm in the lenient model matches the lazy finger bound. We discuss this again in § 2.5.3 and § 2.7.

The "lazy finger" approach can also be combined with rotations, i.e. it can be defined in a dynamic model. The result of Demaine, Iacono, Langerman, and Özkan [33] implies that even algorithms in this powerful dynamic lazy finger model can be simulated with a small overhead cost by algorithms in the BST model, i.e. with root-access. Surprisingly, this holds even if we allow a multiple (constant) number of fingers (i.e. pointers) to be used. (With multiple fingers, the algorithm can decide which finger to choose when performing an access,

and the finger is left at the location of the access.) Apart from the simple lazy finger bound, the optimum quantities in these extended models are not well understood yet. For instance, it is not known whether any online BST algorithm can match the optimum performance of the best static tree with two or more fingers. We refer to the recent paper [26] for more information.

2.4 Upper bounds

In this section we review some known upper bounds on OPT from the literature. These are quantities that depend on an access sequence $X \in [n]^m$, and are asymptotically at least as large as $\mathsf{OPT}(X)$. Clearly, the cost of any $O(1)$-competitive algorithm has to be at or below these bounds (asymptotically). Thus, the main reason for defining upper bounds is to provide benchmarks for analyzing and comparing algorithms.

An obvious set of upper bounds on OPT are the costs of known algorithms (online or offline). Intuitively, if an algorithm wants to match the theoretical optimum, it must first match the costs of all other algorithms. Although simple, this observation turns out to be useful: to prove that some quantity $\mathcal{C}$ is an upper bound on OPT, it is sufficient to design an algorithm (perhaps an offline algorithm tailored to the quantity in question) whose cost is below $\mathcal{C}$. The kinds of quantities $\mathcal{C}$ we are primarily interested in are *formulaic*, i.e. can be expressed as some simple function of the access sequence.

Let $X = (x_1, \ldots, x_m) \in [n]^m$, where $m \geq n$. (For some of the bounds we require $m \geq n \log n$.) Perhaps the simplest non-trivial upper bound on $\mathsf{OPT}(X)$ is the quantity $m \cdot O(\log n)$. Since there exist static balanced trees in which every access takes $O(\log n)$ time, the cost of any $O(1)$-competitive algorithm $\mathcal{A}$ has to be $\mathrm{cost}_{\mathcal{A}}(X) = m \cdot O(\log n)$. We call this the *balance condition* of a BST algorithm.

A slightly stronger upper bound for $\mathsf{OPT}(X)$ is $O\big(\sum_{i=1}^{m} \log(x_i)\big)$. To see that this quantity is a valid upper bound, consider the BST over $[n]$ whose right spine consists of the nodes $1, 2, 4, \ldots, 2^{\lfloor \log n \rfloor}$, and where the subtrees hanging to the left of the spine are balanced binary search trees (over the elements that fall between two neighbors on the spine). This tree corresponds to a *doubling binary search*, typically used when the size of the searched list is not known in advance. It is easy to verify that the depth of node j in the constructed tree is at most $2 \cdot \lfloor \log j \rfloor + 1$, for all $j \in [n]$. Bentley and Yao [13] show that the leading constant can be reduced to 1, using a nested construction.

An even stronger upper bound on OPT is the static optimum $\mathsf{OPT}^{\mathrm{stat}}$, i.e. the cost of executing X with the best static BST (which includes the previously mentioned balanced trees). We have seen the static optimum to be asymptotically equivalent with the *entropy* of a sequence (Theorem 2.1). An algorithm $\mathcal{A}$ matches the *static optimality bound*, if for all X (sufficiently long), we have $\mathrm{cost}_{\mathcal{A}}(X) = O\big(\mathsf{OPT}^{\mathrm{stat}}(X)\big)$.

We have seen in § 2.1 a simple upper bound of a different kind. An algorithm that is $O(1)$-competitive on sequences of length n must have cost $O(n)$ when serving the sequence $S = (1, 2, \ldots, n)$. This is because Rotate-once with a particular initial tree achieves this cost, therefore $\mathsf{OPT}(S) = O(n)$. This is called the *sequential access* condition. To make this into a well-defined bound for all sequences of length n, we can define a function that equals n for the sequence S and $+\infty$ for all other sequences.

Sequential access is a special case of *traversal access*. A sequence $X \in S_n$ is a traversal sequence if it is the preorder sequence of a BST, or equivalently, if it is a 231-avoiding permutation (Lemma 1.4). Observe that $(1, 2, \ldots, n)$ is the preorder sequence of a right-leaning path. We say that an algorithm fulfills the traversal access condition, if it has cost $O(n)$ for all traversal sequences. For an arbitrary traversal sequence $X \in S_n$ we have $\mathsf{OPT}(X) = O(n)$, although

this is less obvious than for sequential access (we discuss this in § 2.5.2 and § 4.3). Traversal sequences were studied by Sleator and Tarjan [91], who asked whether Splay achieves the traversal access condition. Particularly for the case of the traversal access condition and its special cases, when we discuss online algorithms, it is important to specify the choice of initial tree.

The upper bounds discussed in the remainder of the section concern "locality of reference" in a broad sense. With the exception of the lazy finger bound, introduced recently by Bose et al. [20], they were defined by Sleator and Tarjan in the original Splay paper [91].

Let $t_X(i)$ be the *last access time* of an element x_i, i.e. $t_X(i) = \max\{j : j < i \text{ and } x_j = x_i\}$. By convention we set $t_X(i) = 0$, if x_i is accessed at time i for the first time. The *working set* at time i is defined as $w_X(i) = \{x_j : t_X(i) < j \le i\}$, i.e. the set of distinct elements accessed since the previous access of x_i. The *working set bound* for sequence X is the quantity $\sum_{i=1}^{m} \log|w_X(i)|$, and this quantity is known to be an upper bound on $\mathsf{OPT}(X)$.

Next we define bounds involving "fingers". The intuition behind these bounds is that an access sequence should be easier if its elements are "clustered together". Let $f \in [n]$ be an arbitrary fixed key. The *static finger bound* is defined as $\sum_{i=1}^{m} \log\big(|x_i - f| + 1\big)$.

Whereas static finger refers to the distance from a fixed key, the *dynamic finger bound* depends on distances between successive accesses. It is defined as $\sum_{i=2}^{m} \log\big(|x_i - x_{i-1}| + 1\big)$.

Finally, the *lazy finger bound*, mentioned in § 2.3, is defined as $\min_T \sum_{i=2}^{m} d_T(x_i, x_{i-1})$. Here T is a fixed *reference BST*, and $d_T(x, y)$ is the distance between two nodes x and y in T. The reference tree T is chosen such as to minimize the total quantity. Intuitively, the lazy finger bound is similar to the static optimum in that it captures the access cost in a static tree. However, in the case of static optimum each access starts from the root, whereas in the case of lazy finger each access starts at the location of the previous access. Despite what might seem a minor difference, the lazy finger bound turns out to be rather more powerful: with the exception of working set, it subsumes all other bounds defined in this section, including the traversal, static optimality, and static and dynamic finger bounds. (By "subsume" we mean that it is provably asymptotically smaller.) We refer to [20, 54, 26] for details.

Various other upper bounds have been defined in the literature, but those mentioned are perhaps the most important. In § 4 we describe a new class of upper bounds of a different flavor, generalizing the traversal and sequential bounds discussed above in a natural direction. Other than serving as benchmarks for BST algorithms, the purpose of the upper bounds discussed in this section is to capture structural properties that make access sequences "easy" in the BST model. The easiest sequences are those that can be accessed with constant average cost per access (i.e. with total cost $O(m)$). Several such classes of easy sequences are known, but a full characterization is lacking.

Problem 12. Characterize the sequences $X \in [n]^m$ (or at least $X \in S_m$) for which $\mathsf{OPT}(X) = O(m)$.

For most of the bounds discussed in this section it is not a priori obvious that any online algorithm should match them, but this is the case for all of them, although some of the bounds are not known to be matched in the strict online model. We discuss the known results when we introduce the algorithms to which they apply (see § 2.5.2 and § 2.5.3). Here we state two questions which we believe to be open.

Problem 13. Is there a strict online algorithm (with a fixed initial tree of its choice) that has cost $O(n)$ on every traversal sequence of length n?

Problem 14. Is there a strict online algorithm that matches the lazy finger bound?

2.5 More advanced algorithms

2.5.1 Move-to-root

We describe an online dynamic BST algorithm that is more aggressive in its re-arrangement strategy than Rotate-once. This algorithm, called Move-to-root, was introduced by Allen and Munro [7] at the same time as Rotate-once. It works as follows: after accessing an element x, it rotates the edge between x and its current parent, until x becomes the root of the tree. The algorithm is well-defined for any initial BST, and it works in the strict online model. (Again, this is easy to verify. We return to this fact in § 3.2.) We only specify the operation when accessing a key x_i, assuming that the state of the BST before the access is T.

Move-to-root(T, x_i)
1. Access(T, x_i)
2. While $\mathsf{parent}(x_i) \neq \mathsf{null}$, rotate the edge $\big(x_i, \mathsf{parent}(x_i)\big)$.

In practice, Move-to-root appears to be superior to Rotate-once. In particular, Allen and Munro show that it satisfies the static optimality condition *in expectation*, if the accesses are drawn independently at random from the frequency distribution. Nevertheless, a simple example shows that Move-to-root fails to satisfy the balance condition, in fact it can be forced to take $\Omega(n)$ time per access, thus it is not a contender for dynamic optimality. Such a bad example is illustrated in Figure 2.2. A similar example shows that Move-to-root has total cost $\Omega(n^2)$ for sequential access if the initial tree is a left-leaning path, therefore it fails to satisfy the sequential access condition.

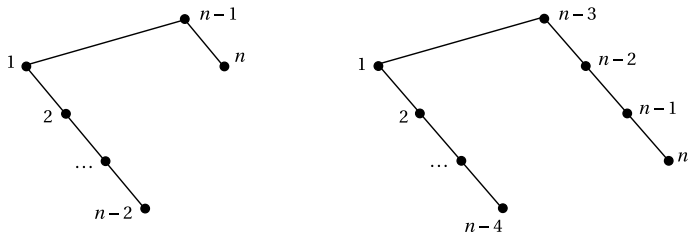

Figure 2.2: Move-to-root execution. Starting with an arbitrary initial tree, access first the sequence $(n, n-1, \ldots, 1)$, transforming the tree into a right-leaning path. Accessing $(n, n-1, \ldots, 1)$ again leaves the tree in the state of a right-leaning path. The total access cost is $\Omega(n^2)$, whereas the offline optimum is $O(n)$. (left) Tree after accessing $(n, n-1)$ in the second phase. (right) Tree after accessing $(n, n-1, n-2, n-3)$ in the second phase.

2.5.2 Splay

I see only one move ahead, but it is always the correct one.
— attributed to JOSÉ RAÚL CAPABLANCA

Splay is an algorithm introduced in 1983 by Sleator and Tarjan [91]. It resembles Move-to-root, yet its properties are far more sophisticated. Many of the intriguing questions in the dynamic BST model have first been raised in the context of Splay. The Splay algorithm conforms to the strict online model (in fact we defined this model just around Splay). This fact is easy to verify, but we return to it in § 3.2. Splay is well-defined for any initial BST. We only specify the operation when accessing a key x_i, assuming that the state of the BST before the access is T.

Splay(T, x_i)
1. Access(T,x_i)
2. While parent(x_i) $\neq$ null, repeat:
   Let p = parent(x_i) and gp = parent(p).

   2.1. If p = root(T), then rotate (x_i, p) and stop. (ZIG)

   2.2. If (x_i, p) and (p, gp) are both left-edges or both right-edges, then rotate (p, gp), then rotate (x_i, p). (ZIG-ZIG)

   2.3. Otherwise, rotate (x_i, p), then rotate (x_i, gp). (ZIG-ZAG)

Instead of simply rotating x_i to the root, Splay considers the next two edges above x_i. (This is unless x_i is one edge away from the root, in which case it simply rotates x_i to the root in step 2.1.) If the next two edges on the path from x_i to the root are of a different type, i.e. one $\diagdown$ edge and one $\diagup$ edge, then we rotate the edge between x_i and its parent twice (observe that the original grandparent of x_i becomes the parent of x_i after the first rotation) (step 2.3). If the next two edges are of the same type, then we first rotate the edge between the parent and grandparent of x_i, and only afterwards we rotate the edge between x_i and its parent (step 2.2). Step 2.2 is the only case in which something different happens compared to Move-to-root. This difference might seem insignificant, but it turns out to affect the behavior of the algorithm dramatically.

The three different cases are shown in Figure 2.3. Every rotation we perform brings x_i one step closer to the root. Therefore, the total number of rotations is exactly the length of the search path to x_i. We thus assume (as we always do in the strict model) that the rotation cost is absorbed in the cost of access.

We first state some known facts about Splay, then some long-standing open questions.

Theorem 2.11 ([91])**.** Splay (with arbitrary initial tree) matches the *balance*, *static optimality*, *static finger*, and *working set* bounds.

The following theorem was proved by Cole et al. [29, 28] in 2000, resolving the long-standing question of Sleator and Tarjan [91]. The proof is very involved.

Theorem 2.12 ([29, 28])**.** Splay (with arbitrary initial tree) matches the *dynamic finger* bound.

We sketch the proof of Theorem 2.11 later. As for Theorem 2.12, we ask the following.

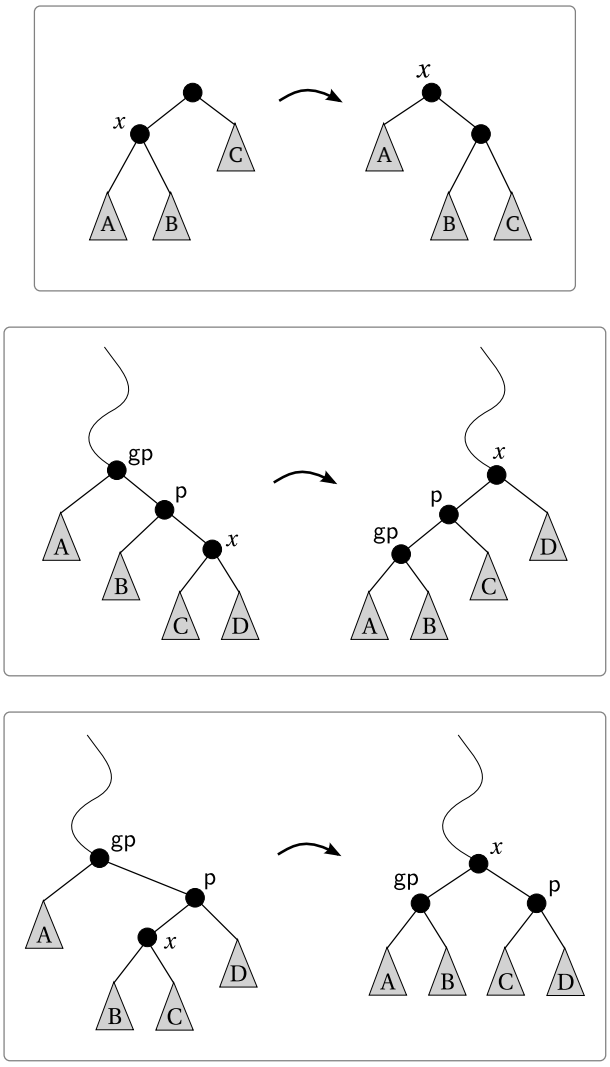

Figure 2.3: Splay local operations. (from top to bottom) ZIG, ZIG-ZIG, and ZIG-ZAG cases. x is the accessed element, A, B, C, D denote subtrees that are unaffected by the transformation. Symmetric cases omitted.

Problem 15. Is there a simpler proof that Splay matches the dynamic finger bound?

The following is a special case of Problem 14. An answer to this question would also yield a new proof for Theorem 2.12, since the lazy finger condition is known to imply the dynamic finger condition [20].

Problem 16. Does Splay match the lazy finger bound?

The following result was first shown by Tarjan [101] in 1985. Later other proofs were found that improve the constant factors in the bound.

Theorem 2.13 ([101, 100, 37, 80])**.** Splay (with arbitrary initial tree) satisfies the *sequential access* condition.

The following questions are related to Problem 13. The first is the original *traversal conjecture* of Sleator and Tarjan [91], still wide open. The second is a significantly easier, as far as we know, still open question.

Problem 17.

- Is the cost of Splay (with every initial tree) $O(n)$ for every traversal sequence of length n?
- Is the cost of Splay (with some fixed initial tree) $o(n \log n)$ for every traversal sequence of length n?

Towards proving the traversal conjecture for Splay, we are aware only of the following result of Chaudhuri and Höft [27] from 1993, in which the initial tree depends on the input sequence. Observe that this result also implies that $\mathsf{OPT}(S) = O(n)$, if S is a traversal sequence.

Theorem 2.14 ([27])**.** The cost of Splay with initial tree T (of size n) for the traversal (preorder) sequence of T as input is $O(n)$.

Finally, despite the many attractive properties of Splay, and despite its observed good performance in practice, we currently have no guarantees on its competitiveness (apart from the $O(\log n)$-competitiveness that trivially follows from the balance condition).

Problem 18. Is Splay c-competitive for some $c = o(\log n)$?

An easier question that is similarly open:

Problem 19. Is there some $c = o(\log n)$, such that the cost of Splay for X is at most $c \cdot \mathsf{OPT}^{\mathrm{str,root}}(X)$?

In the remainder of the section we sketch the proof of some of the known properties of Splay (i.e. Theorem 2.11), in order to introduce some of the tools that are used in § 3. More precisely, we state the *access lemma*, a technical condition from which the properties mentioned in Theorem 2.11 follow. We show that these properties are implied by the access lemma for every strict online algorithm with the access-to-root property. In § 3.3 we define a broad class of strict online algorithms that satisfy the access lemma. (We also show that this class includes Splay.)

Let T be a BST over $[n]$. Let $w : [n] \to \mathbb{R}_{>0}$ be a positive weight function, and for any set $S \subseteq [n]$, let $w(S) = \sum_{a \in S} w(a)$. Finally, denote $W = w(T) = w([n])$. Sleator and Tarjan [91] define the sum-of-logs potential function

$$\Phi(T) = \sum_{a \in [n]} \log\big(w\big(\mathsf{subtree}(a)\big)\big).$$

We remark that other potential functions that are of approximately the same magnitude have also been used in the literature (e.g. [88, 48, 26], [71, § 6.1.2]).

Definition 2.15 (Access lemma [91])**.** Let $\mathcal{A}$ be a strict online BST algorithm with the access-to-root property. Suppose that $\mathcal{A}$ accesses a key x in a BST T, re-arranging it into a BST T'. Let P be the search path in T to x. We say that the *access lemma* holds for $\mathcal{A}$, if there is some fixed constant $c > 0$ such that for every such access it holds that:

$$c \cdot |P| \leq \Phi(T) - \Phi(T') + O\left(1 + \log \frac{W}{w(x)}\right).$$

We state the following without proof, and refer to [91]. Alternative proofs are also given in § 3.1 and § 3.4.

Theorem 2.16 ([91])**.** The access lemma holds for Splay.

Observe that the quantity $|P|$ in the access lemma (i.e. the length of the search path) is exactly the cost of accessing x. Let $\mathcal{A}$ be a BST algorithm that satisfies the access lemma. Then, the total cost of $\mathcal{A}$ for serving X (by a telescoping sum argument) is:

$$\mathrm{cost}_{\mathcal{A}}(X) = \sum_{i=1}^{m} |P_i| = \frac{1}{c}\big(\Phi(T_m) - \Phi(T_0)\big) + O\left(m + \sum_{i=1}^{m} \log \frac{W}{w(x_i)}\right).$$

Here T_0 is the initial tree, T_i is the tree after accessing x_i, and P_i is the search path when accessing x_i in T_{i-1}.

Definition 2.15 is a slightly weaker form of the original [91], but sufficient for our purposes. The constant factor $\frac{1}{c}$ for the first term of the cost can be removed, if we change the weights $w(\cdot)$ by a factor of 2^c, since this contributes a factor of c to $\Phi(\cdot)$. Observe that this does not affect the constant factor for the other terms, hidden in the $O(\cdot)$ notation.

The proof of the following statement follows the original Splay paper (alternative proofs for some of these properties are given in [26]).

Theorem 2.17. Let $\mathcal{A}$ be a strict online algorithm with the access-to-root property for which the access lemma holds. Then $\mathcal{A}$ fulfills the balance, static optimality, static finger, and working set conditions.

Proof. Given $X = (x_1, \ldots, x_m)$, we want to bound the total cost of accessing X by $\mathcal{A}$.

Balance. Set the weights $w(a) = 1$, for all $a \in [n]$. Observing that $\Phi(T_m) \leq n \log n$ yields $\mathrm{cost}_{\mathcal{A}}(X) = m \cdot O(\log n)$.

Static optimality. Let n_i denote the number of times key i appears in X. Setting the weights $w(a) = \frac{n_a+1}{m}$ for all $a \in [n]$, and assuming that $m \geq n \log n$, we obtain that $\mathrm{cost}_{\mathcal{A}}(X)$ is less than the entropy bound (Theorem 2.1(iii)).

Static finger. For any fixed $f \in [n]$, we set the weights $w(a) = 1/\big(|a-f|+1\big)^2$. We observe that $w(T_i)$ is constant for all i (since $\sum_k 1/k^2$ converges to a constant), and obtain

$$\mathrm{cost}_{\mathcal{A}}(X) \leq O(n \log n) + O\left(m + \sum_{i=1}^{m} \log\big(|x_i - f| + 1\big)\right).$$

Working set. We set the weights $w(x_i) = 1/|w_X(x_i)|^2$, where $w_X(x_i)$ is the *working set* at time i, defined in § 2.4. Notice that the weights are now time-dependent, but at all times, the weights of the nodes are a permutation of $\{1/k^2\}$ for $k = 1, \ldots, n$. Initially we can set the weights to an arbitrary permutation of $\{1/k^2\}$. Observe that we still have that $w(T_i)$ is constant for all i. In [91] it is argued that the change of weights after an access (according to the above scheme) can only make the total potential decrease, and therefore the access lemma still holds, even with the time-dependent weights (in the case of Splay). This is because only the root node (after the access) increases its weight, and for other nodes the weight either decreases or stays the same. From the definition of the potential function Φ it is clear that the root node appears in only one term, the one containing every node – since we are only permuting the weights, the contribution of this term remains the same.

We observe that the only property of Splay necessary for this argument to go through is the access-to-root property. Thus, we conclude that the access lemma implies the working set property for any strict online algorithm with the access-to-root property, since the following holds:

$$\mathrm{cost}_{\mathcal{A}}(X) \leq O(n \log n) + O\left(m + \sum_{i=1}^{m} \log |w_X(x_i)|\right). \qquad \blacksquare$$

Semi-splay. A variant of Splay called Semi-splay has also been described by Sleator and Tarjan [91]. Here, the ZIG-ZIG case is different: only the rotation $(\mathsf{p}, \mathsf{gp})$ is performed, and the algorithm continues by moving not x, but p towards the root. Semi-splay shares many of the attractive properties of Splay, and it performs less re-arrangements than Splay. It is therefore preferable to Splay in many applications, even though it does not have the access-to-root property.

2.5.3 GreedyFuture

> Who controls the past controls the future.
> — GEORGE ORWELL, *1984*

We describe an *offline* BST algorithm called GreedyFuture, introduced by Lucas [68] in 1988 and described independently by Munro [76] in 2000. We define GreedyFuture in the second offline model. GreedyFuture satisfies the search-path-only restriction, i.e. after each access it only re-arranges the search path. GreedyFuture is an offline algorithm, in the sense that its re-arrangement of the search path depends on accesses in the future (hence the name "future"). Informally, it re-arranges the search path in such a way, that the sooner an element is accessed in the future, the closer it gets to the root (hence the name "greedy"). The initial tree in GreedyFuture can be arbitrary, although we will see that there is a *canonical* initial tree that plays an important role. Our description is loosely based on Demaine et al. [31].

We only specify the operation when accessing a key x_i, and denote by T the BST before the access. We assume that the sequence of future accesses $x_{i+1}, \ldots, x_m$ is known.

GreedyFuture(T, x_i)

1. Access(T, x_i) and let P denote the search path.
2. Let $p = (p_1, \ldots, p_{|P|})$ be the sorted keys of P, let $p_0 = -\infty$, and $p_{|P|+1} = +\infty$.
3. Let $t_j = \min\{t : t > i \text{ and } p_{j-1} < x_t < p_{j+1}\}$, for all $j = 1, \ldots, |P|$,
   or $t_j = +\infty$ if the set is empty.
4. Let $\tau = (t_1, \ldots, t_{|P|})$ and let $Q = \mathsf{treap}(p, \tau)$.
5. Replace P by Q in T.

GreedyFuture accesses a key x_i in the usual way, following the search path from the root to x_i (step 1). Then, for each key p_j in the search path we define the *next access time* t_j, which is the index of the next access that is on p_j, or in the open interval between the predecessor and successor of p_j in the search path. We set this value to $+\infty$ if there is no access in the future that falls in the interval in question (steps 2 and 3). We then replace the search path by a treap built of the search path, using the next access times as priorities (steps 4 and 5). Recall the definition of the treap that guarantees that the root is the node p_j with smallest value t_j, i.e. the one which is accessed first in the future. The same property is maintained recursively in the subtrees of the root.

We observe the following: if the next access x_{i+1} is in the current search path (say p_j), then it is guaranteed to become the root. This is because its next access time is $t_j = i + 1$, and for all other keys p_k, we have $t_k > i + 1$ (since p_j is not contained in any of the open intervals around other keys on the search path). If the next access x_{i+1} is in one of the subtrees

hanging from the search path, then we have exactly two keys p_j and p_{j+1} in the search path such that $p_j < x_{i+1} < p_{j+1}$. In this case, $t_j = t_{j+1} = i+1$, and all other next access times are greater than $i+1$. Therefore, one of p_j and p_{j+1} has to become the root, and the other one its child, according to the tie-breaking decision of the treap. (In the original description of GreedyFuture, Lucas suggests breaking the tie according to whether the left subtree of p_j or the right subtree of p_{j+1} is accessed earlier in the future.) The subtrees are built recursively in the same way. Keys on the search path that are not accessed in the future (those with next access time $+\infty$) are kept at the bottom of the constructed BST Q_i (with ties broken arbitrarily by the treap construction). Figure 2.4 illustrates the execution of GreedyFuture.

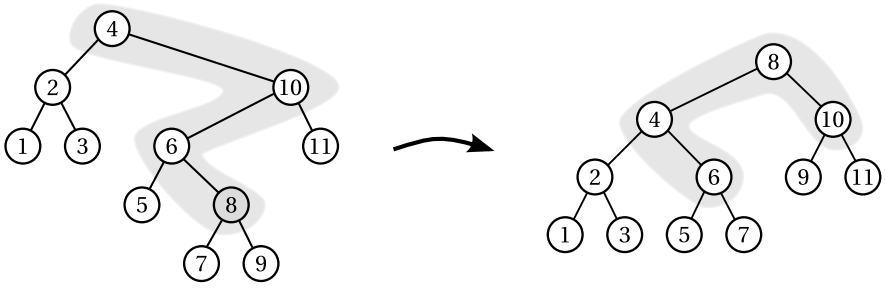

Figure 2.4: GreedyFuture execution. (left) Tree before accessing 8. (right) Tree after accessing 8, with future access sequence $(9,4,6,10,11,2,6)$. GreedyFuture replaces the search path to 8 with $\mathsf{treap}\big((4,6,8,10),(2,3,1,1)\big)$. The first sequence consists of the sorted keys in the search path, the second sequence consists of the future access times of the keys in the search path. Observe that 8 is not accessed in the future, but 9 is accessed at time 1, and it has 8 as predecessor. Similarly, although 10 is accessed only at time 4, since it is the successor of 9, it is assigned priority 1.

The initial tree T_0 of GreedyFuture can be arbitrary. Since GreedyFuture is an offline algorithm, we may as well construct T_0 depending on the input X. Perhaps the most natural choice of initial tree is the one in which keys are already ordered by their first access time. More precisely, let $t_i = \min\{t : x_t = i\}$, for all $i \in [n]$, again assuming $t_j = +\infty$ if j is never accessed. Then, let $\tau = (t_1, \ldots, t_n)$ and choose the initial tree $T_0 = \mathsf{treap}(\tau)$. The choice of this initial tree is natural, because it simulates the normal behavior of the algorithm. If at time zero GreedyFuture would "touch" every node, the resulting tree would be exactly the tree T_0 described above. We will refer to this particular choice of the initial tree as *canonical*.

GreedyFuture is of interest for two main reasons. First, because it was proposed as a theoretical construction that could plausibly achieve a cost close to the offline optimum. Second, in a surprising development, Demaine et al. [31] showed in 2009 that GreedyFuture can be simulated by an *online* algorithm with asymptotically the same cost. (As we discuss in § 2.7, this online algorithm conforms to our lenient online, but not to the strict online model.) Thus, besides Splay, GreedyFuture and its online variant discussed in § 2.7 are the most promising candidates for dynamic optimality. In the past few years it was shown that GreedyFuture matches essentially all bounds known to be matched by Splay, and also several that are not known to be matched by Splay. (To our knowledge the first such bounds were the pattern-avoiding bounds discussed in § 4. Another recent example is the lazy finger bound [54].) Furthermore, proofs for GreedyFuture (and its online variant) tend to be simpler than those for Splay, especially in the geometric model described in § 2.7. In § 5 we revisit GreedyFuture and give a number of new interpretations for it.

We mention some known results about the behavior of GreedyFuture. In 2011, Fox [42] showed that an access lemma similar to Theorem 2.15 holds for GreedyFuture, from which the results of the following theorem follow. (Observe that GreedyFuture does not have the access-to-root property. Nevertheless, the properties can be proven in the geometric view of GreedyFuture, described in § 2.7.) The balance condition was also independently shown by Goyal and Gupta [50].

Theorem 2.18 ([42])**.** GreedyFuture satisfies the *balance*, *static optimality*, *static finger*, *working set*, and *sequential access* conditions.

In 2016, Iacono and Langerman showed the following.

Theorem 2.19 ([54])**.** GreedyFuture satisfies the *lazy finger* condition.

Similarly as for Splay, we have no non-trivial results about the approximation ratio of GreedyFuture.

Problem 20. Is there some $c = o(\log n)$, such that the cost of GreedyFuture for X is at most $c \cdot \mathsf{OPT}(X)$?

Problem 21. Is there some $c = o(\log n)$, such that the cost of GreedyFuture for X is at most $c \cdot \mathsf{OPT}^{\mathrm{str,root}}(X)$? Even more specifically, can we bound the cost of GreedyFuture as c times the cost of Splay?

The performance of GreedyFuture for traversal sequences is studied in § 4 in a more general context and we present several results in this direction. Let us nevertheless state here an open question that is *not* answered by our work.

Problem 22. Is the cost of GreedyFuture $O(n)$ for every traversal sequence of length n with every initial tree?

Finally, we mention again that GreedyFuture can be simulated by an online algorithm with a constant factor overhead. We discuss this in § 2.7. It is somewhat unsatisfactory that this online algorithm does not conform to the strict model. Therefore we ask the question.

Problem 23. Is there a strict online algorithm $\mathcal{A}$, and some $c = o(\log n)$, such that the cost of $\mathcal{A}$ for every (sufficiently long) input sequence X is at most c times the cost of GreedyFuture for X? In particular, is Splay such an algorithm?

2.5.4 Other algorithms

A number of further BST algorithms and families of algorithms have been discussed in the literature, some of which fit in the strict online model. Generalizations of Splay have also been studied before (Subramanian [99], Georgakopoulos and McClurkin [48]). We postpone the discussion of these algorithms to § 3.

A class of BST algorithms of a different flavor are those related to Tango trees, achieving the currently best known competitive ratio of $O(\log\log n)$. We discuss these algorithms briefly in § 2.6.3.

2.6 Lower bounds

In this section we review known *lower bounds* for OPT. These are quantities that depend on an access sequence $X \in [n]^m$, and are asymptotically at most as large as $\mathsf{OPT}(X)$. Most of the results in this section are due to Wilber [106].

First, we ask which access sequences are hard, in the sense of requiring a total cost of $m \cdot \Theta(\log n)$. The (perhaps surprising) answer is that almost all access sequences are like this. The following statement due to Blum et al. [16, Thm. 4.1] shows this in a stronger form than the original result of Wilber [106]. The proof of Blum et al. relies on an encoding/compression argument. A similar argument by Kurt Mehlhorn [24, Thm. F.1] works also if we restrict attention to permutation sequences.

Theorem 2.20 ([16])**.** The number of access sequences having optimal offline cost (per access) k is at most 2^{12k}, for all $k \geq 0$, regardless of n or m.

Theorem 2.20 also implies that for an access sequence X drawn uniformly at random from $[n]^m$ we have $\mathsf{OPT}(X) = m \cdot \Omega(\log n)$ with high probability. This can also be shown with arguments about Wilber's lower bounds (defined later).

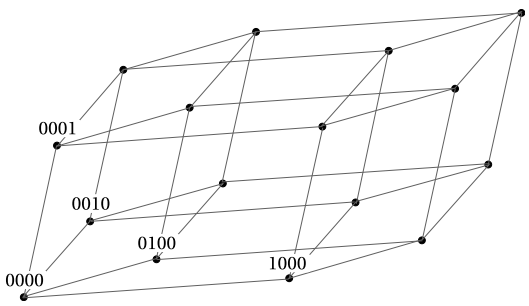

Figure 2.5: Bitwise reversal sequence $R_{16} = (0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15)$. Plot of $\big(\mathsf{rev}_4(i), i\big)$ left-to-right, bottom-to-top. Lines connect entries differing in one bit.

Wilber also showed that there exist highly structured individual sequences $X \in S_n$, for which $\mathsf{OPT}(X) = \Omega(n \log n)$. An example of such a sequence is the *bitwise reversal* sequence which we describe in more detail, as it is used in the subsequent discussion (especially in § 5).

For an integer t with $0 \leq t < 2^k$, let $\mathsf{rev}_k(t)$ be the number whose binary representation of length k (i.e. padded from the left with 0s if necessary) is the reverse of the binary representation of length k of t. For instance, $\mathsf{rev}_4(3) = 12$, since 1100 is the reverse of 0011.

Let $n = 2^k$, for some integer $k > 0$. We define R_n, the *bitwise reversal sequence* of length n as $R_n = \big(\mathsf{rev}_k(0), \mathsf{rev}_k(1), \ldots, \mathsf{rev}_k(n-1)\big)$. For instance, we have $R_8 = (0,4,2,6,1,5,3,7)$. It is instructive to look at the geometric plot of R_n, as it resembles the projection of a k-dimensional hypercube, suggesting recursive ways of constructing R_n (Figure 2.5). For values of n that are not powers of two, the sequence R_n is similarly defined (we simply truncate at the end). Wilber shows the following.

Theorem 2.21 ([106])**.** For every n we have $\mathsf{OPT}(R_n) = \Omega(n \log n)$.

We sketch the proof after we describe the first lower bound of Wilber. The two lower bounds of Wilber are efficiently computable functions of X that yield quantities asymptotically not larger than $\mathsf{OPT}(X)$. Particularly Wilber's first bound has played an important role in the development of algorithms, and the competitive ratios of Tango, Multi-splay, and Chain-splay hinge on this bound (§ 2.6.3). Wilber's second bound is in some sense cleaner, but so far it has not lead to algorithmic results, certainly not for a lack of trying.

2.6.1 Wilber's first lower bound

We describe the bound slightly differently compared to the original description [106], in a way more similar to [32] (the definitions are asymptotically equivalent). The bound has also been called *alternation* or *interleave* bound.

Let $X = (x_1, \ldots, x_m) \in [n]^m$. Wilber's first lower bound is parameterized by a *reference BST* T over $[n]$. We denote the bound as $\mathcal{W}^1_T(X)$, and define it recursively as follows. Recall that $\mathsf{root}(T)$ denotes the root of T, and let $L(T)$ and $R(T)$ denote the left, respectively right subtree of $\mathsf{root}(T)$. We define the crossing number of a sequence X with respect to a value t, denoted $\mathsf{cr}(X,t)$, as the number of neighboring pairs of entries in X that fall on different sides of t. More precisely, let

$$\mathsf{cr}(X,t) = \Big|\big\{i : x_i \le t \le x_{i+1}\big\}\Big| + \Big|\big\{i : x_i \ge t \ge x_{i+1}\big\}\Big|.$$

Wilber's first bound for X with respect to T is defined as

$$\mathcal{W}^1_T(X) = \mathsf{cr}\big(X, \mathsf{root}(T)\big) + \mathcal{W}^1_{L(T)}\big(X_{\le \mathsf{root}(T)}\big) + \mathcal{W}^1_{R(T)}\big(X_{> \mathsf{root}(T)}\big).$$

At the base of the recurrence, we let $\mathcal{W}^1_T(X) = 0$ if T is empty, or if X has fewer than two elements. Observe that for every BST T over $[n]$, we have $\mathcal{W}^1_T(X) \ge m-1$ for all $X \in [n]^m$. It is intuitive to view the lower bound geometrically (Figure 2.6). The following result holds.

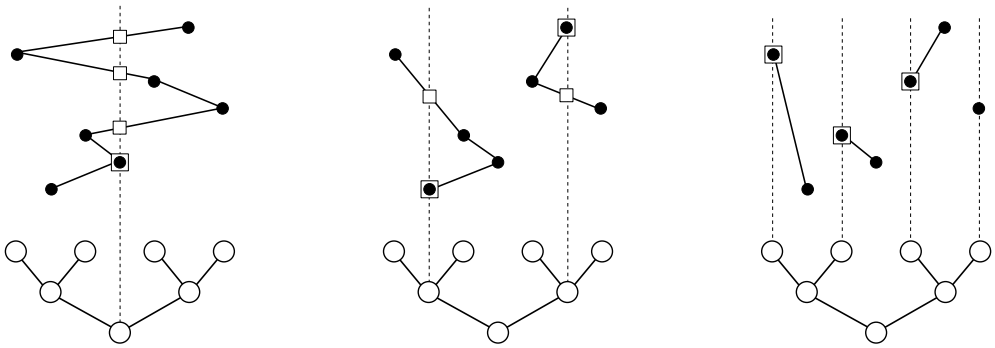

Figure 2.6: Wilber's first lower bound. The access sequence $X = (x_1, \ldots, x_m)$ is plotted as a point set $\{(x_i, i)\}$ in the plane. Here $X = (2,4,3,7,5,1,6)$. Below we show the reference tree T (a balanced tree of depth 2). Squares indicate the crossings. From left to right we show the crossings corresponding to the first, second, resp. third levels of the reference tree. The quantity $\mathcal{W}^1_T(X)$ counts the total number of crossings on all levels.

Theorem 2.22 ([106])**.** For all $X \in [n]^m$, such that $m \ge n$, and for all BSTs T over $[n]$, we have $\mathsf{OPT}(X) = \Omega\big(\mathcal{W}^1_T(X)\big)$.

The dependence on the reference tree makes this lower bound slightly inconvenient. It is not clear which reference tree is the best for given X, in the sense of maximizing $\mathcal{W}^1_T(X)$, although such a tree is certainly computable with a simple dynamic program. In the original definition, a balanced tree over $[n]$ is used, and this is the choice used in Tango as well (§ 2.6.3). Demaine et al. [32] argue that for any *fixed* tree T over $[n]$ there is some sequence $X \in [n]^m$, such that $\mathcal{W}^1_T(X) = O\big(\mathsf{OPT}(X)/\log\log n\big)$. To see this, consider an access sequence of length m that consists of elements randomly sampled from a path of T of length $\log n$. By Theorem 2.20, for such a sequence we have $\mathsf{OPT}(X) = m \cdot \Theta(\log\log n)$. On the other hand, it is not hard to show that in this case $\mathcal{W}^1_T(X) = O(m)$.

We know that for a balanced BST T, the gap of $\Theta(\log\log n)$ between $\mathcal{W}^1_T$ and OPT is the maximum possible. This is because the cost of Tango is above OPT, but it is at most a $\Theta(\log\log n)$ factor greater than $\mathcal{W}^1_T$. For particularly bad choices of T, the gap between $\mathcal{W}^1_T$ and OPT can be as large as $\Theta(\log n)$.

It remains open, whether Wilber's first lower bound is tight with a reference tree suitably chosen for the access sequence X. Even with a balanced reference tree, for the above separation it seems essential to query elements repeatedly. Let us denote $\mathcal{W}^1(X) = \max_T \big(\mathcal{W}^1_T(X)\big)$, and state the following open questions.

Problem 24. Prove or disprove that $\mathcal{W}^1(X) = \Theta\big(\mathsf{OPT}(X)\big)$ for all $X \in [n]^m$.

Problem 25. Let T be a balanced BST over $[n]$. Prove or disprove that $\mathcal{W}^1_T(X) = \Theta\big(\mathsf{OPT}(X)\big)$ for all $X \in S_n$.

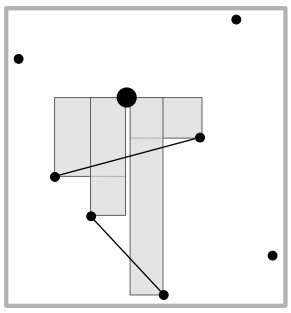

Figure 2.7: Wilber's second lower bound. Example sequence $X = (5,8,3,2,6,4,1,7)$ plotted geometrically. An axis-parallel rectangle with corners $x, y \in X$ is *empty*, if it has nonzero area, and it contains no points of $X \setminus \{x, y\}$. The *funnel* of x_i is the set of elements x_j with $j < i$, such that x_i and x_j form an empty rectangle. In the figure, $x_6 = 4$ is highlighted, and its funnel shown with rectangles. Sweeping the funnel with a horizontal line bottom to top, n_i counts the number of times we switch between the two sides (left and right) of x_i. In the figure the switches are shown with lines, and $n_6 = 2$. Wilber's second lower bound is $\sum_i (n_i + 1)$.

It is now easy to prove Theorem 2.21. For a balanced BST T over $[n]$, Wilber's first lower bound for R_n is easily seen to be $\mathcal{W}^1_T(R_n) = \Theta(n \log n)$. The tree can even be constructed in such a way that R_n produces every possible crossing at every level.

2.6.2 Wilber's second lower bound

The second lower bound of Wilber [106] does not require a reference tree (informally, the role of the reference tree is played by the Move-to-root algorithm). The bound has also been called *funnel* bound. The definition below is based on [53].

Given $X = (x_1, \ldots, x_m) \in [n]^m$, Wilber's second lower bound for X, denoted $\mathcal{W}^2(X)$, is defined as follows. Let T_0 be the right-leaning path over $[n]$, and execute the Move-to-root algorithm (§ 2.5.1) on X, starting with T_0. Let P_i be the search path for searching x_i, for each $i \in [m]$. Let n_i denote the number of *zigzags* on the path P_i, i.e. the number of successive pairs of edges in P_i consisting of a $\diagdown$ and a $\diagup$ edge. In other words, n_i counts the number of successive pairs of elements on the search path that are on different sides of x_i (one smaller, one greater). We define $\mathcal{W}^2(X) = \sum_{i=1}^m n_i + m$.

There is an alternative, geometric definition of Wilber's second lower bound, shown in Figure 2.7. We omit here the proof that the two definitions are equivalent, and discuss this in § 2.7.1.

Theorem 2.23 ([106])**.** For all $X \in [n]^m$, such that $m \geq n$, we have $\mathsf{OPT}(X) = \Omega\big(\mathcal{W}^2(X)\big)$.

Nothing seems to be known about the possible gap between Wilber's second bound and OPT or the relation between the two lower bounds of Wilber.

Problem 26. Prove or disprove that $\mathcal{W}^2(X) = \Theta\big(\mathsf{OPT}(X)\big)$ for all $X \in [n]^m$.

Problem 27. Prove or disprove that $\mathcal{W}^1(X) = \Theta\big(\mathcal{W}^2(X)\big)$ for all $X \in [n]^m$.

2.6.3 Algorithms based on Wilber's first lower bound

We briefly mention three online algorithms that are based on a similar idea, and that all achieve $O(\log\log n)$-competitiveness. All three algorithms rely on Wilber's first bound, discussed in § 2.6.1. The algorithms mentioned in this section achieve a cost at most an $O(\log\log n)$ factor greater than the lower bound, consequently their approximation of OPT is not worse than this factor. (As a second consequence, they show that the gap between Wilber's first lower bound and OPT is not larger than this factor.)

The detailed description and analysis of these algorithms is out of scope here. We mention only that their analysis works (informally) by *charging* the cost of certain operations, i.e. of rotations and of following pointers to Wilber's lower bound $\mathcal{W}^1_T$, where T is a balanced tree over $[n]$. Operations charged to the lower bound can be considered "free". For the algorithms we mention, it can be shown that on average for every roughly $\log\log n$ operations, one operation is "free" in the above sense, yielding the overall competitive ratio. The algorithms work in the lenient model, and they heavily rely on annotations, as well as on re-arrangements outside the search path. The competitive ratio of $O(\log\log n)$ seems to be inherent to the approach used in this class of algorithms, therefore it is not clear whether this line of study will lead to further improvements. It is an interesting question whether the techniques used by these algorithms can be simulated in the strict model, although this seems rather difficult as well.

Tango was introduced by Demaine, Harmon, Iacono, and Pătraşcu [32] in 2004. It is compatible with the lenient online model, as it stores $O(\log\log n)$ bits of annotation per node, no persistent state between accesses (except for the annotations), and uses a modest amount of computation. (In fact, we defined the lenient model partly with Tango in mind.) It maintains a *preferred path decomposition* of a balanced reference tree, and each preferred path is maintained internally as a red-black tree. Besides the balance-bits required by the red-black trees, further bookkeeping information of at most $O(\log\log n)$ bits per node is stored. When the preferred path decomposition changes, *split* and *merge* operations are employed. Since every path in the preferred path decomposition is of length $O(\log n)$, once we reach the path in which the searched element resides, we need to spend only $O(\log\log n)$ time to locate it. (Since each path is internally stored as a balanced BST.) The cost of locating the correct path is charged to Wilber's lower bound, as mentioned above. (This works, because switching from one path to another when searching for a given element corresponds exactly to a crossing in Wilber's first bound.)

Both Chain-splay, introduced by Georgakopoulos [47] in 2005, and Multi-splay, introduced independently by Wang, Derryberry, and Sleator [105] in 2006 are based on the same idea as Tango, except that they use Splay trees at multiple levels, instead of the red-black tree components. As such, they are conceptually more uniform, and (particularly Multi-Splay) fulfill properties that Tango does not. (For instance, in its original form, Tango does not satisfy the balance condition, whereas Multi-splay does.) The factor of $\Theta(\log\log n)$ is "hard-coded" into Tango – for the other two algorithms a better competitive ratio can be conjectured (perhaps with a similar or lower confidence as for Splay itself). Nonetheless, the proven competitive ratio of both Chain-splay and Multi-splay is $O(\log\log n)$, similarly to Tango.

2.7 The geometric view

In this section we describe briefly the geometric BST model of Demaine et al. [31, 52]. As the model concerns only the relative positions of points and not their distances, the term "geometric" should be understood only as "visual". All arguments in this geometric view can readily be translated back to the original, geometry-free setting.

Again, consider an access sequence $X = (x_1, \ldots, x_m) \in [n]^m$. We view X at the same time as a collection of points in the plane in a straightforward way: $X = \{(x_i, i) : 1 \le i \le m\} \subset [n] \times [m]$. (By $[n] \times [m]$ we denote the integer grid with n columns and m rows.) We refer to the x-coordinate and the y-coordinate of a point as its *key value* and its *time*, respectively.

We call a point set $Y \subseteq [n] \times [m]$ *satisfied* (the term used by Demaine et al. is *arborally satisfied*), if for every pair of points $a, b \in Y$, one of the following holds: (i) a and b are on the same horizontal or vertical line, or (ii) the unique axis-parallel rectangle with corners a and b contains some point in $Y \setminus \{a, b\}$, possibly on the boundary of the rectangle. We call a pair a, b of points that violate both conditions (i) and (ii) an *unsatisfied pair*.

The *minimum satisfied superset* problem asks, given a point set X, to find a satisfied superset $Y \supseteq X$ of smallest cardinality. It is easy to see that if $X \subseteq [n] \times [m]$, then a minimum satisfied superset of X can also be assumed to contain points only from $[n] \times [m]$, therefore, in the following we assume all satisfied supersets to be of this kind (i.e. not to contain fractional, or out-of-bounds points). Let us call this first variant of the problem *offline* for reasons that will become clear soon.

Minimum satisfied superset problem **(Offline version)**

1. Read point set $X \subset [n] \times [m]$.
2. Output satisfied point set $Y \supseteq X$, such that $Y \subseteq [n] \times [m]$.

Next, we define the *execution trace* (or simply *execution*) of a BST algorithm. Let $\mathcal{A}$ be an offline BST algorithm in the second model. Let $Q_1, \ldots, Q_m$ be the sequence of BSTs produced by $\mathcal{A}$ while serving X. Recall that $Q_1, \ldots, Q_m$ contain the nodes that are "touched" when accessing $x_1, \ldots, x_m$, and the cost of $\mathcal{A}$ on X is $\mathrm{cost}_{\mathcal{A}}(X) = \sum_i |Q_i|$.

The execution trace of $\mathcal{A}$ when serving X is defined as the point set $\{(x, i) : x \in Q_i\} \subseteq [n] \times [m]$. In words, the execution trace of $\mathcal{A}$ contains a point (x, i), exactly if $\mathcal{A}$ "touches" x at time i (when accessing x_i). The following theorem captures the *offline equivalence* between the satisfied superset problem and the BST problem.

Theorem 2.24 ([31])**.** A point set Y is an execution trace of some offline BST algorithm (with some initial tree T_0) serving X exactly if Y is a satisfied superset of X.

Theorem 2.24 implies that the cardinality of a satisfied superset of X equals the cost of *some* offline algorithm serving X, it is therefore an upper bound on $\mathsf{OPT}(X)$. Thus, the geometric view can be used for designing offline algorithms for the BST problem: given an access sequence X, it is sufficient to solve the (perhaps) cleaner geometric problem of finding a satisfied superset of X of small cardinality. We illustrate the geometric view of BST in Figure 2.8.

For the proof of Theorem 2.24 we refer to [31]. The fact that the execution trace of a BST algorithm is a satisfied superset of the input is easy to show. For the other direction, to "simulate" a satisfied superset Y by an offline algorithm in the second model, one needs to construct the trees T_0 and Q_i, for all i, in such a way that an access at a future time j will not have to "touch" any nodes other than those indicated by Y. This can be achieved by using treaps with priorities given by "future touch times", similarly to the algorithm GreedyFuture

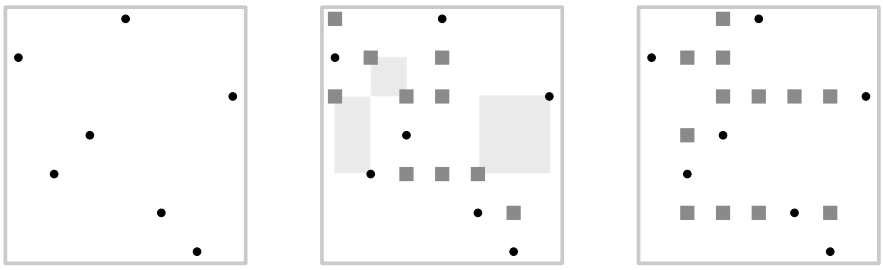

Figure 2.8: Geometric view of BST. (left) Access sequence $X = (6,5,2,3,7,1,4)$, with key values on x axis, and time on the y axis bottom-to-top, (middle) A superset of X that is not satisfied (newly added points are shown as squares, and some unsatisfied pairs are highlighted), (right) A satisfied superset of X corresponding to a BST algorithm serving X.

(§ 2.5.3). The proof of Theorem 2.24 is constructive: given a satisfied superset of X, we "recover" an offline BST execution, whose "touched" nodes are exactly those given by Y.

It may seem that this dependence on the future is essential in the correspondence between the tree-view and geometric view of BST. Surprisingly, Demaine et al. show that the correspondence between the tree-view and geometric view can also be maintained in an online fashion, i.e. without knowledge of the future. The following definition and theorems capture this result.

Recall that the satisfied superset problem asks to find a satisfied superset of a given set of points $X \subset [n] \times [m]$. Consider now the following problem, in which the input and output are processed line-by-line, with no possibility of going back.

Minimum satisfied superset problem **(Online version)**

For each $i = 1, \ldots, m$:

1. Read point $(x_i, i) \in [n] \times [m]$.
2. Output set of points Y_i with y-coordinates equal to i, such that $Y = Y_1 \cup \cdots \cup Y_m$ is a satisfied superset of $X = \{(x_i, i) : 1 \le i \le m\}$.

The cost of the solution for the online satisfied superset problem is, as in the offline case, the cardinality of the full output Y.

Theorem 2.25 ([31])**.** Let $\mathcal{A}$ be an online BST algorithm (in any model) with arbitrary initial tree, and let Y be the execution trace of $\mathcal{A}$ when accessing X. Let Y_i be the set of points in Y with y-coordinate equal to i. Then, $Y_1, \ldots, Y_m$ is a valid output for the online satisfied superset problem with input X.

Theorem 2.25 follows immediately from Theorem 2.24, since the restriction for $\mathcal{A}$ to be an online algorithm only makes the statement more specific. For any online algorithm (in any model) there is a corresponding offline algorithm, which "happens to do the same thing", and the execution trace of this algorithm revealed line-by-line is a valid solution to the online satisfied superset problem. The following result, however, is far from obvious.

Theorem 2.26 ([31])**.** Let $\mathcal{A}$ be an algorithm for the online satisfied superset problem, whose total cost for input X is $\mathrm{cost}_{\mathcal{A}}(X)$. Then, there is an online BST algorithm with some initial tree T_0 (independent of X), whose cost for input X is $O\big(\mathrm{cost}_{\mathcal{A}}(X)\big)$.

The proof of Theorem 2.26 from [31] relies on the non-trivial *split-tree* data structure. It essentially simulates the future-dependent treap construction of Theorem 2.24 in a lazy

(non-future-dependent) way. We can interpret Theorem 2.26 as saying that to solve the online BST problem, it is sufficient to design a (geometric) algorithm for the online satisfied superset problem, and to simulate it by an online BST algorithm. Two caveats of this result are the constant factor slowdown, and the fact that the resulting online algorithm requires annotations, and may access elements outside the search path (due to the technical details of the split-tree). It conforms therefore not to the strict, but at best to the lenient online model. We say "at best" because we have not made any assumptions on the computational model in which the online satisfied superset algorithm lives. In case this algorithm relies on "exotic" operations of any kind, the tree-view algorithm simulating it will also need to execute similar kinds of operations. This issue, however, is not encountered in any of the algorithms we discuss.

2.7.1 Algorithms in geometric view

From the previous discussion we conclude that it is worthwile to try designing algorithms for the geometric satisfied superset problem, whether in the online or in the offline variant (i.e. whether we require the input to be revealed, and the solution to be constructed, *line-by-line* or not).

Let us look first at existing BST algorithms from tree-view, and see what they do in geometric view. Figure 2.9 shows the execution traces of various algorithms on the same access sequence. The reader can easily verify that the resulting point sets are satisfied (no unsatisfied pairs).

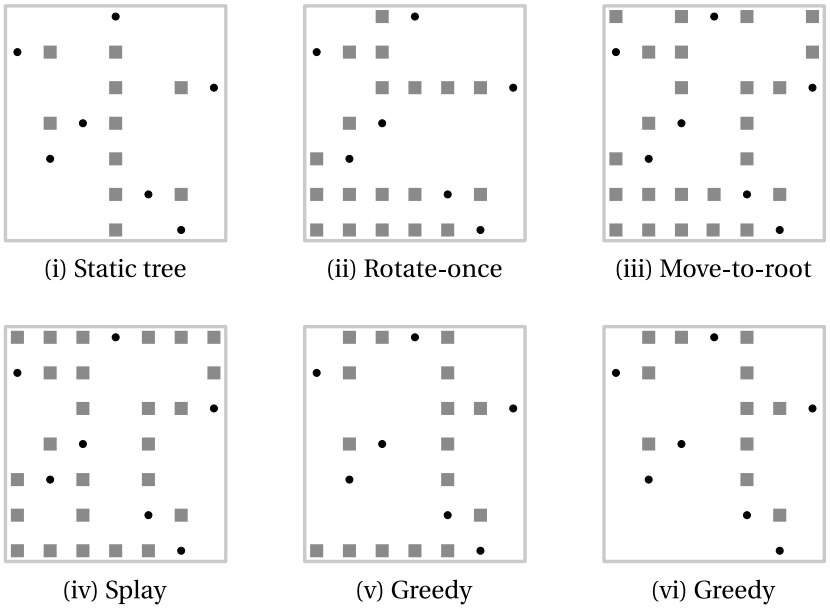

Figure 2.9: BST algorithms in geometric view. Execution traces for access sequence $X = (6,5,2,3,7,1,4)$. Accesses shown as dots, touched nodes shown as squares. (i) Static balanced BST, (ii) Rotate-once with right-leaning path initial tree, (iii) Move-to-root with right-leaning path initial tree, (iv) Splay with right-leaning path initial tree, (v) GreedyFuture with right-leaning path initial tree, (vi) GreedyFuture with canonical initial tree.

Figure 2.9(i) shows the execution of the access sequence $X = (6,5,2,3,7,1,4)$ in a static balanced BST having node 4 at the root, nodes 2 and 6 as the children of the root, and nodes $1,3,5,7$ as leaves. The interpretation of the execution trace is straightforward. Since the root node 4 is on the search path of every accessed key, it is touched every time (in every row).

Node 2, the left child of the root is touched whenever an element smaller than 4 is accessed, namely in the cases $x_3 = 2$, $x_4 = 3$, and $x_6 = 1$. The fact that the execution trace of a static BST is a satisfied point set follows from Theorem 2.24. It is, however, instructive to prove this through a direct (easy) argument for static trees.

We briefly discuss the geometric view of Move-to-root, illustrated in Figure 2.9(iii). Consider two points $(x_i, i), (x_j, j) \in X$, assuming $i > j$ and $x_i > x_j$. Suppose there is no other access point "in between", i.e. there is no $(x_k, k) \in X$ with $j < k < i$, and $x_j \le x_k \le x_i$. Then, we claim that at time i, node x_j is on the search path of x_i, it is therefore touched by Move-to-root. This is because, at time j, node x_j becomes the root (and as such, the ancestor of x_i). It is easy to see that in Move-to-root, only the accessed node can gain new descendants. Since no other node between x_j and x_i is accessed between time j and i, it is impossible for some node other than x_j to become $\mathrm{lca}(x_i, x_j)$, therefore x_j is still the ancestor of x_i at time i. In geometric view, this means that if two *input points* $(x_i, i), (x_j, j) \in X$, with $i > j$, form a rectangle containing no other points of X, then the output Y will contain the point (x_j, i).

The previous observation resembles the earlier definition of unsatisfied pairs, except that here we refer only to points of X, whereas an unsatisfied pair may consist of arbitrary points in $Y \supseteq X$. This difference hints at why Move-to-root may be inefficient. The pair $(x_i, i), (x_j, j)$ may already be satisfied at time i by some point in $Y \setminus X$. In this case, it may be redundant to add point (x_j, i) to the solution, but Move-to-root adds it anyway.

It can be shown that points of the above kind (i.e. corners of rectangles defined by input points) are essentially the *only* points in the output of the geometric Move-to-root, apart from the input points themselves, and apart from some points added due to the initial tree. Again, the fact that the execution trace of Move-to-root is a satisfied point set follows from Theorem 2.24, but it can also be shown through a simpler direct argument. Based on the above equivalence between the tree-view and geometric view of Move-to-root, it can be verified easily why the two earlier definitions of Wilber's second lower bound are equivalent (see § 2.6.2 and Figure 2.7). We omit the details.

Let us now formulate a necessary and sufficient requirement for algorithms solving the *online* satisfied superset problem. Consider the execution of an algorithm $\mathcal{A}$ at time i. At this time, the point sets $Y_1, \ldots, Y_{i-1}$ have already been output (and cannot be changed), input point (x_i, i) has been read, and the algorithm is about to construct the output point set Y_i, i.e. the ith line of the overall output Y. Observe that if $\mathcal{A}$ is a correct algorithm, then the set $Y_1 \cup \cdots \cup Y_{i-1}$ must be satisfied. Suppose otherwise, that $Y_1 \cup \cdots \cup Y_{i-1}$ contains an unsatisfied pair a, b. Since all future outputs $Y_i, \ldots, Y_m$ are disjoint from the rectangle formed by a and b, the overall output Y will remain unsatisfied, contradicting the correctness of $\mathcal{A}$.

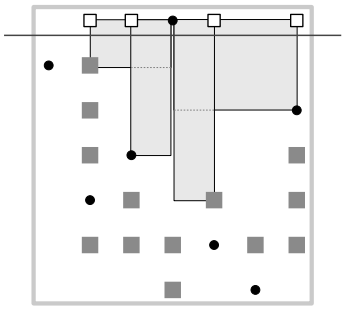

Figure 2.10: Stair in geometric view. Horizontal line shows current time i. Points below line form a satisfied set. Rectangles indicate unsatisfied pairs with the current access point x_i, and hollow squares indicate $\mathsf{stair}(x_i)$.

Let us define the *stair of x_i at time i*, denoted $\mathsf{stair}_i(x_i)$ or simply $\mathsf{stair}(x_i)$ as the set of x-coordinates of points in $Y_1 \cup \cdots \cup Y_{i-1}$ that form unsatisfied pairs with (x_i, i) (with respect to the already constructed set of points $Y_1 \cup \cdots \cup Y_{i-1}$). The following lemma claims that at every time, the points corresponding to the stair of the current access point have to be added, at a minimum, to the solution. This statement resembles (not accidentally) the observation that the search path to the accessed element x_i has to be touched by every BST algorithm. We illustrate the concept of "stair" in Figure 2.10.

Lemma 2.27 ([31])**.** Let $\mathcal{A}$ be an algorithm for online satisfied superset. Then, the output Y_i of $\mathcal{A}$ at time i must contain all points (y,i), where $y \in \mathsf{stair}_i(x_i)$. Furthermore, the set containing only these points in addition to the access point (x_i,i) is a valid output.

Proof. Suppose that $Y_1 \cup \cdots \cup Y_i$ is satisfied and for some $y \in \mathsf{stair}_i(x_i)$, we have $(y,i) \notin Y_i$. By the definition of the stair, in $Y_1 \cup \cdots \cup Y_{i-1} \cup \{(x_i,i)\}$ there is an unsatisfied pair (x_i,i), (y,j) for some $j < i$. Since this pair is satisfied in $Y_1 \cup \cdots \cup Y_i$, there must be a point $(q,i) \in Y_i$ such that (q,i) is in the rectangle formed by (x_i,i) and (y,j). Assume w.l.o.g. that $y < x_i$ and that (q,i) is the leftmost such point. By our assumption, $q > y$ must hold, but then the pair (q,i), (y,j) is unsatisfied, a contradiction. Therefore, all points (y,i), where $y \in \mathsf{stair}_i(x_i)$ must be contained in Y_i.

For the second part, we need to show that $Y_i = \{(x_i,i)\} \cup \{(y,i) : y \in \mathsf{stair}_i(x_i)\}$ is a valid output. Suppose for contradiction that $Y_1 \cup \cdots \cup Y_i$ contains an unsatisfied pair (y,i), (q,j), for some $j < i$. (We assume that i is the first time this happens.) Assume w.l.o.g. that $q < y$. Observe that (y,i) cannot be the same as (x_i,i), for in that case (q,i) would have been added to the solution, making the pair (y,i), (q,j) satisfied. There are three cases to consider: (1) If $x_i < q$, then, by the fact that (y,i) was added to the solution, there must exist some point $(y,k) \in Y_1 \cup \cdots \cup Y_{i-1}$, such that $j < k < i$, contradicting that the pair (q,j), (y,i) is unsatisfied. (2) If $q \le x_i < y$, then (x_i,i) itself is in the rectangle formed by (q,j) and (y,i), making the pair satisfied. (3) Finally, if $x_i > y$, then, again, there must exist some point $(y,k) \in Y_1 \cup \cdots \cup Y_{i-1}$, such that $k < i$. If $k \ge j$, then (q,j), (y,i) is satisfied, if $k < j$, then (q,j) and (x_i,i) must have formed an unsatisfied pair, therefore $(q,i) \in Y_i$, a contradiction. ∎

Lemma 2.27 gives a minimum necessary set of points that every valid online satisfied superset algorithm must output at a given time. This minimum set of points is also sufficient. It is natural to propose the algorithm whose output is (at every time) just this minimum, and nothing else. Let us call the resulting algorithm GeometricGreedy, defined as follows. (Demaine et al. [31] introduced this algorithm with the name GreedyASS.)

GeometricGreedy
For each $i = 1, \ldots, m$:
1. Read point $(x_i,i) \in [n] \times [m]$.
2. Output $Y_i = \{(x_i,i)\} \cup \{(y,i) : y \in \mathsf{stair}_i(x_i)\}$.

GeometricGreedy appears to be a very reasonable geometric sweepline algorithm for the minimum satisfied superset problem: processing the input one row at a time, it adds the minimum number of points necessary to make the point set on one side of the sweepline satisfied. Observe that GeometricGreedy solves the *online* satisfied superset problem. Remarkably, this algorithm turns out to be an old friend in disguise.

Theorem 2.28 ([31])**.** For every X, the output of GeometricGreedy for X is the execution trace of GreedyFuture for X with the canonical initial tree.

Proof. We refer to § 2.5.3 for the description of GreedyFuture. We run simultaneously GreedyFuture in tree-view (starting from the canonical tree) and GeometricGreedy in geometric view on the same input X. Suppose by induction that until time i (before accessing x_i) the execution trace of GreedyFuture equals the output of GeometricGreedy. Since at time i GreedyFuture only touches the search path of x_i, and GeometricGreedy only adds the points with x-coordinate in $\mathsf{stair}_i(x_i) \cup \{x_i\}$, we only need to show that these two sets are equal.

Let $y \in [n]$ be an arbitrary key. Suppose that $y \in \mathsf{stair}_i(x_i)$, and let j be the *last touched time* of y. More precisely, j is the largest integer such that $j < i$ and $(y,j) \in Y_j$. (If no such

integer exists, then y could not be in $\mathsf{stair}_i(x_i)$.) Assume w.l.o.g. that $y < x_i$. Observe that after accessing x_j at time j in the execution of GreedyFuture, node y becomes the ancestor of x_i. To see this, observe that it cannot be the case that $y < \mathrm{lca}(y, x_i) \le x_i$, since no element from this interval is in the search path of x_j (in Q_j), since the rectangle between (y, j) and (x_i, i) is empty in the GeometricGreedy execution. Therefore, at time j, it holds that $y = \mathrm{lca}(y, x_i)$. Since no key in the interval $[y, x_i]$ is on any of the search paths for $x_{j+1}, \ldots, x_{i-1}$, node y remains an ancestor of y, it is therefore on the search path of x_i at time i.

For the converse, suppose that $y \notin \mathsf{stair}_i(x_i) \cup \{x_i\}$. We need to argue that at time i, node y is not an ancestor of x_i in the GreedyFuture execution. Again, assume w.l.o.g. that $y < x_i$, and let j be the maximum integer such that $j < i$ and $y \in Y_j$.

Suppose first that no such integer j exists, i.e. y was never touched. Then, in the treap construction of the canonical initial tree the priority of x_i is smaller than the priority of y (since x_i is accessed earlier than y). In the canonical tree, y cannot thus be the ancestor of x_i. Since y is not touched before time i, i.e. it is not on any of the search paths in GreedyFuture, it cannot gain new descendants, therefore, it is not an ancestor of x_i at time i. We assume therefore that the last touched time j of y exists ($0 < j < i$).

Since $y \notin \mathsf{stair}_i(x_i) \cup \{x_i\}$, there is some point $(q, k) \in Y_k$, such that $j \le k < i$ and $y < q \le x_i$. Let (q, k) be such a point with maximum value of k, and among those with the same value of k, the one with maximum value of q. Observe that $q \in \mathsf{stair}_i(x_i) \cup \{x_i\}$, and by a similar argument as before, after the access at time k we have that q is an ancestor of x_i (possibly q equals x_i).

Suppose for contradiction that y is an ancestor of x_i at time i. Since y is not touched (i.e. not on a search path) when accessing $x_{k+1}, \ldots, x_{i-1}$, it must hold that y is an ancestor of x_i already after the access at time k. Since, at this time q is the ancestor of x_i, by the ordering condition it must hold that y is the ancestor of q. Thus, y is touched at time k, therefore $k = j$. Recall the treap construction in GreedyFuture at time k. Since x_i is in the subtree of q, the priority of q is at most i. For y to become the ancestor of q, it must have a priority less than i, in other words, there must exist some x_t in the left subtree of y, such that $j < t < i$. Let x_t be such an element with minimum t. Then, when x_t is accessed, y is on its search path, therefore $(y, t) \in Y_t$ must hold. This contradicts our choice of j as the last touched time of y. ∎

The remarkable aspect of Theorem 2.28 is that GeometricGreedy, an (online) algorithm for satisfied superset turns out to be the geometric counterpart of the (offline) BST algorithm GreedyFuture (§ 2.5.3). By Theorem 2.26, we obtain an *online* BST algorithm that simulates GreedyFuture with a constant factor overhead. Let us call the resulting online BST algorithm OnlineGreedy. When there is no chance for confusion, we will refer to both GeometricGreedy and OnlineGreedy simply as Greedy. We refer to Figure 2.9(vi) for an example run of GeometricGreedy (or equivalently, the execution trace of GreedyFuture with canonical initial tree).

It is natural to ask what the execution trace of GreedyFuture looks like, if we have an initial tree other than the canonical one. The equivalence between stairs and search paths from the proof of Theorem 2.28 suggests a method of inserting an arbitrary initial tree in geometric view.

For an arbitrary BST T over $[n]$, consider the point set $\mathcal{P}_T = \mathcal{P}_1 \cup \cdots \cup \mathcal{P}_n$, where $\mathcal{P}_i = \{(i, -\mathrm{depth}_T(i)), (i, -\mathrm{depth}_T(i) - 1), \ldots, (i, -n + 1)\}$. In words, $\mathcal{P}_T$ consists of columns of points stacked for every key in $[n]$, equal in height to n minus the depth of the key in T. These columns are placed under X, i.e. with y-coordinates at most 0.

For an arbitrary access sequence $X \in [n]^m$ (and corresponding point set), and an arbitrary BST T over $[n]$, we define the point set $X_T = \mathcal{P}_T \cup X$. We then run GeometricGreedy on X_T

(with y-coordinates of the input and output now running from $-n+1$ to m). Observe that the rows of X_T corresponding to the initial tree T are already satisfied, therefore GeometricGreedy does not add any new points in these rows. When we refer to the output of GeometricGreedy for X with initial tree T, we mean the output of GeometricGreedy on X_T, excluding the points in $\mathcal{P}_T$ from the output. The following theorem captures the correspondence between GreedyFuture and GeometricGreedy with initial trees.

Theorem 2.29 ([31])**.** For every sequence $X \in [n]^m$ and BST T over $[n]$, the output of GeometricGreedy for X with initial tree T is the execution trace of GreedyFuture with the initial tree T.

The proof of this result requires only a small modification of the proof of Theorem 2.28. Namely, we observe that before the first access x_i, the equivalence between stairs and search paths already holds. That is, for every element $y \in [n]$, we have that the search path of y in T consists exactly of the keys in $\mathsf{stair}_1(y)$. Note that the unsatisfied rectangles that define $\mathsf{stair}_1(y)$ contain one corner from the initial tree. The usage of initial trees in geometric view is illustrated in Figure 2.11, and the execution trace of GreedyFuture with initial tree is illustrated in Figure 2.9(v).

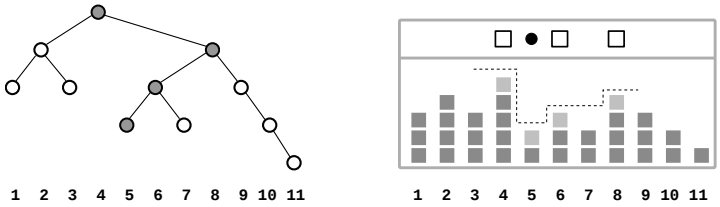

Figure 2.11: Tree-view and geometric view of initial tree. Search path and stair (at time 1) of key 5.

In summary, GeometricGreedy can exactly capture the execution trace of GreedyFuture with arbitrary initial tree. Most often we look at GeometricGreedy with "no initial tree", which corresponds to GreedyFuture with canonical initial tree. In the online BST simulation of GeometricGreedy, the "no initial tree" state corresponds to the initial split-tree data structure. The important difference between the two is that the canonical initial tree of GreedyFuture depends on the future, whereas the split-tree initial tree of OnlineGreedy does not. Thus, the statements we prove about GeometricGreedy "with no initial tree" can be translated to statements about OnlineGreedy started from a particular fixed initial tree. Statements we prove about GeometricGreedy started "from initial tree T" can be translated to statements about OnlineGreedy started from initial tree T. One can also view the "no initial tree" state of GeometricGreedy as a "preprocessing". Namely, we can think of the OnlineGreedy algorithm as if it starts by rotating its initial tree (whatever it may be) to the split-tree data structure (in linear time). It is worth mentioning that Demaine et al. [31] define GreedyFuture and GeometricGreedy without initial tree, and thus their algorithm (implicitly) performs this preprocessing.

We have little to say about the geometric views of Rotate-once (Figure 2.9(ii)), Splay (Figure 2.9(iv)), and other natural BST algorithms. The geometric view discussed in this section seems to give limited intuition about these algorithms. We have seen that every online satisfied superset algorithm must, at every time, touch at least the stair of the current access point. This is also true for the geometric view of Splay (and other BST algorithms). Also, the choice of initial tree can be enforced in geometric view for Splay and other algorithms similarly as for GreedyFuture.

Algorithms other than Greedy, however, also touch other points besides the (local) minimum given by the stair, even if they have the search-path-only property in tree-view, as Splay does. This may seem contradictory, since we said that the stair in geometric view corresponds to the search path in tree view. This is however, only true for Greedy. For Splay, and for other algorithms, the exact re-arrangement of the search path is not explicitly represented (or rather, it is not represented where it happens, but it is spread around in different rows in the future). The reason why Splay in geometric view touches more points at time i than the minimum necessary, is that it re-arranged the search paths at earlier times in particular ways. Touching more points than necessary at any given time may turn out to be helpful in the future. It remains an interesting question to decide locally which extra points may be worth touching, a question equivalent to designing a good online BST algorithm.

One can attempt to encode in the geometric view the exact re-arrangement of the search path in the exact line where it happens. Doing this would re-establish the link between search paths and stairs for every algorithm, at the cost of the simplicity of the geometric model [25, § C]. It could well be that for analyzing Splay and other algorithms, some entirely different geometric or combinatorial representation will prove to be useful (see e.g. the techniques used by Pettie [80, 82]).

We close this subsection with the following whimsical question. OnlineGreedy is a tree-view algorithm with cost proportional to the cost of GreedyFuture. What does the execution trace of OnlineGreedy look like in geometric view? We know that due to the split-tree data structure, OnlineGreedy may touch elements outside the search path, and thus, its geometric view will add points other than the stair to the solution (it is therefore different from GeometricGreedy). However, since OnlineGreedy is online, its geometric view is an algorithm for the online satisfied superset problem. Call this algorithm GeometricOnlineGreedy. We need not stop after one step, and can look at the tree-view simulation of GeometricOnlineGreedy (perhaps called OnlineGeometricOnlineGreedy) and so on, ad infinitum. Does this process have a non-trivial fixed point?

2.7.2 A closer look at GeometricGreedy

Based on the results of the previous subsection, it is tempting to "forget about trees" and analyze GeometricGreedy directly. We have seen that GeometricGreedy is a natural sweepline algorithm for solving the online satisfied superset problem. In fact, as it does the minimum work necessary at every time, one might at first think that it is optimal. The minimal example in Figure 2.12 shows that this is not the case. Informally, GeometricGreedy is not optimal, because, as the example shows, it may be worth to do more work locally than what is strictly necessary, as this may reduce the amount of work we need to do in the future.

We have already seen in § 2.5.3 that the performance of GreedyFuture (and by extension OnlineGreedy) is very good, matching several upper bounds. This makes OnlineGreedy, due to its simplicity in geometric view, perhaps the most promising contender for dynamic optimality (even if, in tree-view, OnlineGreedy is far from simple). In § 4 we analyze the behaviour of OnlineGreedy (in fact, GeometricGreedy) on a variety of input sequences, and in § 5 we give additional interpretations of GeometricGreedy.

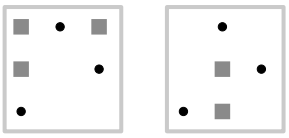

Figure 2.12: Smallest example where GeometricGreedy is not optimal. $X = (1,3,2)$. (left) GeometricGreedy output. (right) A smallest satisfied superset.

It appears quite plausible that GeometricGredy computes a constant approximation of the optimum (this must be the case, if GreedyFuture is constant-competitive, as conjectured, see e.g. Problem 20). In the following, let $\mathsf{OPT}(X)$ denote the optimum *satisfied superset* solution

size for X, and let $\text{cost}_{GG}(X)$ denote the cost of GeometricGreedy for X. Due to Theorem 2.24 and Theorem 2.28, these quantities *exactly* match the offline BST optimum, respectively the cost of GreedyFuture for X, according to the second offline BST model.

The worst known example [31] for the approximation ratio of GeometricGreedy is a construction where the ratio is arbitrarily close to $\frac{4}{3}$, see Figure 2.13. The following conjecture is therefore natural, although much stronger than the conjectured dynamic optimality of GreedyFuture.

Problem 28. Is $\text{cost}_{GG}(X) \leq \frac{4}{3} \cdot \mathsf{OPT}(X)$ for every X?

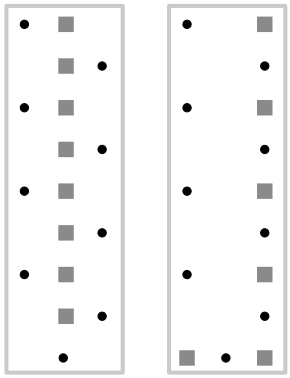

Figure 2.13: Example where the approximation ratio of GeometricGreedy is $\frac{4}{3}$. $X = (2,3,1,3,1,\ldots)$. (left) GeometricGreedy output. (right) Smallest satisfied superset.

In fact, an even stronger conjecture is suggested by Demaine et al. [31], and earlier by Munro [76], who speculate that the cost of GreedyFuture matches $\mathsf{OPT}(X)$ with an *additive* error of at most $O(m)$, or even just m. This conjecture is sensitive to constant factor modifications of the cost, therefore we need to be careful in specifying exactly which cost model we use. As far as we understand, it was formulated in what we call the second offline model, therefore, if true, it would also hold for GeometricGreedy. The example in Figure 2.14 shows that such a strong form of the conjecture is false. The figure presents an input sequence $X \in S_{30}$, i.e. a permutation of length $m = 30$, such that $\text{cost}_{GG}(X) \geq \mathsf{OPT}(X) + m + 1$. The exact sequence shown in the figure is $X = (18,9,17,16,25,19,4,10,26,24,8,13,5,28,23,20,7,12,1,29,27,3,22,11,14,2,30,21,6,15)$.

Observe that the example does not settle Problem 28, since the approximation ratio is not known to be more than $\frac{4}{3}$ in this example. The example in Figure 2.14 was obtained with computer search. It is not immediately clear how to extend it into an infinite family of examples, although visually it does seem to have a certain structure. It seems plausible that there are permutations of arbitrary size with a compact description, on which the error of GeometricGreedy is larger than m (computer experiments suggest that such examples are *easier* to come by as m grows, until the computation itself becomes too costly). Constructing structured examples (whether permutations or not) on which Greedy performs suboptimally remains an interesting direction. Whether such constructions will give new insight on the behavior of Greedy remains to be seen.

A last remark on the example in Figure 2.14 is that the cost of GeometricGreedy is compared here not with the exact optimum, but with the cost of GeometricGreedy ran sideways (which is an upper bound on OPT). It is easy to see that running GeometricGreedy (or any valid satisfied superset algorithm) on reversed, mirrored, or 90-degrees rotated variants of the input still produces a valid solution. The fact that time and (key-)space in the BST model can be interchanged is one of the more surprising and non-trivial insights of the geometric view. The exact relation between the costs for these variants is, however, poorly understood.

Problem 29. Is the cost of GeometricGreedy asymptotically the same on X and X^{rot}, where X^{rot} is X rotated by 90 degrees?

Based on the above discussion, the possibility that the following conjecture holds is not ruled out (again, this is a statement stronger than the dynamic optimality of GreedyFuture).

Problem 30. Is $\text{cost}_{GG}(X) \leq \mathsf{OPT}(X) + O(m)$ for every $X \in [n]^m$?

Regardless of GeometricGreedy, one may try to attack the geometric satisfied superset problem directly, using various tools from the field of geometric approximations. One possible direction would be to formulate the problem as a linear program. Surprisingly little has

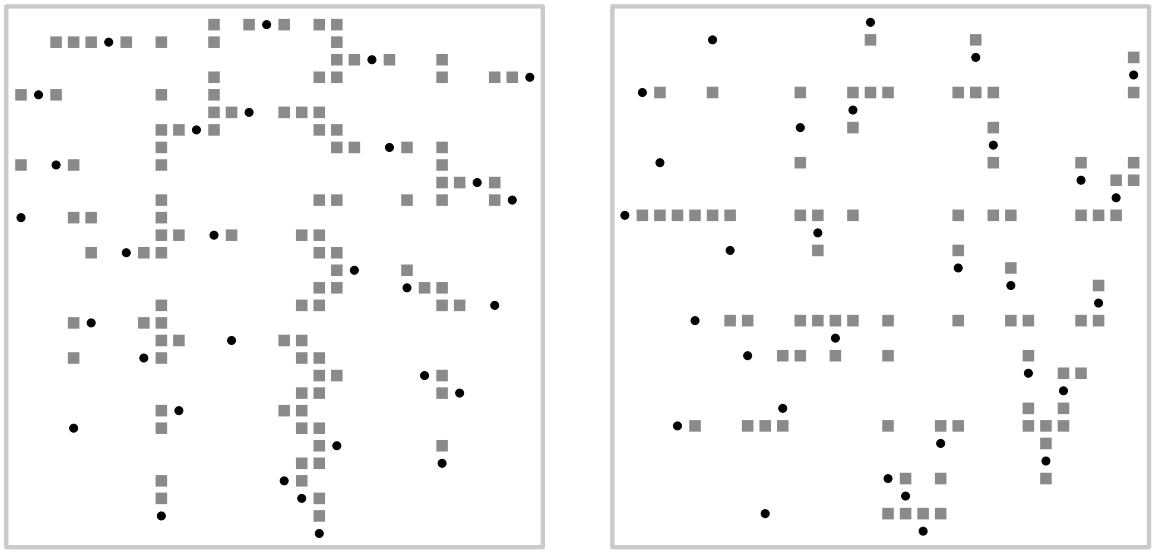

Figure 2.14: Counterexample to additive error conjecture for Greedy. Permutation sequence X of length 30. (left) GeometricGreedy output with $\text{cost}_{GG}(X) = 140$. (right) a different solution showing $\mathsf{OPT}(X) \leq 109$.

come so far from such attempts. One aspect of the satisfied superset problem that makes it unusual is its non-monotonicity: adding more points to a satisfied point set may render it unsatisfied. The hardness of the problem is also poorly understood (see Problem 3). Demaine et al. [31] prove the NP-hardness of the problem in the general case where the input may contain multiple points in the same column, as well as in the same row. The following intriguing question is thus open.

Problem 31. Is minimum satisfied superset NP-hard for point sets with one point in every row?

2.7.3 Lower bounds in geometric view

A rectangle *formed* by two points x and y (not on the same vertical or horizontal line) is the unique axis-parallel rectangle whose two corners are x and y. If one of x and y is to the right and above of the other, we say that the rectangle formed by x and y is a $\boxslash$-rectangle. Otherwise, we call it a $\boxbslash$-rectangle.

Let $\mathcal{I}_X$ denote the set of rectangles formed by *unsatisfied* pairs of points in X. A set of rectangles $\mathcal{R} \subseteq \mathcal{I}_X$ is called $\boxslash$-*independent*, if all rectangles in $\mathcal{R}$ are $\boxslash$-rectangles, and for any two rectangles $R_1, R_2 \in \mathcal{R}$, no corner of R_1 is fully inside R_2 (a corner of R_1 on the boundary of R_2 is allowed, see Figure 2.15). We define the condition for a set of rectangles $\mathcal{R} \subseteq \mathcal{I}_X$ to be $\boxbslash$-*independent* analogously.

Let $X \in [n]^m$ be an access sequence (and corresponding point set). The *maximum independent rectangle* (MIR) bound is defined as follows: $\mathsf{MIR}(X) = m + |\mathcal{R}_{\boxslash}| + |\mathcal{R}_{\boxbslash}|$, where $\mathcal{R}_{\boxslash}, \mathcal{R}_{\boxbslash} \subseteq \mathcal{I}_X$ are the largest $\boxslash$-*independent*, respectively $\boxbslash$-*independent* sets of rectangles in $\mathcal{I}_X$. The following result was shown by Demaine et al. [31, 52]. A similar lower bound was also described by Derryberry et al. [34].

Theorem 2.30 ([31])**.** $\mathsf{OPT}(X) = \Omega\big(\mathsf{MIR}(X)\big)$.

Theorem 2.30 provides yet another lower bound for OPT. In fact, MIR can be thought of as a family of bounds, since all (possibly non-maximal) sets of independent rectangles give a lower bound.

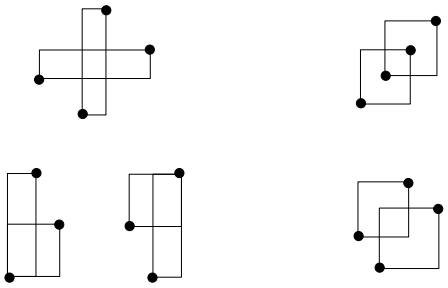

Figure 2.15: (left) Pairs of independent ◪-rectangles. (right) Pairs of non-independent ◪-rectangles.

In [31] it is shown that the MIR bound subsumes Wilber's two bounds (i.e. $\mathsf{MIR}(X)$ is asymptotically at most as large as $\mathcal{W}^1(X)$ or $\mathcal{W}^2(X)$ for all X). This can be seen by constructing particular sets of independent rectangles that exactly capture Wilber's bounds. Thus, Theorem 2.30 implies Theorems 2.22 and 2.23, which is fortunate, since the original proofs of Wilber for these results are quite complicated compared to the proof of Theorem 2.30 [31].

Surprisingly, there is a simple algorithm for computing a constant-approximation of the quantity $\mathsf{MIR}(X)$, found by Demaine et al. [31]. Moreover, this algorithm shows a deep similarity to GeometricGreedy. We describe in the following the two algorithms Greedy$^{◪}$ and Greedy$^{◹}$, which are similar to Greedy, differing only in the fact that they consider unsatisfied rectangles of the ◪-, respectively ◹-type only. We are interested in the union of their outputs.

For that purpose, let us define $\mathsf{stair}^{◪}$ and $\mathsf{stair}^{◹}$ at time i, similarly to the normal stair defined earlier. Thus, let

$$\mathsf{stair}^{◪}_i(x_i) = \mathsf{stair}_i(x_i) \cap \{1, \ldots, x_i - 1\},$$

and similarly:

$$\mathsf{stair}^{◹}_i(x_i) = \mathsf{stair}_i(x_i) \cap \{x_i + 1, \ldots, n\}.$$

Greedy$^{◪}$
For each $i = 1, \ldots, m$:
1. Read point $(x_i, i) \in [n] \times [m]$.
2. Output $Y_i = \{(x_i, i)\} \cup \{(y, i) : y \in \mathsf{stair}^{◪}_i(x_i)\}$.

Greedy$^{◹}$
For each $i = 1, \ldots, m$:
1. Read point $(x_i, i) \in [n] \times [m]$.
2. Output $Y_i = \{(x_i, i)\} \cup \{(y, i) : y \in \mathsf{stair}^{◹}_i(x_i)\}$.

The outputs of Greedy$^{◪}$ and Greedy$^{◹}$ are not, in general, valid solutions for the satisfied superset problem with input X (and neither is their union). More precisely, the output of Greedy$^{◪}$ (Greedy$^{◹}$) contains no unsatisfied ◪-type (◹-type) pairs of points, therefore the union of the Greedy$^{◪}$ and Greedy$^{◹}$ solutions contains no unsatisfied pairs of *input points* of either type. It may, however, contain unsatisfied pairs involving one or two output points.

Let $\mathrm{cost}_{G^{◪}}(X)$, and $\mathrm{cost}_{G^{◹}}(X)$ denote the cost of Greedy$^{◪}$, respectively Greedy$^{◹}$, i.e. the cardinalities of their outputs. The SignedGreedy bound, denoted $\mathsf{SG}(X)$ is defined simply as

$$\mathsf{SG}(X) = \mathrm{cost}_{G^{◪}}(X) + \mathrm{cost}_{G^{◹}}(X).$$

The following theorem holds.

Theorem 2.31 ([31])**.** For every $X \in [n]^m$ we have $\mathsf{SG}(X) = \Theta\big(\mathsf{MIR}(X)\big)$.

We revisit these quantities in § 4.7 where we analyze $\mathsf{MIR}(X)$ for various special sequences X, and in § 5, where we look at other quantities asymptotically equal to $\mathsf{MIR}(X)$, and give different interpretations of this quantity (which also yield alternative proofs for Theorem 2.30). The following important questions are open.

Problem 32. Prove or disprove that $\mathcal{W}^1(X) = \Theta\big(\mathsf{MIR}(X)\big)$ or $\mathcal{W}^2(X) = \Theta\big(\mathsf{MIR}(X)\big)$ hold for all $X \in [n]^m$.

Problem 33. Prove or disprove that $\mathsf{MIR}(X) = \Theta\big(\mathsf{OPT}(X)\big)$ for all $X \in [n]^m$.

Chapter 3

Splay tree: variations on a theme

> There is no excellent beauty that hath not some strangeness in the proportion.
>
> — FRANCIS BACON, *Of Beauty (1612)*

In this chapter we study the Splay algorithm ([91], § 2.5.2) in a broader context, as a particular member of a class of algorithms. Recall that Splay is a strict online BST algorithm according to the model defined in § 2.2. Splay is known to be very efficient both in theory and practice. We would like to understand why Splay is efficient, when similar algorithms such as Move-to-root and Rotate-once can be very inefficient, even on some of the simplest inputs.

By efficiency of an algorithm we mean that its cost is below the various upper bounds discussed in § 2.4. At a minimum, we expect a BST algorithm to satisfy the balance condition, i.e. to have cost $m \cdot O(\log n)$, when serving an access sequence $X \in [n]^m$, since this can already be achieved by a static balanced BST. More strongly, we will be concerned mainly with the upper bounds implied by the access lemma (Definition 2.15, Theorem 2.17).

Since we want logarithmic cost *on average* (i.e. not necessarily for every access), we do not require algorithms to maintain a balanced BST at all times. Nevertheless, for most access sequences, we expect efficient algorithms to maintain a reasonably balanced tree *most of the time* (as otherwise an adversary could always be asking for keys with large depth).

As a first attempt, let us look at the trees maintained by the various algorithms in some concrete (small) examples. For a BST algorithm $\mathcal{A}$ and some fixed n, consider the graph $G_{\mathcal{A}}$ whose vertices correspond to the C_n different BSTs over $[n]$. Accessing a certain key by algorithm $\mathcal{A}$ transforms the underlying BST. We represent such a transformation as a directed edge in $G_{\mathcal{A}}$ from the BST before the access to the BST after the access. We call $G_{\mathcal{A}}$ the *transition graph* of $\mathcal{A}$. We illustrate in Figure 3.1 the transition graphs of three algorithms for very small values of n (3 and 4).

Surprisingly, in the $n = 3$ case, the transition graph of Splay is not strongly connected, i.e. there exist pairs of trees T_1, T_2, such that having T_1 as initial tree, T_2 is unreachable. In the case of Move-to-root and Rotate-once, it can be shown easily that from an arbitrary initial tree, every possible tree is reachable through some sequence of accesses, for arbitrary size n. Even for Splay, the above observation turns out to be an artefact for the small value of n considered, and not a general phenomenon (Robert Tarjan, personal communication).

Second, at least in these tiny examples, intuitively it seems that BSTs with smaller depths (i.e. that are more balanced) are somehow "more central", i.e. "easier to reach" in the case of Splay than in the case of Rotate-once. (Such a difference between Splay and Move-to-root is less apparent.) Is it possible to precisely formulate and prove such an intuition?

Recall from § 2.2 that a strict online BST algorithm $\mathcal{A}$ is defined by its initial tree T_0 and its transition function (denoted $\Gamma_{\mathcal{A}}$). Can we understand the structure of the transition graph

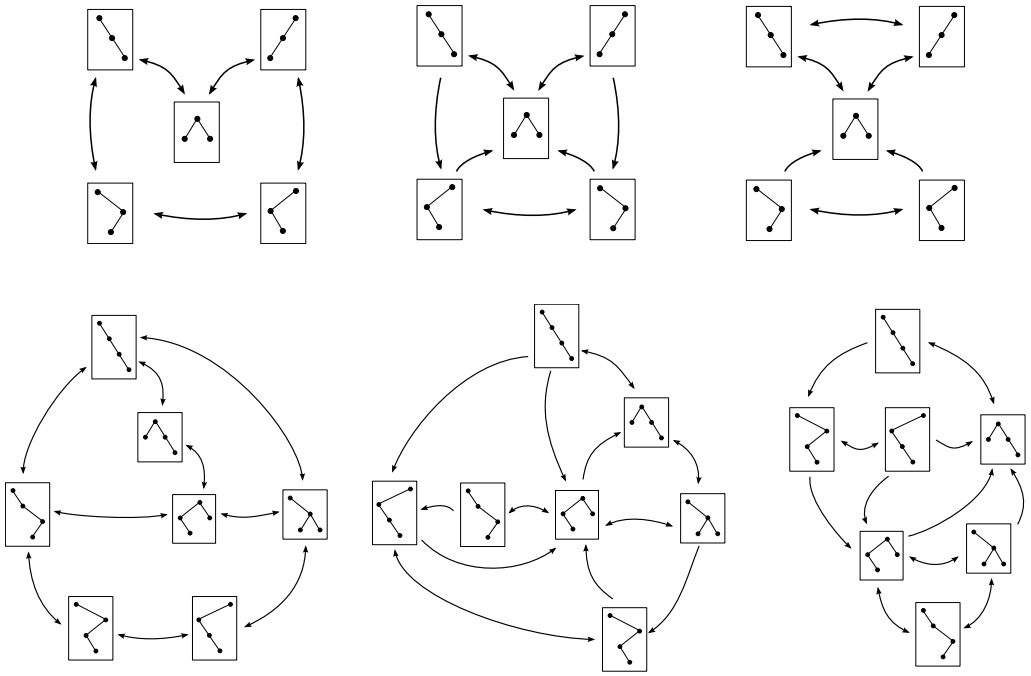

Figure 3.1: Transition graphs of Rotate-once (left), Move-to-root (middle), and Splay (right), for BSTs over [3] (above) and over [4] (below). Self-loops and multiple edges omitted. In the $n = 4$ case, isomorphic states are collapsed.

$G_{\mathcal{A}}$ in terms of properties of $\Gamma_{\mathcal{A}}$? More importantly, from structural properties of $G_{\mathcal{A}}$ can we directly infer something about the behavior and efficiency of $\mathcal{A}$?

At this point, such a full understanding of BST algorithms from "first principles" seems too ambitious. We will instead look at restricted classes of strict online BST algorithms, or equivalently, at transition functions $\Gamma_{\mathcal{A}}$ with certain properties. We would like to identify properties of $\Gamma_{\mathcal{A}}$ that guarantee the efficiency of the BST algorithm $\mathcal{A}$, and to understand how this relates to the efficiency of Splay and other known algorithms.

3.1 Splay revisited: a local view

Recall the definition of a strict online BST algorithm (§ 2.2) accessing a sequence $X = (x_1, \ldots, x_m) \in [n]^m$. Such an algorithm is fully described by an initial tree T_0 and a transition function Γ that transforms, after every access x_i, the search path P_i of x_i into a tree Q_i of the same size. We call the tree Q_i resulting from this transformation the *after-tree*. All subtrees hanging from the search path P_i are re-attached in a unique way to the after-tree Q_i. Transforming P_i to Q_i changes the tree T_{i-1} into T_i.

In this chapter we only consider strict online BST algorithms that have the access-to-root property. That is, we require Γ to be such that x_i is the root of Q_i (and consequently, of T_i). For most of the discussed results the restriction can be relaxed, although at the expense of some technicalities. Observe that both Splay and Move-to-root satisfy the access-to-root condition, but Rotate-once, in general, does not.

Recall the definition of Splay (§ 2.5.2). The transition function Γ is, in the case of Splay, defined by a series of local transformations (the ZIG, ZIG-ZAG, and ZIG-ZIG cases) applied to the search path until the accessed key becomes the root. This suggests a generalization of Splay, where Γ similarly moves the accessed key to the root, through a possibly different set of

local transformations. (Move-to-root also fits this general model – in the case of Move-to-root the local transformations are simple rotations.)

Such a generalization of Splay was studied by Subramanian [99] in 1996, and in a slightly more general form by Georgakopoulos and McClurkin [48] in 2004. In the remainder of the section we give a brief account of Subramanian's result.

Consider the search path P_i and the after-tree Q_i when accessing x_i. For simplicity, we drop the indices and refer to these objects as P, Q, and x. A *decomposition* of the transformation $\Gamma : P \to Q$ is a sequence of BSTs ($P = Q^0 \xrightarrow{P^0} Q^1 \xrightarrow{P^1} \cdots \xrightarrow{P^{k-1}} Q^k = Q$), such that for all i, the tree Q^{i+1} can be obtained from the tree Q^i, by rearranging a path P^i contained in Q^i into a tree R^i, and linking all the attached subtrees in the unique way given by the ordering condition. Clearly, every transformation $P \to Q$ has such a decomposition, since a sequence of rotations fulfills the requirement.

We call a strict online BST algorithm $\mathcal{A}$ with the access-to-root property *local with window-size c*, if all the transformations $P \to Q$ defined by its transition function $\Gamma_{\mathcal{A}}$ have a decomposition with the following properties:

(i) **(start)** $x \in P^0$, where x is the accessed element in P,

(ii) **(progress)** $P^{i+1} \setminus P^i \neq \emptyset$, for all i,

(iii) **(overlap)** $P^{i+1} \cap P^i \neq \emptyset$, for all i,

(iv) **(no-revisit)** $(P^i - P^{i+1}) \cap P^j = \emptyset$, for all $j > i$,

(v) **(window-size)** $|P^i| \leq c$, for some constant $c > 0$.

The following statements are straightforward to verify.

Theorem 3.1. Splay is local with window-size 3. Move-to-root is local with window-size 2.

We define two additional conditions on the decomposition of transformations $P \to Q$.

(vi) **(progress)** x is the leaf of P^i, and the root of R^i, for all i,

(vii) **(depth-reduction)** the transformation $P^i \to R^i$ strictly reduces the depths of both children of x (if they exist), for a constant fraction of the cases i.

Again, it is straightforward to verify that Splay fulfills the conditions (vi) and (vii) by inspecting Figure 2.3. Observe that, it is only true for the ZIG-ZIG and ZIG-ZAG cases that the two children of the accessed element x decrease their depth. This condition does not hold in the ZIG case, but since this case is applied at most once for every access, condition (vii) is satisfied. On the other hand, Move-to-root does not satisfy condition (vii).

The goal of the above definitions is to generalize Splay to a broader class of algorithms, replacing the local ZIG, ZIG-ZAG, and ZIG-ZIG cases by more general families of transformations. The above definition of locality (conditions (i)-(v)) is slightly different from the definitions of [99, 48], but not in an essential way. We state the result of Subramanian [99] in a slightly weaker form than the original. In particular, [99] does not require the access-to-root property. Recall the access lemma (Definition 2.15) from which several properties of a strict online algorithm with the access-to-root property can be derived (Theorem 2.17). The following result also implies Theorem 2.16.

Theorem 3.2 ([99])**.** Let $\mathcal{A}$ be a strict online BST algorithm with the access-to-root property that is local with window-size $c = O(1)$ (conditions (i)-(v)), and additionally, it satisfies conditions (vi) and (vii). Then the access lemma holds for $\mathcal{A}$. ∎

Proof. Recall the definition of the potential function Φ from § 2.5.2. Observe that the subtrees of nodes that are not on the search path P do not change during the re-arrangement, and therefore their contribution to the potential function Φ remains the same. We only need to account for the contribution of nodes of P. We apply the transformation $P \to Q$ step-by-step, according to the decomposition. Let $\Delta\Phi^i$ denote the increase in potential during the application of a single transformation $P^i \to R^i$. Let $\mathsf{subtree}^i(x)$ denote the subtree of node x after the transformation $P^i \to R^i$ has been applied. For convenience, we denote $w\big(\mathsf{subtree}^i(x)\big)$ by $ws^i(x)$. We have:

$$\Delta\Phi^i = \sum_{v \in P^i} \Big(\log\big(ws^i(v)\big) - \log\big(ws^{i-1}(v)\big)\Big) \tag{3.1}$$

$$\leq c \cdot \Big(\log\big(ws^i(x)\big) - \log\big(ws^{i-1}(x)\big)\Big). \tag{3.2}$$

Inequality (3.2) follows from condition (vi), namely, the observations that $v \in \mathsf{subtree}^i(x)$ and $x \in \mathsf{subtree}^{i-1}(v)$, for every $v \in P^i$. So far, we have not used condition (vii), and indeed, (3.2) is not strong enough to prove the access lemma. Therefore, we "perturb" the potential difference $\Delta\Phi^i$ in (3.1) to bring in an additional constant term. A local transformation $P^i \to R^i$ can be of two types. The tree R^i is either a path, or not. We consider the two cases in turn.

Case 1 (R^i is a path): Let L and R be the left, respectively right child of x before the transformation $P^i \to R^i$, and let B be the node in R^i of largest depth. If, as required by condition (vii), the depths of L and R decrease, then $\mathsf{subtree}^i(B)$ is disjoint from both $\mathsf{subtree}^{i-1}(L)$ and $\mathsf{subtree}^{i-1}(R)$.

Thus,

$$ws^i(x) \geq ws^i(B) + ws^{i-1}(x).$$

From the fact that $(X - Y)^2 \geq 0$, it follows that:

$$\Big(ws^i(x)\Big)^2 \geq 4 \cdot ws^i(B) \cdot ws^{i-1}(x).$$

Taking logarithms, we obtain

$$2 \cdot \log\big(ws^i(x)\big) \geq 2 + \log\big(ws^i(B)\big) + \log\big(ws^{i-1}(x)\big). \tag{3.3}$$

Case 2 (R^i is not a path): Let $r = \mathsf{root}(P^i)$, and let L and R be two sibling nodes in R^i (since R^i is not a path, two such nodes must exist). By disjointness of $\mathsf{subtree}^i(L)$ and $\mathsf{subtree}^i(R)$, we get

$$ws^{i-1}(r) \geq ws^i(L) + ws^{i-1}(R).$$

In consequence,

$$2 \cdot \log\big(ws^{i-1}(r)\big) \geq 2 + \log\big(ws^i(L)\big) + \log\big(ws^i(R)\big). \tag{3.4}$$

Plugging (3.3) and (3.4) into (3.1), we obtain:

$$\Delta\Phi^i \leq (c+1) \cdot \Big(\log\big(ws^i(x)\big) - \log\big(ws^{i-1}(x)\big)\Big) - 2. \tag{3.5}$$

For transformations $P^i \to R^i$ with the depth-reduction property we have (3.5). For the remaining transformations we fall back to (3.2). Let T and T' denote the tree before and after the access by $\mathcal{A}$. Since a constant fraction of the transformations have the depth-reduction property, by a telescoping sum we obtain:

$$\Phi(T) - \Phi(T') = \sum_i \Delta\Phi^i \le O\left(1 + \log \frac{W}{w(x)}\right) - \Omega(1) \cdot \frac{|P|}{c}.$$

The statement is the same as the access lemma (Definition 2.15). In the last step we used the facts that $x = \mathsf{root}(T)$, $w\big(\mathsf{subtree}(x)\big) \ge w(x)$, and that there are at least $\frac{|P|}{c}$ steps in the decomposition of the transformation. A factor of $c+1$ and an additive constant have been hidden in the $O(\cdot)$ term. ∎

Theorem 3.2 (and the more general formulations in [99, 48]) already give a broad generalization of Splay, and a sufficient condition for the efficiency of a strict online BST algorithm (in the sense of the access lemma). The resulting general class of algorithms is however, still "local", relying on the decomposition of the transition function Γ into constant-sized transformations. In the next sections we develop a more "global" description of BST algorithms satisfying the access lemma.

3.2 Splay revisited: a global view

Let P be the search path when accessing x in T. The transition function Γ of Splay transforms P into a tree Q. We view this transformation globally, illustrated in Figure 3.2.

Let $S = (a_0, a_1, \ldots a_{|P|-1})$ be the sequence of the keys in P, in their order of appearance on the search path from x to $\mathsf{root}(T)$, i.e. $a_0 = x$ and $a_{|P|-1} = \mathsf{root}(T)$.

Consider the following two-step transformation (Figure 3.2). First, make x the root and split the search path P into two paths, the path of elements smaller than x, and the path of elements larger than x. Second, for all i, if the pair of keys a_{2i+1} and a_{2i+2} are on the same side of x (i.e. both smaller or both larger), rotate the edge (a_{2i+1}, a_{2i+2}). In other words, remove a_{2i+2} from the path and make it a child of a_{2i+1}.

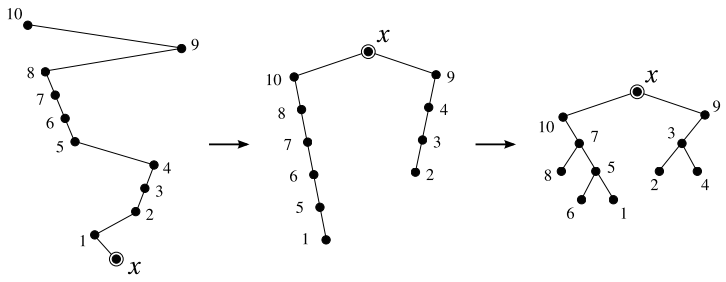

Figure 3.2: Global view of Splay. The first transformation illustrates Move-to-root. The first and second transformations together illustrate Splay. Subtrees hanging from the search path are omitted.

Theorem 3.3. The above two-step transformation describes exactly the transformation of Splay. The first step alone describes the transformation of Move-to-root.

Proof. We execute a Splay access to x according to the original definition (§ 2.5.2) and maintain the invariant that for the nodes of the search path P that have already been visited, the subtree of x (in the tree Q being constructed) has exactly the structure given by the new definition. Observe that once a node is in the subtree of x, it stays there until the end of

the process. The invariant holds in the beginning when only x is visited, as the subtree of x contains only x. The inductive step is straightforward to verify: whether a ZIG-ZIG, ZIG-ZAG, or ZIG step is executed, the invariant is maintained. ∎

3.3 Sufficient conditions for efficiency

In this section we give conditions on the transition function $\Gamma_{\mathcal{A}}$ of a strict online algorithm $\mathcal{A}$ with the access-to-root property that are sufficient to guarantee that the access lemma holds for $\mathcal{A}$. First we define the combinatorial properties of trees that will be used.

Let P be the search path when accessing x in T by algorithm $\mathcal{A}$. The transition function $\Gamma_{\mathcal{A}}$ of $\mathcal{A}$ transforms P into a tree Q. Again, we view this transformation globally, and refer to Q as the *after-tree*. The transformation is illustrated in Figure 3.3.

Let z denote the number of edges on the search path connecting nodes on different sides of x. We refer to z as the *number of zigzags*. Define the *left-depth*, respectively *right-depth* of a node v as the number of $/$, respectively $\backslash$ edges on the path from the root to v. We will consider left- and right-depth mostly in the after-tree. We illustrate these parameters (and others) in Figure 3.3.

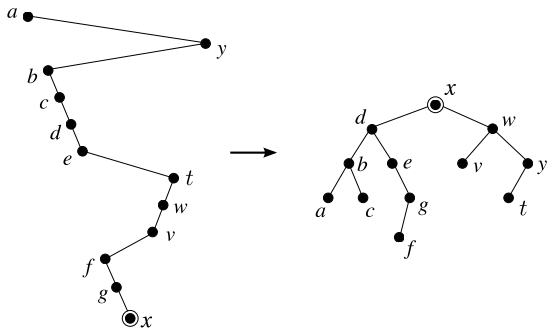

Figure 3.3: Parameters of the search path and the after-tree. The search path to x is shown on the left, and the after-tree is shown on the right. The search path consists of 12 nodes and contains four edges that connect nodes on different sides of x ($z = 4$). The after-tree has five leaves. The left-depth of a in the after-tree is three (the path from the root a to x goes left three times) and the right-depth of y is two. The set $\{a, c, f, v, y\}$ is subtree-disjoint. The sets $\{d, e, g\}$, $\{b, f\}$, $\{t, y\}$, $\{w\}$ are monotone. Subtrees hanging from the search path and from the after-tree are not shown.

The following theorem is the main result of this chapter.

Theorem 3.4. Let $\mathcal{A}$ be a strict online BST algorithm with the access-to-root property. If for every access x:

(i) the number of leaves of the after-tree is $\Omega(|P| - z)$ where P is the search path of x, and z is the number of zigzags in P, and

(ii) for every key $t \in P$ such that $t > x$ (resp. $t < x$), the right-depth of t (resp. left-depth of t) in the after-tree is $O(1)$, then the access lemma holds for $\mathcal{A}$.

Let T be the tree before accessing x and let T' be the tree after the re-arrangement. (Remember, that the search path P and the after-tree Q are subtrees of T, respectively, T'.) In the following, we write $\mathsf{subtree}_T(x)$ for the subtree of x in T, and we write $\mathsf{subtree}_{T'}(x)$, for the subtree of x in T'.

The main task in proving Theorem 3.4 is to relate the potential difference $\Phi_T - \Phi_{T'}$ to the length of the search path P. It is convenient to split the potential into parts that we can

argue about separately. For a subset $X \subseteq [n]$ of the nodes, define a partial potential on X as $\Phi_T(X) = \sum_{a \in X} \log\big(w\big(\mathsf{subtree}_T(a)\big)\big)$.

Again, observe that the potential change is determined only by the nodes on the search path, i.e.

$$\Phi_T - \Phi_{T'} = \Phi_T(P) - \Phi_{T'}(P).$$

Further, observe that we can argue about disjoint sets of nodes separately.

Lemma 3.5. Let P be the search path to x. Let $X = \bigcup_{i=1}^{k} X_i$ where the sets X_i are pairwise disjoint. Then $\Phi_T(X) - \Phi_{T'}(X) = \sum_{i=1}^{k} \big(\Phi_T(X_i) - \Phi_{T'}(X_i)\big)$.

We introduce three kinds of sets of nodes, namely *subtree-disjoint*, *monotone*, and *zigzag* sets, and derive bounds on the potential change for each of them. See Figure 3.3 for illustration.

A set of nodes $X \subseteq P$ is *subtree-disjoint* if $\mathsf{subtree}_{T'}(a) \cap \mathsf{subtree}_{T'}(b) = \emptyset$ for all pairs of distinct $a, b \in X$. Remark that subtree-disjointness is defined with respect to the subtrees after the access.

Again, let $(a_0, a_1, \ldots, a_{|P|-1})$ be the nodes of the search path P from x to the root of T. For each i, define the set $Z_i = \{a_i, a_{i+1}\}$ if a_i and a_{i+1} lie on different sides of x, and let $Z_i = \emptyset$ otherwise. The zigzag set Z_P is defined as $Z_P = \bigcup_i Z_i$. In words, the number of non-empty sets Z_i is exactly the number of zigzags in the search path, and the cardinality of Z_P is the number of elements involved in such zigzags.

A set of nodes $X \subseteq P$ is *monotone* if all elements in X are larger (smaller) than x and have the same right-depth (left-depth) in the after-tree Q.

We bound first the change of potential for subtree-disjoint sets. The proof of the following lemma was inspired by the proof of a statement similar to the access lemma for GreedyFuture by Fox [42]. In the following we use the notation $ws(x)$ for $w\big(\mathsf{subtree}_T(x)\big)$ and $ws'(x)$ for $w\big(\mathsf{subtree}_{T'}(x)\big)$. Further, we use the standard notation for open, closed, and mixed intervals of integers, e.g. for $a \le b$ we denote $[a,b] = \{a, a+1, \ldots, b\}$, $(a,b) = \{a+1, \ldots, b-1\}$, $(a,b] = \{a+1, \ldots, b\}$. Recall that $W = w(T)$.

Lemma 3.6. Let X be a subtree-disjoint set of nodes. Then

$$|X| \le 2 + 8 \cdot \log \frac{W}{ws(x)} + \Phi_T(X) - \Phi_{T'}(X).$$

Proof. We consider the nodes smaller than x (i.e. $X_{<x}$) and greater or equal to x (i.e. $X_{\ge x}$) separately. We show

$$|X_{\ge x}| \le 1 + \Phi_T(X_{\ge x}) - \Phi_{T'}(X_{\ge x}) + 4 \cdot \log \frac{W}{ws(x)}.$$

A symmetric statement holds for $X_{<x}$. We only give the proof for $X_{\ge x}$.

Denote $X_{\ge x}$ by $Y = \{a_0, a_1, \ldots, a_q\}$ where $x = a_0 < \ldots < a_q$. Before the access, x is a descendant of a_0, a_0 is a descendant of a_1, and so on. Let $\mathsf{subtree}_T(a_0) = [c,d]$. Then $[x, a_0] \subseteq [c,d]$ and $d < a_1$. Let $w_0 = ws(a_0)$. For $j \ge 0$, define σ_j as the largest index ℓ such that $w\big([c, a_\ell]\big) \le 2^j \cdot w_0$. Then $\sigma_0 = 0$ since weights are positive and $[c,d]$ is a proper subset of $[c, a_1]$. The set $\{\sigma_0, \ldots\}$ contains at most $\left\lceil \log \frac{W}{w_0} \right\rceil$ distinct elements. It contains a_0 and a_q.

Now we bound from above the number of indices i with the property $\sigma_j \le i < \sigma_{j+1}$. We call such an element a_i *heavy* if $ws'(a_i) > 2^{j-1} \cdot w_0$. There can be at most 3 heavy elements as otherwise

$$w\big([c, a_{j+1}]\big) \ge \sum_{\sigma_j \le k < \sigma_{j+1}} ws'(a_k) > 4 \cdot 2^{j-1} \cdot w_0,$$

a contradiction.

Next we count the number of light (a.k.a. non-heavy) elements. For every light element a_i, we have $ws'(a_i) \le 2^{j-1} \cdot w_0$. We also have $ws(a_{i+1}) \ge w\big([c, a_{i+1}]\big) > w\big([c, a_{\sigma_j}]\big)$ and thus $ws(a_{i+1}) > 2^j \cdot w_0$ by the definition of σ_j. Thus the ratio $r_i = \frac{ws(a_{i+1})}{ws'(a_i)} \ge 2$ whenever a_i is a light element. Moreover, for any $i = 0, \ldots, q-1$ (for which a_i is not necessarily light), we have $r_i \ge 1$. Thus,

$$2^{\text{number of light elements}} \le \prod_{0 \le i \le q-1} r_i = \left(\prod_{0 \le i \le q} \frac{ws(a_i)}{ws'(a_i)} \right) \cdot \frac{ws'(a_q)}{w_0}.$$

So the number of light elements is at most $\Phi_T(Y) - \Phi_{T'}(Y) + \log \frac{W}{w_0}$.
Putting the bounds together, and denoting $L = \log \frac{W}{w_0}$, we obtain

$$|Y| \le 1 + 3(\lceil L \rceil - 1) + \Phi_T(Y) - \Phi_{T'}(Y) + L \le 1 + 4L + \Phi_T(Y) - \Phi_{T'}(Y). \quad \blacksquare$$

We look next at monotone sets. We first make a simple observation, then we bound the contribution of monotone sets to the potential difference.

Lemma 3.7. Assume $x < a < b$ and that a is a proper descendant of b in P. If $\{a, b\}$ is monotone, then $\mathsf{subtree}_{T'}(a) \subseteq \mathsf{subtree}_T(b)$.

Proof. Clearly $[x, b] \subseteq \mathsf{subtree}_T(b)$. The smallest item in $\mathsf{subtree}_{T'}(a)$ is larger than x, and since a and b have the same right-depth, b is larger than all elements in $\mathsf{subtree}_{T'}(a)$. $\blacksquare$

Lemma 3.8. Let X be a monotone set of nodes. Then

$$\Phi(X) - \Phi'(X) + \log \frac{W}{w(x)} \ge 0.$$

Proof. We order the elements in $X = \{a_1, \ldots, a_q\}$ such that a_i is a proper descendant of a_{i+1} in the search path for all i. Then $\mathsf{subtree}_{T'}(a_i) \subseteq \mathsf{subtree}_T(a_{i+1})$ by monotonicity, and hence

$$\Phi(X) - \Phi'(X) = \log \frac{\prod_{a \in X} ws(a)}{\prod_{a \in X} ws'(a)} = \log \frac{ws(a_1)}{ws'(a_q)} + \sum_{i=1}^{q-1} \log \frac{ws(a_{i+1})}{ws'(a_i)}.$$

The second sum is nonnegative. Thus $\Phi(X) - \Phi'(X) \ge \log \frac{ws(a_1)}{ws'(a_q)} \ge \log \frac{w(x)}{W}$. $\blacksquare$

Theorem 3.9. Suppose that for every access to an element x, we can partition the elements on the search path P into k subtree-disjoint sets D_1 to D_k and ℓ monotone sets M_1 to M_ℓ. Then

$$\sum_{i \le k} |D_i| \le \Phi_T(S) - \Phi_{T'}(S) + 2k + (8k + \ell) \cdot \log \frac{W}{w(x)}.$$

The proof of Theorem 3.9 follows immediately from Lemma 3.6 and 3.8.

Finally, we look at zigzag sets. We view the transformation as a two-step process, i.e. we first rotate x to the root and then transform the left and right subtrees of x. Since we assume the access-to-root property, this is no restriction.

Lemma 3.10. $|Z| \le \Phi(Z_P) - \Phi'(Z_P) + O\left(1 + \log \frac{W}{ws(x)}\right)$.

Proof. Because x becomes the root, and ancestor relationships are otherwise preserved, $\mathsf{subtree}_{T'}(a) = \mathsf{subtree}_T(a) \cap (-\infty, x)$ if $a < x$, and $\mathsf{subtree}_{T'}(a) = \mathsf{subtree}_T(a) \cap (x, \infty)$ if $a > x$. We first deal with a single zigzag.

Lemma 3.11. $2 \le \Phi(Z_i) - \Phi'(Z_i) + \log \frac{ws(a_{i+1})}{ws(a_i)}$.

Proof. This proof is essentially the same as the proof for the ZIG-ZAG case for splay trees [91]. We give the proof only for the case where $a_i > x$ and $a_{i+1} < x$, as the other case is symmetric. Let a' be the left ancestor of a_{i+1} in P and let a'' be the right ancestor of a_i in P. If these elements do not exist, we set the values to $-\infty$ and $+\infty$, respectively. Let $W_1 = w\big((a',0)\big)$, $W_2 = w\big((0,a'')\big)$, and $W' = w\big((a_{i+1},0)\big)$. In T we have $ws(a_i) = W' + w(x) + W_2$ and $ws(a_{i+1}) = W_1 + w(x) + W_2$, and in T' we have $ws'(a_i) = W_2$ and $ws'(a_{i+1}) = W_1$.

Thus

$$\begin{aligned}
\Phi(Z_i) - \Phi'(Z_i) + \log \frac{W_1 + w(x) + W_2}{W' + w(x) + W_2} & \\
&\ge \log(W_1 + w(x) + W_2) - \log W_1 + \log(W_2 + w(x) + W') - \log W_2 \\
&+ \log \frac{W_1 + w(x) + W_2}{W' + w(x) + W_2} \\
&\ge 2 \cdot \log(W_1 + W_2) - \log W_1 - \log W_2 \\
&\ge 2,
\end{aligned}$$

since $(W_1 + W_2)^2 \ge 4 W_1 W_2$ for all W_1, W_2. ∎

Let Z_{even} (Z_{odd}) be the union of the Z_i sets with even (odd) indices. One of the two sets has cardinality at least $\frac{|Z_P|}{2}$. Assume that it is the former, the other case is symmetric. We sum the statement of the claim over all i in Z_{even} and obtain

$$\sum_{i \in Z_{even}} \left(\Phi(Z_i) - \Phi'(Z_i) + \log \frac{ws(a_{i+1})}{ws(a_i)} \right) \ge 2 \cdot |Z_{even}| \ge |Z_P| .$$

The elements in $Z_P \setminus Z_{even}$ form two monotone sets and hence

$$\Phi(Z_P \setminus Z_{even}) - \Phi'(Z_P \setminus Z_{even}) + 2 \cdot \log \frac{W}{ws(x)} \ge 0.$$

This completes the proof. ∎

The following theorem combines all three tools: subtree-disjoint, monotone, and zigzag sets.

Theorem 3.12. Suppose that for every access we can partition $P \setminus \{x\}$ into at most k subtree-disjoint sets D_1 to D_k and at most ℓ monotone sets M_1 to M_ℓ. Then

$$\sum_{i \le k} |D_i| + |Z_P| \le \Phi(P) - \Phi'(P) + O\left(\big(k + \ell\big)\big(1 + \log \frac{W}{w(x)}\big) \right).$$

Proof. We view the transformation as a two-step process, i.e. we first rotate x to the root and then transform the left and right subtrees of x. Let Φ'' be the potential of the intermediate tree. By Lemma 3.10,

$$|Z_P| \le \Phi(P) - \Phi''(P) + O\left(1 + \log \frac{W}{ws(x)}\right).$$

By Theorem 3.9,

$$\sum_{i \le k} |D_i| \le \Phi''(P) - \Phi'(P) + O\left(\big(k + \ell\big)\big(1 + \log \frac{W}{w(x)}\big) \right). \qquad ∎$$

To prove the main theorem, we just need the following proposition that follows directly from the definition of monotone set.

Lemma 3.13. Let $S \subseteq P$ be a set of keys consisting only of elements larger than x. Then S can be decomposed into ℓ monotone sets if and only if the elements of S have only ℓ different right-depths in the after-tree. A symmetric statement holds for elements smaller than x.

We are ready to prove the main theorem.

Proof of Theorem 3.4. Let $\mathcal{L}$ be the set of leaves of Q. By assumption (i) there is a positive constant c such that $|\mathcal{L}| \geq \frac{|Q|-z}{c}$. Then $|Q| \leq c \cdot |\mathcal{L}| + z$. We decompose $P \setminus \{x\}$ into $\mathcal{L}$ and ℓ monotone sets. By assumption (ii) $\ell = O(1)$. An application of Theorem 3.12 with $k = 1$ and $\ell = O(1)$ completes the proof. ∎

3.4 Applications

Theorem 3.4 implies the access lemma for essentially all BST algorithms for which it is known to hold (we refer here to online BST algorithms with the access-to-root restriction), as well as for some new ones.

Theorem 3.14 (Restatement of Theorem 2.16)**.** The access lemma holds for Splay.

Proof. Recall the global view of Splay from § 3.2. Consider the search path P for a splay access. There are $\frac{|P|}{2} - 1$ odd-even pairs of successive nodes on the search path. For each pair, if there is no side change, then Splay creates a new leaf in the after-tree. Let $\mathcal{L}$ denote the set of leaves in the after-tree, and let z denote the number of side changes (zigzags) in the search path. Thus

$$|\mathcal{L}| \geq \frac{|P|}{2} - 1 - z.$$

Since the right-depth (left-depth) of elements in the after-tree of Splay is at most 2, both requirements of Theorem 3.4 hold, and an application of the theorem finishes the proof. ∎

Theorem 3.4 also implies the access lemma for the generalizations of Splay by Subramanian [99] and by Georgakopulos and McClurkin [48], although only with the access-to-root restriction. We omit the details here, and refer to [25]. Furthermore, an analogue of Theorem 3.4 can be defined in geometric view (§ 2.7), and used to show that GreedyFuture and OnlineGreedy satisfy a statement analogous to the access lemma. (This is somewhat surprising since GreedyFuture does not have the access-to-root property, and OnlineGreedy is not even a strict algorithm in tree-view.) Again, we refer to [25] for details.

PathBalance. The PathBalance BST algorithm transforms the search path P into a balanced BST of depth $\lceil \log |P| \rceil$. This heuristic was suggested by Sleator (as reported in [99]). Observe that in the described form, PathBalance is a strict online algorithm, but in general it does not have the access-to-root property. To make the algorithm amenable to our tools, we change it slightly, such as to transform P into a balanced BST *rooted at the accessed element* x. The depth of this BST is at most $\lceil \log(1 + |P|) \rceil$, and there is no reason to believe that the modified version of the algorithm would have radically different performance compared to the original one. The following basic question raised by Sleator is open.

Problem 34. Is the cost of PathBalance $m \cdot O(\log n)$ for every (sufficiently long) access sequence $X \in [n]^m$?

In 1995 Balasubramanian and Raman [10] showed the upper bound of $m \cdot O\left(\log n \cdot \frac{\log\log n}{\log\log\log n}\right)$ on the cost of PathBalance, using a quite involved argument. This is still the best known guarantee. We show that a simple application of the tools developed in § 3.3 *almost* matches this bound.

First we observe that since the after-tree in PathBalance is a balanced BST, the number of leaves in the after-tree is $\Omega(|P|)$. There are, hovewer, nodes with $\Omega(\log|P|)$ right-depth or left-depth in the after-tree, which can be as large as $\Omega(\log n)$. Thus, we cannot directly apply Theorem 3.4. Nevertheless, the techniques developed in § 3.3 can be used to show the following result for PathBalance.

Lemma 3.15. $|P| \le \Phi(P) - \Phi'(P) + O\left(\left(1+\log|P|\right)\left(1+\log\frac{W}{w(x)}\right)\right)$.

Proof. We decompose P into sets P_0 to P_c, where P_k contains the nodes of depth k in the after-tree. Each P_k is subtree-disjoint. An application of Theorem 3.9 completes the proof. ∎

Theorem 3.16. PathBalance has cost at most $m \cdot O\left(\log n \cdot \log\log n\right)$ when accessing $X = (x_1, \ldots, x_m) \in [n]^m$.

Proof. We choose the uniform weight function: $w(a) = 1$ for all $a \in [n]$. Let c_i be the cost of accessing x_i, for $1 \le i \le m$, and let $C = \sum_{1\le i\le m} c_i$ be the total cost of accessing $X = (x_1, \ldots, x_m)$. Note that $\prod_i c_i \le (C/m)^m$. Since $\Phi(T) \le n\log n$ for any BST T over $[n]$, we have

$$C \le n\log n + \sum_{1\le i\le m} O\left(\left(1+\log c_i\right)\left(1+\log n\right)\right) = m \cdot O(\log n) \cdot \log(C/m)$$

by Lemma 3.15. Assume $C = K \cdot m\log n$ for some K. Then $K = O(1) + O(1)\cdot\log(K\cdot\log n)$ and hence $K = O(\log\log n)$. ∎

3.5 Monotonicity and locality

In this section we show that *locality* of a strict online BST algorithm, i.e. the fact that its transition function Γ has a decomposition that satisfies conditions (i)-(v) from § 3.1, is, in a precise sense, equivalent with the *monotonicity* condition described in § 3.3.

Recall that a strict online BST algorithm $\mathcal{A}$ with the access-to-root property is called *local with window-size c*, if all the transformations $P \to Q$ defined by its transition function $\Gamma_{\mathcal{A}}$ have a decomposition with properties (i)-(v) from § 3.1. We are mainly concerned with the case when $c = O(1)$.

Recall the definition of *monotone sets* from § 3.3. We call a strict online BST algorithm $\mathcal{A}$ with the access-to-root property *w-monotone*, if all after-trees generated by $\Gamma_{\mathcal{A}}$ can be partitioned into w monotone sets. By Lemma 3.13, this is equivalent with the condition that for every after-tree Q of $\Gamma_{\mathcal{A}}$, the nodes in Q have at most w different left- or right-depths. We are mainly concerned with the case when $w = O(1)$.

In the remainder of the section we prove the following theorem.

Theorem 3.17. Let $\mathcal{A}$ be a strict online BST algorithm with the access-to-root property. (i) If $\mathcal{A}$ is local with window size w, then it is $2w$-monotone. (ii) If $\mathcal{A}$ is w-monotone, then it is local with window-size w.

Let x denote the accessed element in the search path P (i.e. the root of Q).

(i) Suppose for contradiction that the after-tree Q is not decomposable into $2w$ monotone sets. As a corollary of Lemma 3.13, A contains a sequence of elements $x_1, x_2, \ldots, x_{w+1}$ such that either (a) $x < x_1 < \cdots < x_{w+1}$, or (b) $x_{w+1} < x_w < \cdots < x_1 < x$ holds, and x_{i+1} is a descendant of x_i for all i. Assume that case (a) holds, the other case is symmetric.

Let i' be the first index for which $x_{w+1} \in P^{i'}$. From the (window-size) condition we know that $P^{i'}$ contains at most w elements, and thus there exists some index $j < w+1$ such that $x_j \notin P^{i'}$. As x_j is a descendant of x_{w+1} in the search path P, it was on some path $P^{i''}$ for $i'' < i'$, and due to the (no-revisit) condition it will not be on another path in the future. Thus, it is impossible that x_j becomes an ancestor of x_{w+1}, so no local algorithm can create Q from P, a contradiction.

(ii) We give an explicit local algorithm $\mathcal{A}$ that creates the tree Q from the path P. As in Lemma 3.13, we decompose Q into $Q_{>x} = R_1 \cup \cdots \cup R_{w_R}$ and $Q_{<x} = L_1 \cup \cdots \cup L_{w_L}$ where R_i (respectively L_i) denote the set of elements whose search path contains exactly i edges of type $\diagdown$ (resp. of type $\diagup$). In other words, R_i and L_i denote the set of elements with right-depth, resp. left-depth equal to i. Let $L_0 = R_0 = \{x\}$. Denote as $P = (x_1, x_2, \ldots, x_k = x)$ the search path for x, i.e., x_1 is the root of the current tree and x_{j+1} is a child of x_j. For any j, let $t_j(R_i)$ be the element in $R_i \cap \{x_j, \ldots, x_k\}$ with minimal index, and define $t_j(L_i)$ analogously.

For any node s of Q, let the first right ancestor $FRA(s)$ be the first ancestor of s in Q that is larger than s (if any) and let the first left ancestor $FLA(s)$ be the first ancestor of s smaller than s (if any).

We start by making a few structural observations, which we prove later.

Lemma 3.18. Fix j, let $S = \{x_j, \ldots, x_k\}$, consider any $i \geq 1$, and let $s = t_j(R_i)$.

(i) If s is a right child in Q then its parent belongs to $S \cap R_{i-1}$.
(ii) If s is a left child in Q then $FRA(s) \notin S$ and $FLA(s) = t_j(R_{i-1})$.
(iii) If s is a right child and $FRA(s) \in S$ then all nodes in $\mathsf{subtree}_Q(s)$ belong to S.
(iv) If $FRA(s) \in S$ then $FRA(t_j(R_\ell)) \in S$ for all $\ell \geq i$.

We are now ready for the algorithm. We traverse the search path P of x backwards towards the root. Let $P = (x_1, x_2, \ldots, x_k = x)$. Assume that we have reached node x_j. Let $S = \{x_j, \ldots, x_k\}$ (the already *seen* nodes). We maintain an *active* set A of nodes. It consists of all $t_j(R_i)$ such that $FRA(t_j(R_i)) \notin S$ and all $t_j(L_i)$ such that $FLA(t_j(L_i)) \notin S$. When $j = k$, $A = S = \{x\}$. Consider any $y \in A$ such that $y > x$ (the argument for $y < x$ is symmetric and omitted). Assume that $\mathsf{parent}(y) \in S$. Then y must be a right child (since, otherwise $\mathsf{parent}(y) \notin S$, by definition of A). Furthermore, $FRA(y) \notin S$. Since $FRA(y) = FRA(\mathsf{parent}(y))$, it follows that $\mathsf{parent}(y)$ is also active.

By part (iv) of Lemma 3.18, there are indices ℓ and r such that exactly the nodes $t_j(L_\ell)$ to $t_j(R_r)$ are active. When $j = k$, only $t_j(R_0) = t_j(L_0) = x$ is active. We maintain the active nodes in a path P'. By the preceding paragraph, the nodes in $S \setminus A$ form subtrees of Q. We attach them to P' at the appropriate places and we also attach P' to the initial segment x_1 to x_{j-1} of P.

What are the actions required when we move from x_j to x_{j-1}? Assume $x_{j-1} > x$ and let $S' = \{x_{j-1}, \ldots, x_k\}$. Also assume that x_{j-1} belongs to R_i and hence $x_{j-1} = t_{j-1}(R_i)$. For all $\ell \neq i$, $t_j(R_\ell) = t_{j-1}(R_\ell)$. Notice that x_{j-1} is larger than all elements in S and hence $FRA(x_{j-1}) \notin S'$. Thus x_{j-1} becomes an active element and the $t_j(R_\ell)$ for $\ell < i$ are active and will stay active. All $t_j(R_\ell)$, $\ell > i$, with $FRA(t_j(R_\ell)) = x_{j-1}$ will become inactive and part of the subtree of Q formed by the inactive nodes between $t_j(R_i)$ and x_{j-1}. We change the path P' accordingly.

Verifying that the algorithm satisfies conditions (i)-(v) of a local algorithm is straightforward: the path P' is augmented by one new element in every step, its size is at most w at all times (since it contains at most one node from each L_i and R_i), and once a node becomes inactive (leaves P'), it never becomes active again.

It remains to prove the earlier structural observations.

Proof of Lemma 3.18.

(i) The parent of s lies between x and s and hence belongs to S. By definition of the R_i's, it also belongs to R_{i-1}.

(ii) $\mathsf{parent}(s) \in R_i$ and hence, by definition of $t_j(R_i)$, we have $\mathsf{parent}(s) \notin S$. Since $FLA(s) < s$, we have $FLA(s) \in S \cap R_{i-1}$. The element in R_{i-1} after $FLA(s)$ is larger than $\mathsf{parent}(s)$ and hence does not belong to S.

(iii) The elements between x and $FRA(s)$ (inclusive) belong to S.

(iv) Since $z = FRA(s) \in S$, s is a right child and z belongs to R_ℓ for some $\ell < i$. Consider any $\ell > i$, and let $q = t_j(R_\ell)$. Suppose that $q > z$. Then on the path between q and z (in Q) there is some element in $S \cap R_i$ greater than s, contradicting $s = t_j(R_i)$. Therefore, $q < z$, and hence $FRA(q) \le z$. Thus $FRA(t_j(R_\ell)) \in S$. ∎

We remark that the algorithm in the proof of Theorem 3.17 relies on advice about the global structure of the search path to after-tree transformation. In particular, it uses information about the nearest left- or right- ancestor of a node in the after-tree Q. Therefore, the result of this section can be interpreted as saying that transformations that are monotone *can be decomposed* and executed as a local transformation. This does not, however, imply that the resulting local algorithm has a compact description, or that the local transformations can be done using only locally available information (as in Splay).

Nonetheless, a limited amount of information about the already-processed structure of the search path can be encoded in the shape of the path P' that contains the active set A (the choice of the path shape is rather arbitrary, as long as the largest or the smallest element is at its root).

3.6 On the necessity of locality

In this section we look at whether the monotonicity condition (i.e. condition (ii) of Theorem 3.4), or equivalently the locality condition (i.e. conditions (i)-(v) from § 3.1), is necessary for an algorithm to be efficient. We show the following result, which can be seen as a partial converse of Theorem 3.4. We refer to § 3.9 for a discussion of its implications.

Theorem 3.19. If the access lemma with the sum-of-logs potential function holds for a BST algorithm $\mathcal{A}$, then the after-trees created by $\mathcal{A}$ must satisfy condition (ii) of Theorem 3.4, i.e. $\mathcal{A}$ is k-monotone with $k = O(1)$.

Proof. Consider a transformation $\Gamma_{\mathcal{A}} : P \to Q$ of a strict online BST algorithm $\mathcal{A}$ when accessing x. Suppose that $Q \setminus \{x\}$ cannot be decomposed into constantly many monotone sets. We want to show that $\mathcal{A}$ does not satisfy the access lemma with the sum-of-logs potential function Φ.

Assume w.l.o.g. that the right subtree of Q cannot be decomposed into constantly many monotone sets. Let $s > x$ be a node of maximum right depth in Q. By Lemma 3.13, we may assume that the right depth is $k = \omega(1)$. Let $x_1, \ldots, x_k$ be nodes on the path to s, such that $x < x_1 < \cdots < x_k < s$. All these nodes are descendants of s in the search path P.

Let us define a weight assignment to the elements of P and the trees hanging from P, for which the access lemma does not hold. Assign weight zero to all pendent trees, weight 1 to all proper descendants of s in P and weight K (where $K \gg 1$) to all ancestors of s in P. The total weight W then lies between K and $|P| \cdot K$.

We bound the potential change. We use the same notation as in the previous sections. Let $r(a) = \frac{ws'(a)}{ws(a)}$. In words, $r(a)$ is the ratio between the weights of the subtree rooted at a in the after-tree and in the search path. For all nodes x_i, we have $ws(x_i) \le |P|$ and $ws'(x_i) \ge K$.

So $r(x_i) \geq \frac{K}{|P|}$. Consider now any other node $a \in P$. If a is an ancestor of s in the search path, then $ws(a) \leq W$ and $ws'(a) \geq K$. If a is a descendant of s, then $ws(a) \leq |P|$ and $ws'(a) \geq 1$. Thus $r(a) \geq \frac{1}{|P|}$ for every $a \in P$. We conclude

$$\Phi'(T) - \Phi(T) \geq k \cdot \log \frac{K}{|P|} - |P| \cdot \log |P|.$$

If $\mathcal{A}$ satisfies the access lemma with the sum-of-logs potential function, then we must have:

$$\Phi'(T) - \Phi(T) \leq O\left(\log \frac{W}{w(x)} - |P|\right) = O\left(\log\left(K \cdot |P|\right)\right).$$

However, if K is large enough and $k = \omega(1)$, then $k \cdot \log \frac{K}{|P|} - |P| \cdot \log |P| \gg \Omega\left(\log\left(K \cdot |P|\right)\right)$, a contradiction. ∎

3.7 On the necessity of many leaves

In this section we study condition (i) of Theorem 3.4. This condition can be interpreted as saying that the number of leaves in the after-tree Q *or* the number of zigzags in the search path P must be proportional to $|P|$.

The fact that condition (ii) of Theorem 3.4 alone is not sufficient for an algorithm to satisfy the access lemma, follows from the easy observation that Move-to-root satisfies condition (ii), but not the access lemma. Informally, we can say that an algorithm must "do something else" besides being local (=monotone). It would be desirable to show that this "something" must be exactly condition (i) of Theorem 3.4. A conclusive statement in this direction would say that "if a sufficiently high fraction of the transformations done by $\mathcal{A}$ do not satisfy condition (i), then the access lemma cannot hold". Perhaps most insight would be gained from the description of a global adversary strategy that would force any algorithm that consistently violates (i) to have high total cost.

At this point, we are unable to prove such a statement. Instead, we relate condition (i) to a different (reasonable) measure of efficiency: the sequential access condition. (Recall that $\mathcal{A}$ satisfies the sequential access condition, if from every initial tree over $[n]$ it can serve the sequence $(1, \ldots, n)$ with cost $O(n)$.) We show the following theorem.

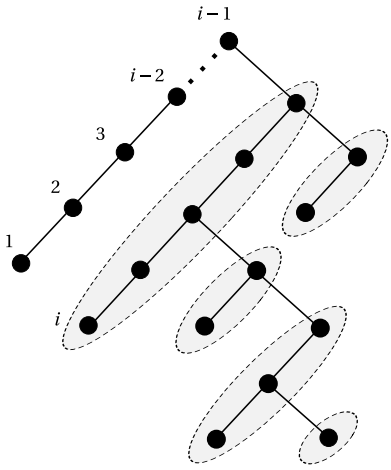

Figure 3.4: Illustration of the proof of Theorem 3.20. Tree after accessing node $i-1$, before accessing i. Wing partition shown with shaded ellipses.

Theorem 3.20. If for all after-trees Q created by algorithm $\mathcal{A}$, it holds that (i) Q can be decomposed into $O(1)$ monotone sets, and (ii) the number of leaves of Q is at most $n^{o(1)}$, then $\mathcal{A}$ does not satisfy the sequential access condition.

Observe that the value n in Theorem 3.20 is the global number of nodes (not just the number of nodes $|Q|$ on the search path). Before proving Theorem 3.20, we state the open question of whether the result can, in some way, be improved.

Problem 35. Can Theorem 3.20 be strengthened in any of the following ways?

1. Involving in the statement (instead of the sequential access condition) the balance condition, the access lemma, or some other measure of efficiency.
2. Involving in the statement the quantity z (number of zigzags).
3. Relaxing the condition that *every* transformation must create only few leaves.
4. Relaxing the dependence on the monotonicity condition.

5. Relaxing the bound $n^{o(1)}$ to, say, $o(n)$ or $o(|Q|)$.

Despite the shortcomings of Theorem 3.20, there are also some apparent strengths of this result (which can be seen as another partial converse of Theorem 3.4). In particular, the statement refers to the sequential access property, without relying on the sum-of-logs potential, or on any other proof technique. Second, as the quantity $n^{o(1)}$ involves n, and not $|Q|$, the result holds regardless of what the algorithm does when $|Q| \leq n^{o(1)}$, i.e. when the search path is very short. The remainder of the section is devoted to the proof of Theorem 3.20.

Let R be a BST over $[n]$. We call a maximal left-leaning path of R a *wing* of R. More precisely, a wing of R is a set $\{x_1, \ldots, x_k\} \subseteq [n]$, with $x_1 < \cdots < x_k$, and such that x_1 has no left child, x_k is either the root of R, or the right child of its parent, and x_i is the left child of x_{i+1} for all $1 \leq i < k$. A wing may consist of a single element. Observe that the wings of R partition $[n]$ in a unique way, and we call the set of wings of R the *wing partition* of R, denoted as $wp(R)$. We define a potential function Ψ over a BST R as follows:

$$\Psi(R) = \sum_{w \in wp(R)} |w| \cdot \log |w|.$$

Let T_0 be a left-leaning path over $[n]$ (i.e. n is the root and 1 is the leaf). Consider a strict online BST algorithm $\mathcal{A}$ with the access-to-root property. Suppose that $\mathcal{A}$ accesses elements of $[n]$ in sequential order, starting with T_0 as initial tree. Let T_i denote the BST after accessing element i. Then T_i has i as the root, and the elements yet to be accessed (i.e. $\{i+1, \ldots, n\}$) form the right subtree of the root, denoted R_i. To avoid treating T_0 separately, we augment it with a "virtual root" 0. This node plays no role in subsequent accesses, and it only adds a constant one to the overall access cost.

Using the previously defined potential function, we denote $\Psi_i = \Psi(R_i)$. We make the following easy observations: $\Psi_0 = n \log n$, and $\Psi_n = 0$.

Next, we look at the change in potential due to the restructuring after accessing element i. Let $P_i = (x_1, x_2, \ldots, x_{n_i})$ be the search path when accessing i in T_{i-1}, and let n_i denote its length, i.e. $x_1 = i - 1$, and $x_{n_i} = i$. Observe that the set $P'_i = P_i \setminus \{x_1\}$ is a wing of T_{i-1}.

Let Q_i be the after-tree resulting from the re-arranging of the path P_i. Observe that the root of Q_i is i, and the left child of i in Q_i is $i-1$. We denote the tree $Q_i \setminus \{i-1\}$ as Q'_i, and the tree $Q'_i \setminus \{i\}$, i.e. the right subtree of i in Q_i, as Q''_i.

The crucial observation of the proof is that for an arbitrary wing $w \in wp(T_i)$, the following holds: (i) either w was not changed when accessing i, i.e. $w \in wp(T_{i-1})$, or (ii) w contains a portion of P'_i, possibly concatenated with an earlier wing, i.e. there exists some $w' \in wp(Q'_i)$, such that $w' \subseteq w$. In this case, we denote as $\text{ext}(w')$ the *extension* of w' to a wing of $wp(T_i)$, i.e. $\text{ext}(w') = w \setminus w'$, and either $\text{ext}(w') = \emptyset$, or $\text{ext}(w') \in wp(T_{i-1})$.

Now we bound the change in potential $\Psi_i - \Psi_{i-1}$. Wings that did not change during the restructuring (i.e. those of type (i)) do not contribute to the potential difference. Also note, that i contributes to Ψ_{i-1}, but not to Ψ_i. Thus, we have for $1 \leq i \leq n$, assuming that $0 \log 0 = 0$, and denoting $f(x) = x \log(x)$:

$$\Psi_i - \Psi_{i-1} = \sum_{w' \in wp(Q''_i)} \left(f\left(\left|w'\right| + \left|\text{ext}(w')\right|\right) - f\left(\left|\text{ext}(w')\right|\right)\right) - f(n_i - 1).$$

By simple manipulation, for $1 \leq i \leq n$:

$$\Psi_i - \Psi_{i-1} \geq \sum_{w' \in wp(Q''_i)} f\left(\left|w'\right|\right) - f(n_i - 1).$$

By convexity of f, and observing that $|Q_i''| = n_i - 2$, we have

$$\Psi_i - \Psi_{i-1} \geq \left|wp(Q_i'')\right| \cdot f\left(\frac{n_i-2}{\left|wp(Q_i'')\right|}\right) - f(n_i-1) = (n_i-2)\cdot \log \frac{n_i-2}{\left|wp(Q_i'')\right|} - f(n_i-1).$$

Lemma 3.21. If R has right-depth m, and k leaves, then $\left|wp(R)\right| \leq mk$.

Proof. For a wing w, let $\ell(w)$ be any leaf in the subtree rooted at the node of maximum depth in the wing. Clearly, for any leaf ℓ there can be at most m wings w with $\ell(w) = \ell$. The claim follows. ∎

Thus, $\left|wp(Q_i'')\right| \leq n^{o(1)}$. Summing the potential differences over i, we get

$$\Psi_n - \Psi_0 = -n\log n \geq -\sum_{i=1}^{n} n_i \log(n^{o(1)}) - O(n).$$

Denoting the total cost of algorithm $\mathcal{A}$ on the access sequence $(1,\ldots,n)$ as C, we obtain $C = \sum_{i=1}^{n} n_i = n\cdot\omega(1)$. This shows that $\mathcal{A}$ does not satisfy the sequential access property.

3.8 Depth-halving

Already Sleator and Tarjan [91] formulated the belief that the property that makes Splay efficient is *depth-halving*, i.e. the fact that every element on the search path reduces its distance to the root by a factor of approximately two. Later authors [99, 10, 48] raised the question, whether such a *global* depth-reduction property is by itself sufficient to guarantee the access lemma. In this section we explore this question.

Let $\mathcal{A}$ be a strict online BST algorithm with the access-to-root property. As before, let P be the search path when searching for key x using algorithm $\mathcal{A}$, let Q be the resulting after-tree, and let T and T' be the trees before and after the access.

We say that $\mathcal{A}$ is *weakly depth-halving* if for every node $x \in P$ it holds that $d_{T'}(x) \leq d_T(x)/2 + c$, for some fixed constant $c \geq 0$. Let us observe the following easy fact.

Theorem 3.22 ([91])**.** Splay is weakly depth-halving.

Proof. The proof follows from the following observations, which can be verified by inspection of Figure 2.3 (note that the final ZIG step affects the depths by a constant one only.)

1. In every ZIG-ZIG or ZIG-ZAG step, descendants of x reduce their depth by at least 1.
2. A node of P may increase its depth by at most 2 before becoming the descendant of x
3. A node that is a descendant of x, will remain so throughout all remaining ZIG-ZIG and ZIG-ZAG steps of the current access.
4. When a node $y \in P$ becomes a proper descendant of x, the number of remaining ZIG-ZIG or ZIG-ZAG steps of the current access is at least $\lfloor d_T(y)/2 \rfloor - 1$.

(We remark that by a similar argument, Semi-splay (§ 2.5.2) is also weakly depth-halving, even though it does not have the access-to-root property.) ∎

If weak depth-halving leads to good properties of algorithms, one would expect that the reduction of depths by some smaller constant factor would also be sufficient. Such a constant-factor depth-reduction can be shown for the local algorithms with *local* depth-reduction conditions discussed in Theorem 3.2 and the rest of § 3.1. The following natural questions are open.

Problem 36. Does every weakly depth-halving strict online algorithm satisfy the balance condition? More strongly, does the access lemma hold for every such algorithm? How about the sequential access, dynamic finger, lazy finger, and $O(1)$-competitiveness properties?

Let us first show a negative result: weak depth-halving by itself does not imply the conditions of Theorem 3.4 (Figures 3.6, 3.7, 3.8 illustrate this). In some sense, Figure 3.8 is the strongest of the counterexamples, since in that case, even though the transformation is weakly depth-halving, the monotonicity-condition is violated (an anti-monotone path of linear size is created). It follows from Theorem 3.19 that weak depth-halving cannot imply the access lemma in its full generality with the sum-of-logs potential function.

A possible way to modify the definition of weak depth-halving would be to require depth-halving for the roots of the subtrees hanging from the search path, instead of the nodes of the search path themselves. The proof of Theorem 3.22 can be minimally adapted to show that Splay satisfies this modified form of depth-halving as well. Unfortunately, the transformations shown on Figures 3.6 and 3.8 satisfy this modified form of depth-halving, violating at the same time, some condition of Theorem 3.4.

Does perhaps weak depth-halving *and* monotonicity together imply the access lemma? This may still be the case, perhaps even provable by an argument involving the sum-of-logs potential, but not by our Theorem 3.4, as Figures 3.6 and 3.7 show.

In general, the reverse implication between the conditions of Theorem 3.4 and depth-halving does not hold either. It is easy to think of a transformation that is monotone, creates a linear number of leaves, yet it is not depth-halving (for instance, this is the case, if we run Splay only halfway up the search path).

In the remainder of the section we describe a stronger form of depth-halving, that does imply the conditions of Theorem 3.4, and thus guarantees the access lemma. This leads to a new family of efficient BST algorithms with a natural global description.

Let $\mathcal{A}$ be a strict online BST algorithm with the access-to-root property. Consider again the search path to after-tree transformation $P \to Q$, and let T and T' be the trees before and after the access.

Let $a, b \in P$ be two arbitrary nodes. If b is an ancestor of a in P, but not in the after-tree Q, then we say that a has *lost* the ancestor b, and b has lost the descendant a. Similarly we define *gaining* an ancestor or a descendant. We stress that only nodes on the search path (resp. the after-tree) are counted as descendants, and not the nodes of the pendent trees (i.e. only nodes in P, and not those in $[n] \setminus P$).

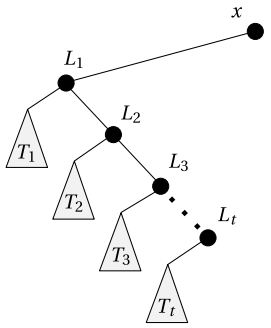

Figure 3.5: Illustration of the proof of Theorem 3.23. Left subtree of x in after-tree Q.

We say that $\mathcal{A}$ is *strongly depth-halving* if (i) every node $a \in P$ loses at least $(\frac{1}{2}+\epsilon)\cdot \mathrm{depth}_T(a) - c$ ancestors, for fixed constants $\epsilon > 0$, $c \geq 0$, and (ii) every node on the search path P, except the accessed element x, gains at most d new descendants, for a fixed constant $d \geq 0$.

Theorem 3.23. Let $\mathcal{A}$ be a strict online BST algorithm with the access-to-root property, that has the strong depth-halving property. Then the access lemma holds for $\mathcal{A}$.

We prove Theorem 3.23 by applying Theorem 3.4. Before that, let us argue that with our current proof techniques the conditions of Theorem 3.23 cannot be trivially strengthened. If we relax the constant in condition (i) from $(\frac{1}{2}+\epsilon)$ to $\frac{1}{2}$, the conditions of Theorem 3.4 are no longer implied. There exist rearrangements in which every node loses a $\frac{1}{2}$-fraction of its ancestors, gains at most two ancestors or descendants, yet both the number of zigzags and the number of leaves created are $O(\sqrt{|P|})$ (this is shown

in Figure 3.6). If we further relax the ratio to $(\frac{1}{2}-\epsilon)$, we can construct an example where the number of zigzags and the number of created leaves are only $O(\log|P|/\epsilon)$.

Allowing more gained descendants and limiting instead the number of gained ancestors is also beyond the strength of Theorem 3.4. It is possible to construct an example (Figure 3.7) in which every node loses an $(1-o(1))$-fraction of ancestors, yet the number of leaves created is only $O(\sqrt{|P|})$ (while having no zigzags in the search path).

Proof of Theorem 3.23. We show that $\mathcal{A}$ satisfies the conditions of Theorem 3.4.

Let x be the accessed element, and let L_1 be its left child in the after-tree. Let $(L_1,\ldots,L_t)$ denote the longest sequence of nodes such that for all $i<t$, L_{i+1} is the right child of L_i in the after-tree, and let T_i denote the left subtree of L_i for all $i\le t$. Observe that the nodes in T_i are ancestors of L_i in the search path, therefore, L_i has gained them as descendants. Thus, from condition (ii) of strong depth-halving, we have that $|T_i|\le d$ for all i. Since there are at most d nodes in each subtree, the largest number of left-edges on a path in the left subtree of x is d. A symmetric statement holds for the right subtree of x. This proves condition (ii) of Theorem 3.4 (see Figure 3.5).

Next, we show that a linear number of leaves are created, verifying condition (i) of Theorem 3.4. By $\mathrm{depth}(\cdot)$ we always mean $\mathrm{depth}_T(\cdot)$, i.e. depth in the tree before the rearrangement.

We claim that there exists a left-ancestor of x in the search path that loses $\frac{\epsilon\cdot\mathrm{depth}(x)}{2}-(c+1)$ left-ancestors, or a right-ancestor of x that loses this number of right-ancestors.

Suppose that there exists such a left-ancestor L of x (the argument on the right is entirely symmetric). Observe that the left-ancestors that L has *not lost* form a right-path, with subtrees hanging to the left. The lost left-ancestors of L are contained in these subtrees. From the earlier argument, each of these subtrees is of size at most d. Since the subtrees contain in total at least $\frac{\epsilon\cdot\mathrm{depth}(x)}{2}-(c+1)$ elements, there are at least

$$\frac{1}{d}\left(\frac{\epsilon\cdot\mathrm{depth}(x)}{2}-(c+1)\right)=\Omega\big(\mathrm{depth}(x)\big)$$

many of them, thus creating $\Omega\big(|P|\big)$ new leaves.

It remains to prove the claim that some ancestor of x loses many ancestors "on the same side". Let L and R be the nearest left- (respectively right-) ancestor of x on the search path. Assume w.l.o.g. that L is the parent of x in the search path. For any node y in the search path P, let $d_\ell(y)$, $d_r(y)$ denote the number of left- respectively right-ancestors of y in P. We consider two cases:

- If $d_\ell(x)>d_r(x)$, then $d_r(L)\le\frac{\mathrm{depth}(x)}{2}$. Since L loses

  $$\left(\frac{1}{2}+\epsilon\right)\cdot\mathrm{depth}(L)-c\ge\left(\frac{1}{2}+\epsilon\right)\cdot\mathrm{depth}(x)-(c+1)$$

  ancestors, it must lose at least $\epsilon\cdot\mathrm{depth}(x)-(c+1)$ left-ancestors.

- If $d_\ell(x)\le d_r(x)$, then $d_\ell(R)<d_r(R)$, and hence $d_\ell(R)\le\frac{\mathrm{depth}(R)}{2}$. At the same time,

  $$\mathrm{depth}(R)\ge d_r(R)=d_r(x)-1\ge\frac{\mathrm{depth}(x)-2}{2}.$$

  Since R loses $\left(\frac{1}{2}+\epsilon\right)\cdot\mathrm{depth}(R)-c$ ancestors, it must lose at least

  $$\left(\frac{1}{2}+\epsilon\right)\cdot\mathrm{depth}(R)-c-d_\ell(R)\ge\epsilon\cdot\frac{\mathrm{depth}(x)-2}{2}-c\ge\epsilon\cdot\frac{\mathrm{depth}(x)}{2}-(c+1)$$

right-ancestors. ∎

Unfortunately, the strong depth-halving condition is indeed rather strong, for instance, Splay does not satisfy it. Therefore, we close the section with the following general question.

Problem 37. Find a natural definition of depth-halving that Splay satisfies, and show that it implies the access lemma (or at least the balance condition).

3.9 Discussion and open questions

In this chapter we discussed a number of results related to the efficiency of strict online BST algorithms, based on properties of their transition function. We used the access lemma as a stand-in for "efficiency", as it implies a number of upper bounds for BST algorithms, including static optimality (Theorem 2.17). The ultimate goal of this research program would be a characterization of all efficient strict online BST algorithms. Unfortunately, we have not quite reached such a full understanding (even if we focus only on the access lemma).

In Theorem 3.4 we gave necessary conditions for a BST algorithm to satisfy the access lemma. We looked at strict online algorithms with the access-to-root property, and additionally we required them to be "monotone", as well as to involve "many leaves" or "many zigzags" in every transformation. The class of algorithms thus obtained includes Splay. An immediate follow-up question is the following.

Problem 38. Does every algorithm that satisfies the conditions of Theorem 3.4 have any of the sequential access, dynamic finger, lazy finger, and O(1)-competitiveness properties?

Of the parameters used in Theorem 3.4, the number of zigzags is to the least extent under the control of the algorithm designer, since it depends on the key being accessed, and it is likely that some key will have few zigzags on its search path. The reason why zigzags are helpful seems to be related to depth-reduction: as we move the accessed element x to the root, we break up edges between nodes on different sides of x, and thus we reduce the depths of elements on both sides of x.

The role played by the monotonicity condition seems relatively well understood. In particular, we have shown that it is equivalent with the "local decomposability" of an algorithm. We also showed that this condition is necessary for showing the access lemma via the sum-of-logs potential. Does this mean that non-monotone (=non-local) algorithms are automatically bad, or is this just a limitation of the sum-of-logs proof technique? We have seen that reasonable families of algorithms, such as PathBalance or weak depth-halving violate the monotonicity property. Can they satisfy the access lemma by some other potential function? Less ambitiously, can they satisfy the balance condition? (See also Problem 34.)

Problem 39. Is there a (natural) non-monotone strict online BST algorithm with access-to-root property that satisfies the balance condition?

Non-monotonicity means that long monotone subsequences of nodes are "reversed", i.e. their orientation changed from left-leaning to right-leaning. Such a transformation has the effect of significantly increasing the depth of some subtrees hanging from the search path. Since in the strict model we have no information about the sizes of these subtrees, Splay, and other algorithms with the monotone property take the conservative route: they do not reverse long monotone paths, since doing this could send large subtrees down the tree. (This is also the intuition behind the proof in § 3.6). It is not clear, however, how frequently such

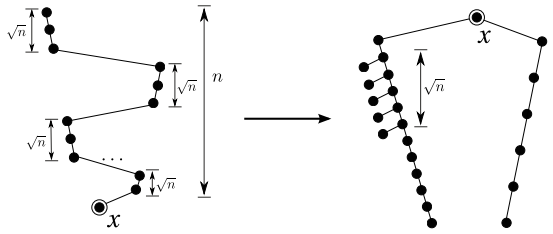

Figure 3.6: Weakly depth-halving re-arrangement of the search path P. The transformation does not satisfy the conditions of Theorem 3.4. Every node of P loses half of its ancestors, gains at most one new ancestor, and every node of P (except x) gains at most one new descendant. On the other hand, $z, \ell = O(\sqrt{n})$, where z is the number of zigzags in P, and ℓ is the number of leaves of the after-tree.

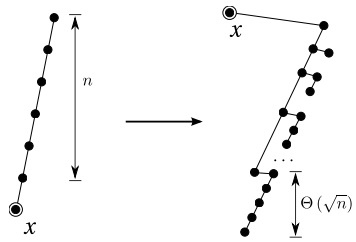

Figure 3.7: Weakly depth-halving re-arrangement of the search path P. The transformation does not satisfy the conditions of Theorem 3.4. Every node of P loses a $(1-o(1))$-fraction of its ancestors and gains at most one new ancestor. On the other hand, $z = 0$, and $\ell = O(\sqrt{n})$, where z is the number of zigzags in P, and ℓ is the number of leaves of the after-tree.

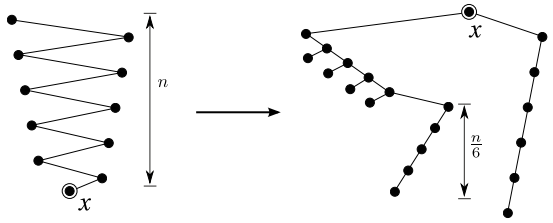

Figure 3.8: Weakly depth-halving re-arrangement of the search path P. The transformation does not satisfy the conditions of Theorem 3.4. Every node of P approximately halves its depth. However, there is a node left of the accessed node x with left-depth $\Omega(n)$.

bad scenarios can arise. To better understand the limitations of non-monotone algorithms, a global adversary argument would be helpful.

Subramanian [99] argues compellingly, that as long as we enforce the locality (=monotonicity) condition, almost every BST re-arrangement satisfies the access lemma. Loosely speaking, as long as our algorithm is local, it will do fine, unless most of the time it "does nothing", or if it "does what Move-to-root would do", and on top of that, it is unlucky enough to meet a search path that is left-leaning or right-leaning (i.e. few zigzags). "Doing fine" refers here to satisfying the access lemma, which is still far from satisfying dynamic optimality. (Proving that almost every algorithm in such a broad class satisfies dynamic optimality would be a very interesting, if somewhat anticlimactic turn of events.)

Perhaps "many leaves" is the most intriguing parameter of Theorem 3.4. It is intuitive that creating many branchings (and in consequence, many leaves) should be helpful, and in § 3.7 we showed that (assuming monotonicity), a weaker form of the "many leaves" condition is necessary for the sequential access property. We already asked whether this condition is necessary in a stronger sense (Problem 35). Could it be that the "many leaves" condition alone is sufficient?

Problem 40. If every transformation creates $\Omega(|P|)$ leaves, is the balance condition guaranteed? How about the sequential, dynamic finger, lazy finger, $O(1)$-competitiveness properties?

Since an algorithm with guaranteed "many leaves" may be non-monotone, it is plausible that the answer to Problem 40 is negative. A positive answer would, for instance, also imply results about PathBalance, since PathBalance fulfills the many leaves property. (For the sequential property a positive answer would be much less surprising.) To settle the balance (i.e. logarithmic average cost) part of Problem 40, it would be sufficient to exhibit a long sequence of "search path → after-tree" transformations in a tree, in which the created subtrees have a linear number of leaves, and the average length of the paths is superlogarithmic. We find this a natural combinatorial question, regardless of its implications to the BST model.

A perhaps easier question would include the zigzags, as they appear in Theorem 3.4.

Problem 41. If every transformation creates $\Omega(|P| - z)$ leaves, where z is the number of zigzags in P, is the balance condition guaranteed? How about the other properties?

In § 3.8 we explored the topic of depth-halving. Since the cost of an access depends on the length of the search path, i.e. the depth of the accessed element, it is reasonable to expect that some proof technique directly using global depths should help for studying BST algorithms. The role played by depth-reduction in the efficiency of BST algorithms seems not fully understood (Problems 36 and 37).

Finally, the access lemma is only part of the story. Addressing the following questions would lead to new insight about the BST model. To answer these questions (even partially), we may try to find alternative proofs or to generalize existing proofs of these properties for Splay or other known algorithms.

Problem 42. Characterize the class of strict online algorithms that satisfy the sequential access property.

Problem 43. Characterize the class of strict online algorithms that satisfy the dynamic finger, lazy finger (if any), or $O(1)$-competitiveness (if any) properties.

Chapter 4

Pattern-avoiding access

Akár egy halom hasított fa,
hever egymáson a világ,
szorítja, nyomja, összefogja
egyik dolog a másikát.
— JÓZSEF ATTILA, *Eszmélet* (1934)

In this chapter we study families of access sequences that can be served by BST algorithms much faster than what the logarithmic worst case bound would suggest. The sequences we study are characterized by their *pattern-avoidance* properties. Compared to structures previously studied in the BST literature, pattern-avoidance captures a different aspect of access sequences. Therefore, the results of this chapter can be seen as complementary to the upper bounds discussed in § 2.4.

Pattern-avoidance in sequences has a rich literature both in combinatorics [103, 56, 18, 89] and in computer science [59, § 2.2.1], [22, 61, 102, 85, 19]. More broadly, the avoidance of substructures in combinatorial objects has often been found to make algorithmic problems easier (a prominent example is the theory of forbidden minors in graphs). Our goal is to explore, to what extent the avoidance of patterns in access sequences is relevant and helpful in the BST model.

We mostly consider access sequences $X = (x_1, \ldots, x_m) \in [n]^m$ that are permutations, i.e. $m = n$, and $X \in S_n$. We remark that many of the results can be extended to non-permutation access sequences. Focusing on permutations allows us to avoid some technicalities, and it is also justified by the following result.

Theorem 4.1. Suppose that there is a BST algorithm $\mathcal{A}$ whose cost is at most $c \cdot \mathsf{OPT}(X)$ for all $X \in S_n$. Then there is a BST algorithm $\mathcal{B}$ whose cost is at most $O(c) \cdot \mathsf{OPT}(X)$ for all $X \in [n]^m$, for $m \geq n$.

We omit the proof of Theorem 4.1, and refer the reader to [24, Thm. 2.1]. The result appears to have been folklore for some time (mentioned in [31]), and if both $\mathcal{A}$ and $\mathcal{B}$ are offline algorithms, then it can be shown without further conditions, using an intuitive "perturbation"-argument. However, if we require $\mathcal{A}$ and $\mathcal{B}$ to be online BST algorithms, then, as far as we know, the result has only been shown with some (mild) assumption on $\mathcal{A}$. (The exact assumption in [24] is that $\mathcal{A}$ does not "touch" a node x, if x has not yet been on any of the search paths for an access.) It is an interesting question whether Theorem 4.1 can be shown unconditionally for online algorithms.

4.1 Sequences and patterns

For convenience, we repeat the main definitions related to pattern-avoidance from § 1.3.

Two sequences $(a_1,\dots,a_n)$ and $(b_1,\dots,b_n)$ of the same length are *order-isomorphic*, if their entries have the same relative order, i.e. $a_i < a_j \iff b_i < b_j$ for all i and j. For example, (581) and (231) are order-isomorphic, whereas (1234) and (1324) are not.

Given a sequence $X \in [n]^m$, and a permutation $\pi \in S_k$, we say that X is *π-avoiding*, if it has no subsequence that is order-isomorphic with π, otherwise we say that X *contains* π. For example, the sequence $(4,7,5,2)$ contains 231 and is 123-avoiding. We often call the avoided (or contained) permutation π a *pattern*.

It is helpful to view sequences geometrically, i.e. we interpret a sequence $X = (x_1,\dots,x_m) \in [n]^m$ at the same time as a set of points $X = \{(x_i, i)\} \subset [n] \times [m]$. Let us remark that all point sets discussed in this chapter are in the plane, and consist only of points with integral coordinates. Observe that a permutation X gives rise to a point set X in which no two points are on the same horizontal or vertical line. Pattern-avoidance can be defined for arbitrary point sets as follows.

A point set $X \subseteq [n] \times [m]$ avoids a point set $P \subseteq [k] \times [k]$, if there is no subset $Y \subseteq X$ that is order-isomorphic with P. Two point sets A and B are order-isomorphic, if there is a bijection $f : A \to B$ that preserves the relative positions of points, i.e. x is strictly to the right of y if and only if $f(x)$ is strictly to the right of $f(y)$, and x is strictly above y if and only if $f(x)$ is strictly above $f(y)$, for all $x, y \in A$.

The correspondence between the two views is evident. For instance, a permutation $X \in S_n$ is 231-avoiding exactly if its corresponding point set avoids the point set $\left(\begin{smallmatrix} \bullet & \\ & \bullet \end{smallmatrix}\right)$. See Figure 4.1 for illustration.

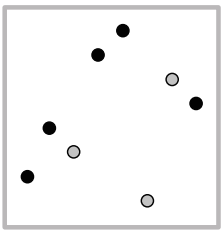

Figure 4.1: Illustration of pattern avoidance. Plot of permutation $X = (6,1,3,2,8,7,4,5)$ containing 213 (highlighted) and avoiding 4321.

As mentioned already, the topic of pattern-avoidance has received enormous attention from the mathematical community. Particularly the enumeration of pattern-avoiding permutations has been extensively studied. Even for very small example patterns π (e.g. π of size 4), the question of determining the number of π-avoiding permutations turns out to be very difficult, and only in very few cases is an exact number (or even a precise asymptotic bound) known.

The following important result was known as the Stanley-Wilf conjecture, formulated independently in the 1980s by Stanley and Wilf, until it was proven in 2004 by Marcus and Tardos [70], building on work by Füredi and Hajnal [45], as well as by Klazar [57].

Theorem 4.2 ([70])**.** There is a function $f(\cdot)$ such that for every permutation $\pi \in S_k$, the number of π-avoiding permutations of size n is at most $\big(f(k)\big)^n$.

From our point of view, the result plays the following role. We would like to show that every access sequence $X \in S_n$ that avoids some fixed pattern $\pi \in S_k$ can be accessed in the BST model with low total cost. Ideally, we would like to show $\mathsf{OPT}(X) \le n \cdot g(k)$, for some function $g(k)$ not depending on n. When such an upper bound on $\mathsf{OPT}(X)$ exists, we say that X has "linear cost". (Linear cost can alternatively be seen as constant average cost per access.) Theorem 4.2 shows that such a result can not be ruled out automatically. If the number of π-avoiding permutations were super-exponentional, then such a result would be impossible, as Theorem 2.20 implies that at most an exponential number of sequences can have linear cost.

In this chapter we prove several statements about the cost of sequences with pattern-avoiding properties. In its full generality, we are, as of yet, unable to settle the question. Let us therefore propose the following as the main open question of this chapter (and perhaps of the entire thesis).

Problem 44. Is there some fixed function $f(\cdot)$ such that for every access sequence $X \in [n]^m$ that avoids an arbitrary fixed pattern $\pi \in S_k$, we have $\mathsf{OPT}(X) \leq m \cdot f(k)$? More strongly, does such an upper bound hold for the cost of some online algorithm?

In § 4.4 we show a relaxed form of the statement, where the upper bound has a mild dependence on n (in the form of the very slowly growing inverse Ackermann function $\alpha(n)$, see e.g. [30, § 21.4]). In § 4.5 and § 4.6 we show that the statement holds for some specific families of patterns π. In most cases we are able to bound not only OPT, but also the cost of the online BST algorithm Greedy (§ 2.7.1).

We also study the independent rectangle bound $\mathsf{MIR}(X)$ in terms of pattern-avoidance properties of the access sequence X (§ 4.7.1). In this case, we can show a linear bound for all access sequences X that avoid an arbitrary fixed pattern π. Recall from § 2.4 that MIR is the strongest known lower bound for OPT, and it is conjectured to asymptotcially match OPT. (In fact, Wilber conjectured that even the weaker $\mathcal{W}^2$ bound matches OPT.) Results of this type give further evidence that the answer to Problem 44 may be affirmative. Indeed, if the cost of Greedy (or any other online algorithm $\mathcal{A}$) on pattern-avoiding input is superlinear, then either Greedy (or $\mathcal{A}$) is *not* dynamically optimal, or the conjecture $\mathsf{MIR}(X) = \Theta\big(\mathsf{OPT}(X)\big)$ must be false.

The family of access sequences that avoid an *arbitrary* fixed pattern is quite general. Special cases that were intensively studied in the BST literature include sequential access (the sequence $S = (1, 2, \ldots, n)$ avoids 21, and it is the only permutation of length n to do so), and traversal access (preorder sequences are exactly the 231-avoiding permutations, see Lemma 1.4). As far as we know, even for the 231-avoiding case, it was not previously shown that an online algorithm can achieve linear cost. (Chaudhuri and Höft [27] show that Splay achieves linear cost on 231-avoiding access sequences, but only with an initial tree that depends on the input, thus the algorithm can not be considered online).

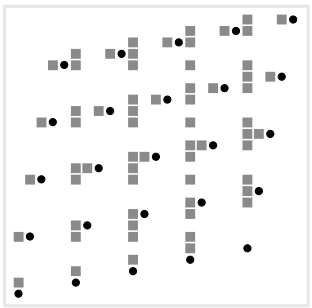

Figure 4.2: Tilted grid permutation of size $n = 25$. It contains all patterns of size up to $\Theta(\sqrt{n})$, and is served with linear cost by Greedy.

It is well-known that a random permutation of size n contains all patterns of size up to $\Theta(\sqrt{n})$. It is also well-known that permutations of size n cannot contain all patterns of size t for $t = \omega(\sqrt{n})$. It can also be shown easily that the bitwise reversal sequence studied in § 2.6 contains all patterns of size up to $\Theta(\sqrt{n})$ (see Theorem 4.3). These observations are consistent with the fact that random permutations, resp. the bitwise reversal sequence, have high cost in the BST model. There are, however, permutations of size n that contain all patterns of size up to $\Theta(\sqrt{n})$, yet have linear cost in the BST model. We illustrate such a permutation (the "tilted grid") and its execution by Greedy in Figure 4.2. The claims about the pattern-containment property and the linear cost (by Greedy) of this family of permutations can be verified by visual inspection, we therefore omit the formal proofs.

Based on the above discussion we conclude that pattern-avoidance is a sufficient condition for the "easiness" of a sequence, but it is not a full characterization. In § 4.8 we discuss the relation between pattern-avoidance and other structural properties of sequences.

Theorem 4.3. The bitwise reversal sequence $R_n \in S_n$ contains every permutation of size t, for some $t = \Omega(\sqrt{n})$.

Proof. Suppose $n = 2^{2k}$, and let $P = (p_0, \ldots, p_{2^k-1}) \in S_{2^k}$ an arbitrary permutation. Let $(x)_k$ denote the value x in binary, padded to length k with zeros on the left. Let $\mathsf{rev}_k(x)$ be the bitwise reversal of $(x)_k$ and let $(x.y)$ denote the bitwise concatenation of x and y.

Denote $R_n = (r_0, \ldots, r_{n-1})$, where $r_i = \mathsf{rev}_{2k}(i)$. We show that R_n contains P. Define the sequence $y_i = \big((i)_k.\mathsf{rev}_k(p_i)\big)$ for $i = 0, \ldots, 2^k - 1$. Observe that y_i is increasing, and with distinct values between 0 and $2^{2k} - 1$. Therefore, r_{y_i} is a subsequence of r_n. Observe that

$$r_{y_i} = \mathsf{rev}_{2k}(y_i) = p_i.\mathsf{rev}_k(i),$$

which is order-isomorphic with P. ∎

In the remainder of this section we define two broad families of pattern-avoiding sequences.

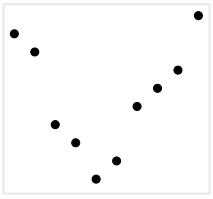

Figure 4.3: Example $\bigvee$-type permutation $X = (5,6,4,3,7,8,9,2,1,10)$.

We call a permutation $X = (x_1, \ldots, x_n) \in S_n$ a $\bigvee$-type permutation, if for all i it holds that $x_i = \max\{x_1, \ldots, x_i\}$ *or* $x_i = \min\{x_1, \ldots, x_i\}$. In words, every entry in a $\bigvee$-type permutation is either larger or smaller than all previous entries. See Figure 4.3 for an illustration.

A $\bigwedge$-type permutation is a permutation whose reverse is a $\bigvee$-type permutation. (By the reverse of a permutation $\pi \in S_n$ we mean the permutation $(\pi_n, \ldots, \pi_1)$.) It is easy to verify that the monotone sequences $(1, \ldots, n)$ and $(n, \ldots, 1)$ are permutations of both $\bigvee$- and $\bigwedge$-type.

In the literature, $\bigvee$-type permutations have been studied in various contexts, and are also known as Gilbreath permutations (or Gilbreath shuffles). Diaconis and Graham [35, § 5] give several characterizations for them, of which we mention two. We also state two other characterizations. Verifying the equivalences between the definitions is left as an easy exercise.

Lemma 4.4. Let $X = (x_1, \ldots, x_n) \in S_n$. The following are equivalent.

(i) X is a $\bigvee$-type permutation,

(ii) [35] for all j, the values in $\{x_1, \ldots, x_j\}$ are consecutive,

(iii) [35] for all j, the values in $\{x_1, \ldots, x_j\}$ are distinct modulo j,

(iv) X avoids both 132 and 312,

(v) X is the reverse of a preorder-sequence of a path (i.e. a BST where every non-leaf node has exactly one child).

The number of $\bigvee$-type (as well as the number of $\bigwedge$-type) permutations of size n is 2^{n-1}. To see this, observe that a $\bigvee$-type permutation is uniquely encoded by a binary string in which the ith digit tells us whether the $(i+1)$th entry of the permutation is a maximum or a minimum of the entries so far. The value of the first entry is uniquely determined by the number of minima and maxima in the permutation (i.e. the number of 0s and 1s in the encoding).

Permutations of $\bigvee$- and $\bigwedge$-type have rather simple structure. We use them next to define a more intricate family of permutations.

We call $X \in S_n$ a $\bigvee_k$*-avoiding permutation*, if there is *an arbitrary* $\bigvee$-type permutation $\pi \in S_k$ such that X is π-avoiding. We define $\bigwedge_k$-avoiding permutations analogously.

As a simple example, observe that preorder sequences are 231-avoiding, and thus they are $\bigvee_3$-avoiding. As another example consider $(23\dots k1)$-avoding permutations, which have been studied in the context of sorting with multiple stacks (see e.g. [17]). Since $(23\dots k1)$ is a $\bigvee$-type permutation of size k, a permutation that avoids it is $\bigvee_k$-avoiding.

As an even simpler example consider $(12\dots k)$-avoiding and $(k\dots 1)$-avoiding permutations, which we call $(k-1)$-*decreasing*, respectively $(k-1)$-*increasing*. As a very special case, $S=(1,2,\dots,n)$ is 1-increasing. By the previous discussion, a $(k-1)$-increasing or $(k-1)$-decreasing permutation is both $\bigvee_k$-avoiding and $\bigwedge_k$-avoiding.

The following lemma states the folklore result that k-increasing and k-decreasing permutations have a natural decomposition. We state this observation only for the increasing case, as the other one is symmetric.

Lemma 4.5. Let $X=(x_1,\dots,x_n)\in S_n$. The following are equivalent:
(i) X is $(k\dots 1)$-avoiding,
(ii) X can be partitioned into pairwise disjoint increasing subsequences $Y_1,Y_2,\dots,Y_{k-1}$.

Proof. (ii) $\Longrightarrow$ (i):
Let $Y_1,\dots,Y_{k-1}$ be pairwise disjoint, increasing subsequences of X, and suppose there exists a subsequence X' of length k of X, order-isomorphic to the pattern $(k\dots 1)$. Since X' is decreasing, no two elements of X' can be in the same subsequence Y_i. This is a contradiction, since we have only $k-1$ subsequences Y_i. Therefore, such an X' cannot exist, and X is $(k\dots 1)$-avoiding.

(i) $\Longrightarrow$ (ii):
Assume that X is $(k\dots 1)$-avoiding. We construct the decomposition of X into increasing subsequences $Y_1,\dots,Y_{k-1}$ as follows. To simplify the argument, we refer to X simultaneously as a set of points $\{(x_i,i)\}$. Let Y_1 be the "wing", i.e. the points of X that form an empty rectangle with the top left corner $(0,n+1)$. (By this definition, we have $x_1\in Y_1$.) Clearly, Y_1 forms an increasing subsequence of X. We remove the elements of Y_1 from X and similarly find Y_2 as the wing of the remaining points. We repeat the process, thereby finding $Y_1,Y_2,\dots$. We claim that we run out of points before reaching Y_k, thus constructing a decomposition of X as required.

Suppose otherwise that we have reached Y_k, and consider an arbitrary point $x_{i_k}\in Y_k$. Since x_{i_k} was not chosen in Y_{k-1}, the rectangle with corners (x_{i_k},i_k) and the top-left corner $(0,n+1)$ contains some point from Y_{k-1}. Pick such a point, and denote it $x_{i_{k-1}}$. Observe that $x_{i_{k-1}}<x_{i_k}$, and $i_{k-1}>i_k$. Since $x_{i_{k-1}}$ was not chosen in Y_{k-2}, we can similarly pick a suitable $x_{i_{k-2}}$ in Y_{k-2}. Continuing in this fashion, we construct the subsequence $(x_{i_1},\dots,x_{i_k})$ of X that forms the forbidden pattern $(k\dots 1)$, a contradiction. ∎

A natural generalization of k-increasing, respectively k-decreasing permutations are k-monotone permutations, i.e. those that can be decomposed into k disjoint monotone subsequences (increasing and decreasing subsequences arbitrarily mixed). Such permutations have been studied in the context of sorting (e.g. [66, 11]) and have been called "shuffled monotone sequences". The family of k-monotone permutations seems not to have a simple avoided-pattern characterization, and such permutations are, in fact, NP-hard to recognize [65].

We have seen that $\bigvee$-type and $\bigvee$-avoiding permutations generalize sequential and traversal (a.k.a. preorder) sequences. Next we generalize these sequences in a different way, and describe yet another decomposition of permutations.

Given a permutation $X = (x_1, \ldots, x_n) \in S_n$ we call an interval $[a, b] \subseteq [n]$ a *block* of X, if, for some c we have

$$\{x_a, x_{a+1}, \ldots, x_b\} = \{c, c+1, \ldots, c+b-a\}.$$

In words, a block is a contiguous interval that is mapped to a contiguous interval.

Observe, that every permutation trivially has a block of size n, and n blocks of size 1. A permutation $X \in S_n$ is *decomposable* if it has a block of size strictly betweeen 1 and n. Otherwise, we say that it is *simple*. See Figure 4.4 for an illustration. (We refer to Brignall [21] for a survey of this well-studied concept. The decomposition described here also appears in the literature as *substitution-decomposition*.)

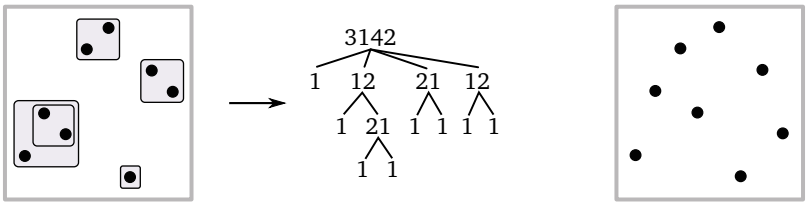

Figure 4.4: (left) Permutation $(6,1,3,2,8,7,4,5)$ and its block-decomposition tree. Permutation (3142) at the root is obtained by contracting the four top level blocks into points. (right) Simple permutation $(6,1,8,4,2,7,3,5)$.

We say that X is decomposable into k blocks if there exist disjoint $[a_1, b_1], \ldots, [a_k, b_k]$ such that each $[a_i, b_i]$ is a block of X and $\bigcup_i [a_i, b_i] = [n]$. To each block $[a_i, b_i]$ we associate a permutation that is order-isomorphic to the sequence of entries $\{x_{a_i}, x_{a_i+1}, \ldots, x_{b_i}\}$ in the block. We call X *recursively d-decomposable*, if it is decomposable into at most d blocks, such that the permutations associated to the blocks are either of length one or themselves recursively d-decomposable. For simplicity, we drop the term "recursive" and refer to such permutations simply as d-decomposable.

The process of recursively decomposing a permutation gives rise to a *block-decomposition tree*, illustrated in Figure 4.4. The nodes of the block-decomposition tree are permutations (of size at most d) that describe the relative positions of the blocks at the same level. More precisely such a permutation is order-isomorphic to the sequence formed by taking an arbitrary representative element from each block of a decomposition, e.g. in the above example we may take the sequence $(x_{a_1}, x_{a_2}, \ldots, x_{a_k})$.

Let us observe that preorder sequences are 2-decomposable. This will imply that $\wedge$- and $\vee$-type sequences are also 2-decomposable, since the first is a preorder sequence of a special tree (Lemma 4.4), and the second is the reverse of the first (reversing a permutation clearly preserves its block-decomposition).

Given a preorder sequence $X = (x_1, \ldots, x_n)$, let k be the maximum index such that $x_2, \ldots, x_k$ are all smaller than x_1. Since $x_2, \ldots, x_k$ are the nodes in the left subtree of x_1 in an underlying BST over $[n]$, we have $\{x_1, \ldots, x_k\} = \{1, \ldots, x_1\}$. By a similar argument for the right subtree of x_1, we have $\{x_{k+1}, \ldots, x_n\} = \{x_1 + 1, \ldots, n\}$. Thus we obtain two blocks of X, and we can continue the same process recursively. See Figure 4.5 for illustration.

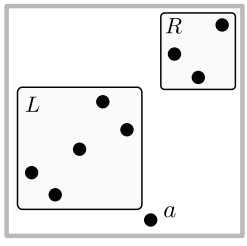

Figure 4.5: Preorder sequence with left (right) subtree of root a highlighted as L (R). Observe that point a together with L can form a single block.

The entire family of 2-decomposable permutations is known as *separable permutations*. Separable permutations are well-studied in the literature (see e.g. [19]). They have a number of equivelent characterizations, for instance, they are exactly the permutations avoiding both 2413 and 3142 (which are the only simple permutations of size 4). The

number of separable permutations of length n is the nth Schröder number [107, 94].

The following statement relates pattern-avoidance and decomposability of permutations.

Lemma 4.6. Let $P \in S_n$, and let k be an integer. The following are equivalent.

(i) P is k-decomposable,

(ii) P avoids all simple permutations of size at least $k+1$,

(iii) P avoids all simple permutations of size $k+1$ and $k+2$.

Proof. (ii) $\implies$ (iii) is obvious.

(iii) $\implies$ (ii) follows from the result of Schmerl and Trotter [86] that every simple permutation of length n contains a simple permutation of length $n-1$ or $n-2$, and the simple observation that if P avoids Q, then P also avoids all permutations containing Q.

(ii) $\implies$ (i) follows from the observation that a permutation P contains all permutations associated to nodes of its block-decomposition tree. Further, it is known [21] that every permutation P has a block-decomposition tree $\mathcal{T}$ in which all nodes are simple permutations. If P contains no simple permutation of size $k+1$ or more, it must have a block-decomposition tree in which all nodes are simple permutations of size at most k, it is therefore, k-decomposable.

(i) $\implies$ (ii): we show the contrapositive $\neg$(ii) $\implies$ $\neg$(i). Indeed, if P contains a simple permutation Q of size at least $k+1$, then any k-decomposition of P would induce a k-decomposition of Q, contradicting the fact that Q is simple. ∎

Since there are no simple permutations of size 3, from Lemma 4.6 it follows that 3-decomposable permutations are exactly the same as 2-decomposable (i.e. separable) permutations.

We remark that the $\bigvee_k$-avoiding and the k-decomposable properties capture very different aspects of permutations. It is easy to construct permutations of length n that are 2-decomposable but not $\bigvee_k$-avoiding for any $k = o(n)$, as well as permutations that are 2-increasing, but simple (i.e. not k-decomposable for any $k < n$). We omit the details.

In closing, we revisit Lemma 4.5 that characterizes permutations that are decomposable into a small number of increasing (or decreasing) subsequences. Can we say something about permutations that are decomposable into subsequences of a more general type? Before making an observation in this direction, let us define the tensor product between two permutations as follows.

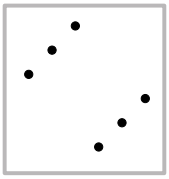

Figure 4.6: Tensor product. (21) ⊗ (123) = (456123)

Given $X = (x_1, \ldots, x_n) \in S_n$ and $P = (p_1, \ldots, p_k) \in S_k$, we denote as $Y = X \otimes P$ the permutation Y of size $n \cdot k$, order-isomorphic with

$$\begin{aligned} Y' = \big(&x_1 \cdot k + p_1,\ x_1 \cdot k + p_2,\ \ldots,\ x_1 \cdot k + p_k,\\ &x_2 \cdot k + p_1,\ \ldots,\ x_2 \cdot k + p_k,\\ &\ldots,\\ &x_n \cdot k + p_1,\ \ldots,\ x_n \cdot k + p_k\big). \end{aligned}$$

In words, we obtain $X \otimes P$ by replacing every entry of X with a block that contains a sequence order-isomorphic to P. See Figure 4.6 for illustration.

Lemma 4.7. Suppose $X \in S_n$ can be partitioned into pairwise disjoint subsequences (not necessarily contiguous) $Y_1, \ldots, Y_k$, such that for all i, the sequence Y_i is π_i-avoiding, for some pattern π_i. Then X avoids $\pi_1 \otimes \pi_2 \otimes \cdots \otimes \pi_k$.

Proof. We prove the following claim from which the original statement follows by induction. If X can be partitioned into disjoint subsequences A and B, such that A is π'-avoiding and B is π''-avoiding, then X is $\pi' \otimes \pi''$-avoiding.

If $\pi' \otimes \pi''$ is longer than X, then we are done. Suppose otherwise, that X contains a subsequence P isomorphic to $\pi' \otimes \pi''$. Partition P into subsequences $P_1, P_2, \ldots$, each order-isomorphic to π''. Notice that A must contain an entry from each P_i, otherwise B would contain a subsequence isomorphic to π''. By the definition of "$\otimes$", these entries form a subsequence isomorphic to π', contradicting that A is π'-avoiding. ∎

4.2 Tools

4.2.1 Hidden elements

In the remainder of this chapter we analyze $\mathsf{OPT}(X)$ for sequences X that avoid some pattern. We mostly do this by analyzing $\text{cost}_{GG}(X)$, the cost of GeometricGreedy on X. Recall that given a point set X, GeometricGreedy outputs a set $Y \supseteq X$ that contains no unsatisfied pairs of points. We also analyze Greedy$^{\boxslash}$ and Greedy$^{\boxbslash}$, whose outputs are, in general, not satisfied supersets, but nevertheless, closely related to OPT. (See § 2.7 for definitions.)

Recall that the cost of these algorithms is the cardinality of their output. When we argue about the output of Greedy, Greedy$^{\boxslash}$ and Greedy$^{\boxbslash}$, a useful property that is often used is that certain elements $a \in [n]$ become "hidden" during the execution, with respect to some interval of $[n]$. This means that as long as no key is accessed in the hidden interval of a, node a is not touched. (In geometric view this means that no point with x-coordinate equal to a is output.) In the following we describe this concept more formally and list some cases when elements become hidden during the execution of Greedy, Greedy$^{\boxslash}$ and Greedy$^{\boxbslash}$. See Figure 4.7 for an illustration.

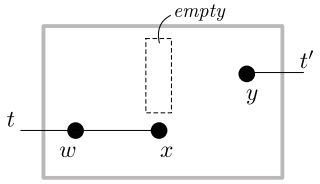

Figure 4.7: Hidden element in geometric view. After time t, element $x \in [n]$ is hidden in $(w, n]$ for Greedy, and in $(w, x]$ for Greedy$^{\boxbslash}$. After time t', element x is hidden in (w, y) for Greedy: Any access outside of (w, y) will not touch x.

Consider an arbitrary algorithm $\mathcal{A}$ that processes a point set $X \subseteq [n] \times [m]$, and outputs a point set $Y \supseteq X$, such that $Y \subseteq [n] \times [m]$. Call points in X *access points*, and let us say that x is *touched* by $\mathcal{A}$ at time t, if $(x, t) \in Y$.

Definition 4.8. For an algorithm $\mathcal{A}$, an element $x \in [n]$ is *hidden* in the interval $[w, y] \subseteq [n]$ after t, if, given that there is no access point $p \in [w, y] \times (t, t']$ for some $t' > t$, then x will not be touched by $\mathcal{A}$ at any time in the interval $(t, t']$.

In the following, we denote by $\tau(x, t)$ the last time at or before t when x is touched, i.e.

$$\tau(x, t) = \max\{q : q \leq t \text{ and } (x, q) \in Y\}.$$

Lemma 4.9. Let $X \subseteq [n] \times [m]$, and let $x \in [n]$ be some element.

(i) If there is an element $w < x$ where $\tau(w, t) \geq \tau(x, t)$, then x is hidden in $(w, n]$ and $(w, x]$ after t for Greedy and Greedy$^{\boxbslash}$ respectively.

(ii) If there is an element $y > x$ where $\tau(y, t) \geq \tau(x, t)$, then x is hidden in $[1, y)$ and $[x, y)$ after t for Greedy and Greedy$^{\boxslash}$ respectively.

(iii) If there are elements w, y where $w < x < y$, and $\tau(w, t), \tau(y, t) \geq \tau(x, t)$, then x is hidden in (w, y) after t for Greedy.

Proof. (i) Consider any $t' > t$. For the case of Greedy, assume that there is no access point in $(w, n] \times (t, t']$. Suppose x is touched in the time interval $(t, t']$, and let (p, t_p) be the first access point in this time interval that causes the touching of x. Then $p \in [1, w]$, and $t_p \in (t, t']$. As x is not touched in the time interval (t, t_p) by the choice of p, we have that $\tau(x, t_p - 1) = \tau(x, t)$. If $\tau(w, t) \geq \tau(x, t)$, then the rectangle with corners (p, t_p), and $(x, \tau(x, t_p - 1))$ contains the point $(w, \tau(w, t))$, and thus it is satisfied before accessing p. Therefore, the accessing of p via Greedy does not touch x, a contradiction. It follows that x is hidden in $(w, n]$ for Greedy after t.

For the case of Greedy$^{\boxtimes}$, assume that there is no access point in $(w, x] \times (t, t']$. Suppose x is touched in the time interval $(t, t']$, and let (p, t_p) be the first access point in this time interval that causes the touching of x. Then $p \in [1, w]$, and $t_p \in (t, t']$. (The case $p \in (x, n]$ is not possible for Greedy$^{\boxtimes}$, since $p < x$ must hold.) The remainder of the argument is the same as for Greedy.

(ii) The argument is analogous to (i).

(iii) Consider any $t' > t$. Assume that there is no access point in $(w, y) \times (t, t']$. Suppose x is touched in the time interval $(t, t']$, and let (p, t_p) be the first access point in this time interval that causes the touching of x. There are two cases. If $p \in [1, w]$, we use the argument of (i). If $p \in [y, n]$, we use the argument of (ii). ∎

4.2.2 Forbidden submatrix theory

Suppose that $X \subseteq [n] \times [n]$ and $P \subseteq [k] \times [k]$ are sets of planar points with integer coordinates, and X is P-avoiding. How many points can X maximally contain? Forbidden submatrix theory studies questions of this type, usually formulated in the language of 0/1-matrices, or in terms of subgraphs of bipartite graphs, instead of planar point sets. The cardinality of X is clearly at most $n \cdot m$, but depending on the structure of P, in many cases, much stronger bounds can be shown.

Perhaps the first question of this kind was the 1951 problem of Zarankiewicz, which asks for the cardinality of X in the case where the avoided pattern P is an a-by-b "rectangular block" of points. (Alternatively, we can ask for the maximum number of edges in a bipartite graph that avoids a certain bipartite subgraph.) Many variants and generalizations of this problem have been studied, we refer to Füredi [44], and Bienstock and Győri [15]. (Problems of a similar flavour have been asked even earlier in graph theory, going back to the theorem of Mantel from 1907, concerning the maximum number of edges in triangle-free graphs.)

In its most general form, the problem of Zarankiewicz is not fully resolved, but for many special cases tight or almost tight bounds are known. We give a simple proof for the basic $a = b = 2$ case.

Lemma 4.10. Let $X \subseteq [n] \times [n]$, and $P = \{(0,0), (0,1), (1,0), (1,1)\}$. If X avoids P, then $|X| = \Theta(n\sqrt{n})$.

Proof. Let Φ_i denote the number of unordered pairs $x, y \in [n]$ such that for some $j \leq i$ we have $(x, j), (y, j) \in X$. Clearly, for all i we have $\Phi_i \leq \binom{n}{2}$, and we set $\Phi_0 = 0$. Let n_k denote the number of points in X with y-coordinate k. We claim that $\Phi_k - \Phi_{k-1} = \binom{n_k}{2}$ for all k. This is because none of the $\binom{n_k}{2}$ pairs with y-coordinate k can appear with y-coordinate less than k, since this would create the forbidden pattern P. We have thus $\binom{n}{2} \geq \Phi_n - \Phi_0 = \sum_{k=1}^{n} \binom{n_k}{2}$. By convexity, the quantity $|X| = \sum_k n_k$ is maximized if the n_ks are equal, and $\binom{n_k}{2} = \binom{n}{2}/n$. It follows that $n_k \leq \sqrt{n} + 1$, and $|X| = O(n\sqrt{n})$. A simple construction shows that this bound is asymptotically tight. ∎

We state further results from the literature that we use in the remainder of the chapter. For details and proofs of these statements (in the language of 0/1-matrices), we refer to Füredi, Hajnal [45, Cor. 2.7], Marcus, Tardos [70], Fox [41], Nivasch [78], and Pettie [81]. We call a point set $X \subset [n] \times [n]$ a *permutation point set*, if there is exactly one point on every horizontal or vertical line with integer coordinates. Relaxing this condition slightly, we call a point set *light* if it has a single point on every vertical line, but possibly multiple points on horizontal lines.

Lemma 4.11. Let $X \subseteq [n] \times [n]$ and $P \subseteq [k] \times [k]$ such that X avoids P.

(i) [45, Cor. 2.7] If $P = \begin{pmatrix} \bullet & \\ & \bullet \end{pmatrix}$, then $|X| \leq 4n$.

(ii) [70, 41] If P is a permutation point set, then $|X| \leq n \cdot 2^{O(k)}$.

(iii) [78, 81] If P is a light point set, then $|X| \leq n \cdot 2^{\alpha(n)^{O(k)}}$.

Our use of this theory is inspired by work of Pettie [80, 82], who applied forbidden pattern techniques to prove bounds about data structures (in particular he reproves the sequential access theorem for Splay, and proves that Splay achieves $O(\alpha^*(n))$ average cost per operation on deque-sequences – sequences of insert and delete operations at minima and maxima of the currently stored key-set). There are marked differences in our use of the techniques compared with the work of Pettie. In particular, we look for patterns directly in the execution trace of a BST algorithm, without any additional encoding step. It appears that the direct use of forbidden submatrix techniques is particularly suited for the study of Greedy. Furthermore, instead of using fixed forbidden patterns, we make use of patterns that depend on the input. For further applications of the technique we refer to [23].

4.3 Sequential and traversal access

As a warmup, in this section we present a direct proof for the linear cost of Greedy on traversal sequences. The proof is a simple application of the forbidden submatrix technique. For an alternative proof (not using forbidden submatrices) we refer to [24, § A].

Theorem 4.12. Let $X \in S_n$ be a 231-avoiding permutation. Then the cost of GreedyFuture on X with the canonical initial tree is at most $4n$.

Proof. We view X as a permutation point set. Then X is $\begin{pmatrix} \bullet & \\ & \bullet \end{pmatrix}$-avoiding. We study the output Y of GeometricGreedy on X with no initial tree, which by Theorem 2.28 equals the execution trace of GreedyFuture on X with the canonical initial tree. We claim that Y is $\begin{pmatrix} \bullet & \\ & \bullet \end{pmatrix}$-avoiding. Then by Lemma 4.11(i), we have $|Y| \leq 4n$, and we are done.

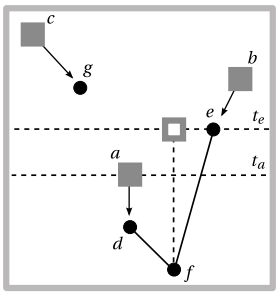

Figure 4.8: Proof of Theorem 4.12.

Let T be the BST whose preorder sequence is X. Suppose for contradiction that Y contains $\begin{pmatrix} \bullet & \\ & \bullet \end{pmatrix}$, and thus there exist $(a, t_a), (b, t_b), (c, t_b) \in Y$, such that $t_a < t_b < t_c$ and $c < a < b$. Observe that there exists an access point $(d, t_d) \in X$, such that $d = a$ and $t_d \leq t_a$, for otherwise a would not be touched at time t_a by Greedy. (Possibly (a, t_a) itself is this access point.) Since b is hidden in $(a, n]$ after t_a, for b to be touched at time t_b, there must be an access point in $(a, n] \times (t_a, t_b]$. Let (e, t_e) be the lowest such access point in X (possibly (b, t_b) itself). Let f be the first left ancestor of e in T, and suppose it is accessed at time t_f (i.e. $(f, t_f) \in X$). (Observe that $d \leq f < e$ holds, since $\mathrm{lca}(d, e)$ is an ancestor of f, and also $t_f < t_e$.) Then, since $f \in \mathsf{stair}_{t_e}(e)$ (since there is no touched or accessed point in $(f, e] \times [1, t_e)$), we

have $(f, t_e) \in Y$. But then, since c is hidden in $[1, f)$ after t_e, for c to be touched at time t_c, there must be an access point in $[1, f) \times (t_e, t_c]$. Let $(g, t_g) \in X$ be such an access point (possibly (c, t_c) itself is this point). We have $(f, t_f), (e, t_e), (g, t_g) \in X$ order-isomorphic to $\left(\begin{smallmatrix} \bullet & & \\ & & \bullet \\ & \bullet & \end{smallmatrix}\right)$, contradicting that X is 231-avoiding. (See Figure 4.8.) ∎

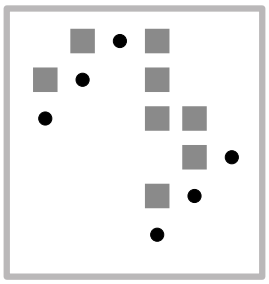

Figure 4.9: Counterexample to naïve approach. Input $X = (4,5,6,1,2,3)$ avoids (321), but output contains (321).

A corollary of Theorem 4.12 (using the equivalences discussed in § 2.7.1) is that the cost of the online BST algorithm OnlineGreedy with a fixed initial tree (or alternatively, with an initial preprocessing that is independent of the access sequence) is $O(n)$.

Curiously, for GeometricGreedy in the case of an access sequence X that is the preorder sequence of a tree T, having no initial tree in geometric view has the exact same effect as having T as an initial tree. (We refer to Theorem 2.29 and Figure 2.11 in § 2.7.1 for discussion of initial trees in geometric view.) The claim follows from the observation that the the canonical tree of GreedyFuture is in this case exactly the tree T that generates the sequence X. (We refer to 2.5.3 for the definition of the canonical tree.)

Looking at the proof of Theorem 4.12 it is tempting to conjecture that if X avoids P, then the GeometricGreedy output avoids P, as we have seen in the proof of Theorem 4.12 for the case $P = \left(\begin{smallmatrix} \bullet & & \\ & & \bullet \\ & \bullet & \end{smallmatrix}\right)$.

Unfortunately, in general, this is not true. Figure 4.9 shows the counterexample $P = \left(\begin{smallmatrix} \bullet & & \\ & \bullet & \\ & & \bullet \end{smallmatrix}\right)$.

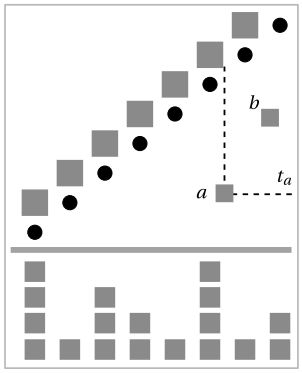

Figure 4.10: Proof of Theorem 4.13. Initial tree shown below line.

Since traversal sequences of length n include $(1, \ldots, n)$ as a special case, Theorem 4.12 also implies the linear cost of Greedy for sequential access. However, in the case of sequential access we can strengthen the result to hold with arbitrary initial tree.

Theorem 4.13. Let $X = (1, \ldots, n)$. Then the cost of GreedyFuture on X with arbitrary initial tree is at most $5n$.

Proof. We split the output $Y \in [n] \times [n]$ into two parts: The points strictly above the diagonal between $(1,1)$ and (n,n) and the points at or below the diagonal. All access points are on the diagonal. By a simple inspection of the Greedy execution on X it is clear that above the diagonal there are exactly $n-1$ points in Y. (See Figure 4.10.) We argue that the points at or below the diagonal avoid $\left(\begin{smallmatrix} \bullet & & \\ & & \bullet \\ & \bullet & \end{smallmatrix}\right)$, and thus, by Lemma 4.11(i), their number is at most $4n$.

Suppose there are $(a, t_a), (b, t_b), (c, t_b) \in Y$, at or below the diagonal, such that $t_a < t_b < t_c$ and $c < a < b$. If (a, t_b) or (b, t_b) were access points, then (c, t_c) would have to be above the diagonal to form the pattern. Therefore, assume that $(a, t_b), (b, t_b) \in Y \setminus X$. Observe that b is hidden after t_a in interval $(a, n]$, therefore if b is touched at time t_b, then there must be an access point in $(a, n] \times (t_a, t_b]$. But then, (c, t_c) is strictly above the diagonal, a contradiction. ∎

The argument in Theorem 4.13 can be extended without much effort to $\vee$-type permutations (with initial tree). Surprisingly, showing the same for $\wedge$-type permutations seems much more difficult. Observe that by Lemma 4.4 this is a very special case of the question whether Greedy satisfies the traversal access condition with an *arbitrary* initial tree (see Problem 22, and the discussion in [24]).

Problem 45. Is the cost of Greedy, Splay, or some other online BST algorithm $O(n)$ on all $\bigwedge$-type permutations of size n with every initial tree?

4.4 The main result for Greedy

In this section we show the following result.

Theorem 4.14. The cost of Greedy with arbitrary initial tree, for every access sequence $X \in S_n$ that avoids an arbitrary pattern $\pi \in S_k$ is $n \cdot 2^{\alpha(n)^{O(k)}}$.

The result implies that Greedy with arbitrary initial tree has cost $n \cdot 2^{\alpha(n)^{O(1)}}$ for all traversal sequences of length n, *almost* matching the conjectured linear bound.

It remains open whether the bound in Theorem 4.14 can be strengthened. We conjecture that it can be improved to a linear bound (Problem 44). Nonetheless, the existing bound is already stronger than what is known for other online BST algorithms such as Splay. We ask therefore the following open question.

Problem 46. Find an upper bound on the cost of Splay for every access sequence $X \in S_n$ that avoids an arbitrary pattern $\pi \in S_k$.

Before proving Theorem 4.14 we introduce a number of simple concepts.

Let $Y \subseteq [n] \times [m]$ be a set of points that *contains* a point set $P \subseteq [k] \times [k]$. A *box* is a set of type $[a,b] \times [c,d]$. We call a box B a *bounding box* of P in Y if $Y \cap B$ contains P, but $Y \cap B'$ avoids P for all boxes $B' \subset B$. Observe that the bounding box of P in Y is not necessarily unique.

We consider arbitrary geometric algorithms that read an input point set $X \subseteq [n] \times [m]$ and output a point set $Y \subseteq [n] \times [m]$ such that $Y \supseteq X$. We say that an algorithm $\mathcal{A}$ is *input-revealing* if, for some constant k there is a point set $G \subseteq [k] \times [k]$ such that whenever Y contains G, every bounding box B of G in Y contains at least one point of X. When such a G exists for an algorithm, we say that G is an *input-revealing gadget*.

We extend now the tensor product notation described earlier to arbitrary point sets. Given a point set $Y \subseteq [n] \times [m]$, we can associate to it a 0/1-matrix M_Y in the obvious way: $M_Y(i,j) = 1$ if $(i,j) \in Y$ and 0 otherwise. The mapping is clearly bijective, so we will refer to a point set and its associated matrix interchangeably.

The *tensor product* between two 0/1-matrices M and G, denoted $M \otimes G$ is obtained by replacing every one-entry of M by a copy of G, and every zero-entry of M by an all-zero matrix equal in size with G. The tensor product between two point sets Y and G, denoted $Y \otimes G$ is the point set corresponding to the matrix $M_Y \otimes G_Y$.

The following simple observation is the key ingredient of the result.

Lemma 4.15. Suppose algorithm $\mathcal{A}$ produces output Y from input X. If G is an input-revealing gadget for $\mathcal{A}$ and X avoids P, then Y avoids $P \otimes G$.

Proof. Suppose Y contains $P \otimes G$, and consider a point set $PG \subseteq Y$ that is order-isomorphic to $P \otimes G$. Consider all bounding boxes $B_1, B_2, \ldots$ of G in PG. By the definition of the input-revealing gadget, each B_i contains a point of X. Denote $X' = \bigcup_i (B_i \cap X)$. Clearly $X' \subseteq X$, and X' contains P, therefore X contains P, a contradiction. (See Figure 4.11.) ∎

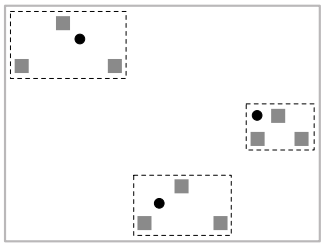

Figure 4.11: Proof of Lemma 4.15 and 4.16. Output Y contains $\left(\begin{smallmatrix}\bullet & \\ & \bullet\end{smallmatrix}\right) \otimes (\begin{smallmatrix}. & \bullet & .\end{smallmatrix})$, where $(\begin{smallmatrix}. & \bullet & .\end{smallmatrix})$ is an input-revealing gadget, therefore input X contains $\left(\begin{smallmatrix}\bullet & \\ & \bullet\end{smallmatrix}\right)$.

The following property of Greedy holds.

Lemma 4.16. $G = (\begin{smallmatrix}. & \bullet & .\end{smallmatrix})$ is an input-revealing gadget for Greedy with arbitrary initial tree.

Proof. Let Y be the output of Greedy for input X, and let $(a,t_a),(b,t_b),(c,t_c) \in Y$ order-isomorphic to G, such that $a < b < c$. Let B be the box with corners $(a,t_a),(c,t_b)$. As $\tau(a,t_a),\tau(c,t_a) \geq \tau(b,t_a)$, element b is hidden in (a,c) after t_a. If there is no access point in B, then (b,t_b) cannot be touched, a contradiction. (See Figure 4.11.) ∎

We are ready to prove the main theorem.

Proof of Theorem 4.14. Observe that for the gadget G of Lemma 4.16, if P is a permutation point set of size k, then $P \otimes G$ is a light point set of size $3k$. By Lemma 4.15 and 4.16 the output of Greedy for X avoids $P \otimes G$. An application of Lemma 4.11(iii) finishes the proof. ∎

To improve the bound of Theorem 4.14 from quasilinear to linear, it would be sufficient to find an input-revealing gadget G that is a permutation point set. (Since, in that case, for permutation point set P, the point set $P \otimes G$ would also be a permutation point set, and the stronger bound of Lemma 4.11(ii) could be applied.) Unfortunately, as a construction of Thatchaphol Saranurak shows, such a gadget does not exist for Greedy with some initial tree. We refer to [24, § G] for details.

4.4.1 An aside: adaptive sorting

One of the very few algorithmic problems that has been studied even more extensively than the BST problem is sorting. It is well-known that in the comparison-only model, sorting a sequence of n elements requires $\Omega(n \log n)$ comparisons in the worst case. Furthermore, this bound is matched by several classical algorithms such as Mergesort, Heapsort, or Quicksort (with proper choice of pivot) [30], [71, § 2].

Given the importance of sorting, significant effort was spent on studying the performance of algorithms on input sequences with a certain amount of *pre-sortedness*, in which case it is possible to beat the $\Theta(n \log n)$ barrier. Various kinds of structures have been proposed to capture pre-sortedness, some of them similar to the BST structures studied in § 2.4. This broad area of research has been called *adaptive sorting*. As an entry point to the vast literature on adaptive sorting, we refer to [77, 39, 66, 69, 75, 11], [60, § 5.3], and references therein.

A remark is in order. Adaptive sorting feels similar to the study of OPT in the BST problem. However, in the case of sorting, true instance-optimality, in a sense similar to dynamic optimality in the BST model, is ill-defined. This is because for every particular sequence X, there is a sorting algorithm tailored to X, that can sort it in linear time (by simply verifying that the input is indeed in the same order as X, and then putting it in the correct order). In fact, a stronger result of Fredman [43] shows that if the input permutation is drawn from an arbitrary subset of S_n, then it can be sorted using a number of comparisons essentially matching the information-theoretic lower bound. However, the result of Fredman does not include the work spent outside of the comparison-model, and since his algorithm explicitly maintains a distribution over the permutations, it is highly impractical.

In light of these, adaptive sorting for restricted *families* of inputs is still an interesting, and practically relevant problem. In the spirit of this chapter, we ask the following question:

Problem 47. Is there a sorting algorithm that can sort in linear time every permutation that avoids an arbitrary fixed pattern?

By Theorem 4.2, such an algorithm can exist, in fact, if we only care for the number of comparisons, the result of Fredman mentioned before answers this question. However, we would like an algorithm that is practical, has low *overall* cost on pattern-avoiding input, and degrades gracefully on arbitrary input, i.e. it is a good general-purpose sorting algorithm. This question was also asked by Arthur [9] in 2007, who gave an adaptive sorting algorithm with running time $O(n \log\log\log n)$ for inputs with certain special avoided patterns.

We observe that the main result of this chapter (Theorem 4.14) addresses Problem 47 and improves the known adaptive sorting results in several ways. Given a BST algorithm that supports insertions, we can sort a sequence X by inserting its entries into an initially empty BST, and reading them out in order in the end. The cost of sorting equals (asymptotically) the total cost of inserts in the BST model. (In fact, if we only care about the number of comparisons, then the cost of sorting seems closely related to the "search-only cost" in the model of Blum et al., § 2.3, since re-arranging the tree is free.) This idea is, of course, not new, but simply a description of the classical Insertion-sort algorithm, with a particular data structure for storing the already sorted elements.

We state without proof that the upper bounds on the cost of Greedy for an access sequence X are also upper bounds on the cost of Greedy for an *insertion sequence* X. For details, we refer to [23]. In fact, the observation holds not just for Greedy, but for a broad class of BST algorithms. (Using Splay instead of OnlineGreedy for sorting is likely a better idea in practice.)

These observations, together with Theorem 4.14 yield a sorting algorithm that can sort every permutation of size n with an avoided pattern of size k in time $n \cdot 2^{\alpha(n)^{O(k)}}$. This algorithm, which we can call "Greedy-sort", has several further attractive features:

(i) It has worst-case $O(n \log n)$ running-time on arbitrary input of size n.
(ii) In contrast to other algorithms proposed in the literature, it does not need to know the avoided pattern at runtime. Pattern-avoidance is only used in the analysis.
(iii) It is "insertion-sort-like", i.e. entries of the input are read one by one, and at time i, we only compare the ith entry with entries previously read.
(iv) It adapts to various other structures besides avoided patterns (like Greedy itself).

The connection to sorting adds another motivation for settling Problem 44. In fact, settling Problem 47 may be easier, since we are free to use structures other than BSTs.

4.5 $\bigvee$-avoiding sequences

In the first part of this section we strengthen the result of Theorem 4.14 from quasilinear to linear, in the special case when the access sequence X is k-increasing or k-decreasing, i.e. if it avoids $\big((k+1)k\dots1\big)$ or $\big(1\dots k(k+1)\big)$.

We show the following result.

Theorem 4.17. The cost of Greedy with arbitrary initial tree, for an access sequence $X \in S_n$ that avoids $\big((k+1)k\dots1\big)$ or $\big(1\dots k(k+1)\big)$ is $n \cdot 2^{O(k^2)}$.

The proof of Theorem 4.17 is similar to the proof of Theorem 4.14 but more involved. As opposed to the earlier proof, in this case we can construct an input-revealing gadget that is a permutation point set. This gadget, however, is no longer a simple constant-sized

construction as in Lemma 4.16. Instead, it depends on the pattern that is avoided in the access sequence. We stress that the gadget is used only for the purpose of analysing the cost of Greedy, the algorithm does not need to know this gadget (or the avoided pattern) during its runtime.

Lemma 4.18. Assuming that the access sequence X avoids $(1\dots k)$, respectively $(k\dots 1)$, the permutation $((k+1)k\dots 1)$, respectively $(1\dots k(k+1))$ is an input-revealing gadget for Greedy with arbitrary initial tree.

Proof. We prove the first case only, the other case is symmetric. Assume therefore that the access sequence X avoids $(1\dots k)$. Let Y be the output of Greedy on X, and suppose that Y contains $((k+1)k\dots 1)$, and let B be a bounding box of Q in Y. Let $Q=(q_0,q_1,q_2,\dots,q_k)$ be the x-coordinates of the points in $Y\cap B$ that form $((k+1)k\dots 1)$, and let $t_0,t_1,\dots,t_k$ be their respective y-coordinates ("times"). We have $q_0>q_1>\dots>q_k$ and $t_0<t_1<\dots<t_k$.

Note that for all $1\le i\le k$, we can assume that there is no point $(q_i,t_i')\in Y$ where $t_{i-1}<t_i'<t_i$, for otherwise we could choose (q_i,t_i') instead of (q_i,t_i). So, for any $t\in[t_{i-1},t_i)$ we have that $\tau(q_{i-1},t)\ge\tau(q_i,t)$ for all $1\le i\le k$.

Suppose for contradiction that there is no access point from X in B. Let ℓ be the x-coordinate of the left margin of B.

By Lemma 4.9(ii), q_1 is hidden in $[1,q_0)$ after t_0. Hence, there must be an access point $(p_1,t_{p_1})\in[1,\ell)\times(t_0,t_1]$, otherwise (q_1,t_1) would not be touched. We choose (p_1,t_{p_1}) such as to maximize p_1. If $t_{p_1}<t_1$, then $\tau(p_1,t_{p_1}),\tau(q_0,t_{p_1})\ge\tau(q_1,t_{p_1})$. So by Lemma 4.9(iii), q_1 is hidden in (p_1,q_0) after t_{p_1} and hence (q_1,t_1) cannot be touched, a contradiction. Thus we have $t_{p_1}=t_1$ and $p_1<\ell$.

Next, we prove by induction for $i=2,\dots,k$ that given the access point $(p_{i-1},t_{p_{i-1}})$ where $t_{p_{i-1}}=t_{i-1}$ and $p_{i-1}<\ell$, there must be an access point (p_i,t_{p_i}) with $t_{p_i}=t_i$ and $p_{i-1}<p_i<\ell$. Again, by Lemma 4.9(iii), there must be an access point $p_i\in(p_{i-1},q_{i-1})\times(t_{i-1},t_i]$, otherwise (q_i,t_i) would not be touched. We choose the point (p_i,t_{p_i}) where p_i is maximized. If $t_{p_i}<t_i$, then $\tau(p_i,t_{p_i}),\tau(q_{i-1},t_{p_i})\ge\tau(q_i,t_{p_i})$. Hence by Lemma 4.9(iii), q_i is hidden in (p_i,q_{i-1}) after t_{p_i} and therefore (q_i,t_i) cannot be touched. Thus, we have $t_{p_i}=t_i$ and $p_{i-1}<p_i<\ell$. We get the points $(p_i,t_{p_i})\in X$, such that $p_1<\dots<p_k<\ell$, and $t_{p_1}<\dots<t_{p_k}$, which contradicts the assumption that X is $(1\dots k)$-avoiding and concludes the proof. (See Figure 4.12.) ∎

Proof of Theorem 4.17. If X avoids $P=((k+1)k\dots 1)$, respectively $P=(1\dots k(k+1))$, then $G=(1\dots(k+1)(k+2))$, resp. $G=((k+2)(k+1)\dots 1)$ is an input-revealing gadget by Lemma 4.18. Observe that $P\otimes G$ is a permutation of size $O(k^2)$. By Lemma 4.15 the output of Greedy for X avoids $P\otimes G$. An application of Lemma 4.11(ii) finishes the proof. ∎

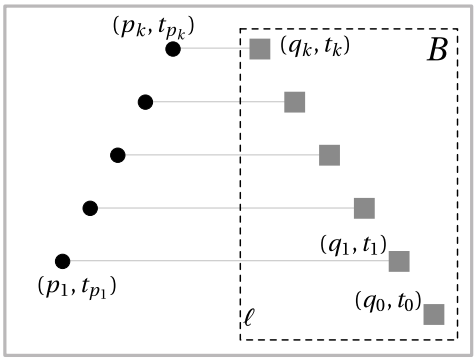

Figure 4.12: Proof of Lemma 4.18.

We remark that recently a linear bound on $\mathsf{OPT}(X)$ for k-increasing (resp. k-decreasing) sequences was shown via a different argument (with a better dependence on k). We briefly sketch the argument (see [26] for details). In § 2.3 we mentioned the extended BST model with multiple pointers. In this model, using the decomposition of k-increasing and k-decreasing sequences (Lemma 4.5) we can serve such sequences in linear time, even if the underlying tree is a static path (each pointer serves one of the monotone subsequences). In fact, in this model we can serve in linear time even the more general family of k-monotone permutations (decomposable into intermixed increasing and decreasing sequences). By the result of Demaine, Iacono, Langerman,

and Özkan [33], serving an access sequence in this powerful k-pointer model can be simulated in the BST model (with a single pointer and root access) at the cost of an $O(k)$ factor slowdown.

We ask whether the results on k-increasing and k-decreasing sequences can be extended to $\bigvee_k$-free sequences. In § 4.7.2 we give some evidence that this may be possible, and we suspect the question to be an accessible intermediate step towards Problem 44.

Problem 48. Is there some function $f(\cdot)$ such that for every $\bigvee_k$-avoiding or $\bigwedge_k$-avoiding sequence $X \in S_n$, it holds that $\mathsf{OPT}(X) \leq n \cdot f(k)$? More strongly, can we bound the cost of Splay or Greedy (with or without initial tree) for such inputs?

4.6 Block-decomposable sequences

In this section we strengthen the result of Theorem 4.14 from quasilinear to linear in another special case, namely for k-decomposable access sequences. We show the following result.

Theorem 4.19. The cost of Greedy with no initial tree, for an access sequence $X \in S_n$ that is k-decomposable is $n \cdot 2^{O(k^2)}$.

The result implies another kind of "almost-traversal" result: it shows that OnlineGreedy with a fixed initial tree has cost $O(n)$ on an arbitrary traversal sequence of length n.

It is open whether the "no initial tree" condition of Theorem 4.19 (i.e. the reliance on a fixed canonical initial tree) can be relaxed, and whether the result can be shown with arbitrary initial tree. This is open even in very special cases of 2-decomposable sequences (Problems 22 and 45). We mention that Thatchaphol Saranurak has recently found a different argument for proving Theorem 4.19, using the Iacono-Langerman [54] result, with an improved dependence on k [26].

The proof of Theorem 4.19 is similar to the proof of Theorem 4.17, but more involved: we construct an input-revealing gadget whose size depends, in this case, on the decomposition parameter k. The gadget, denoted G_k is defined as follows.

Let $G_k = \big(\lfloor (k+1)/2 \rfloor, k, 1, k-1, 2, \ldots\big)$. In words, G_k consists of an initial "middle point", followed by a $\bigwedge$-type permutation alternating between left and right sides of the middle point. For example,

$$G_7 = \left(\begin{array}{ccccccc} & & \bullet & & \bullet & & \\ & \bullet & & & & \bullet & \\ \bullet & & & & & & \bullet \\ & & & \bullet & & & \end{array} \right).$$

Lemma 4.20. Assuming that the access sequence X is k-decomposable, the permutation G_{k+4} is an input-revealing gadget for Greedy with no initial tree.

Then, the result follows.

Proof of Theorem 4.19. If X is k-decomposable, then by Lemma 4.20 the output Y of Greedy on X avoids $P \otimes G_{k+4}$, for all simple permutations P of size at least $k+1$ (using Lemma 4.6). Observe that $P \otimes G$ is a permutation of size $O(k^2)$. An application of Lemma 4.11(ii) finishes the proof. ∎

In the remainder of the section we prove Lemma 4.20. We first prove the following technical lemma. Given a box B, we denote by $B.ymin$, $B.ymax$ the y-coordinates of the lower, resp. upper margin of B, and by $B.xmin$, $B.xmax$ the x-coordinates of the left, resp. right margin of B.

Lemma 4.21. Let Y be the output of Greedy with no initial tree on k-decomposable input X, and let B be a bounding box of G_{k+1} in Y. If there is no access point $p \in B$, then there are k access points (p_i, t_i), for $1 \le i \le k$, such that $B.ymin \le t_1 < \cdots < t_k \le B.ymax$ and $B.xmax < p_i$ for all odd i and $p_i < B.xmin$ for all even i.

Proof. Let (q_i, t'_i) for $0 \le i \le k$ be points in Y such that $t'_i < t'_{i+1}$ for all i and such that the points (q_i, t'_i) form G_{k+1} with bounding box B. Suppose that there is no access point in B.

We prove for $i = 1, \ldots, k$, that there exists an access point $(p_i, t_i) \in (B.xmax, n] \times (t'_{i-1}, t'_i]$ for odd i, and $(p_i, t_i) \in [1, B.xmin) \times (t'_{i-1}, t'_i]$ for even i.

For odd i, by Lemma 4.9(i), q_i is hidden in $(q_{i-1}, n]$ after t'_{i-1}. So there must be an access point $p_i \in (B.xmax, n] \times (t'_{i-1}, t'_i]$, otherwise q_i cannot be touched.

For even i, by Lemma 4.9(ii), q_i is hidden in $[1, q_{i-1})$ after t'_{i-1}. So there must be an access point $p_i \in [1, B.xmin) \times (t'_{i-1}, t'_i]$, otherwise q_i cannot be touched.

Observe that (p_i, t_i) for $1 \le i \le k$ satisfy $B.ymin \le t_1 < \cdots < t_k \le B.ymax$ and $B.xmax < p_i$ for all odd i and $p_i < B.xmin$ for all even i. ∎

Let X be a k-decomposable permutation. Suppose that G_{k+4} appears in the output Y of Greedy, with bounding box B. Suppose for contradiction that B contains no access point. Then, by Lemma 4.21, there exist access points (p_i, t_i) for $1 \le i \le k+3$, satisfying the conditions $B.ymin \le t_1 < \cdots < t_{k+3} \le B.ymax$ and $B.xmax < p_i$ for odd i and $p_i < B.xmin$ for even i.

Let (q_i, t'_i) for $0 \le i \le k+3$ denote the points of Y that form G_{k+4} (where $t'_0 < t'_1 < \cdots < t'_{k+3}$), and let us denote $\mathcal{P} = \{(p_i, t_i) : 1 \le i \le k+3\}$. Let $\mathcal{T}$ be a block decomposition tree of X of arity at most k. We look at each block $P \in \mathcal{T}$ as a minimal rectangular region in the plane that contains all access points in the block.

Let P^* be the *smallest* block in $\mathcal{T}$ that contains two distinct access points $(p_i, t_i), (p_j, t_j) \in \mathcal{P}$ such that i is odd, and j is even. (Observe that p_i is to the right of B, and p_j is to the left of B.)

Observe first that the bounding box of P^* must contain or intersect both vertical sides of B, otherwise it could not contain points on both sides of B. Furthermore, the bottom horizontal side of B must be contained entirely in the bounding box of P^*: If that were not the case, then there would be no accesses below B, and (q_0, t'_0) (the lowest point of G_{k+4}) would not be touched by Greedy (since there is no initial tree). The following is the key observation of the proof.

Lemma 4.22. Let k' be the largest integer such that P^* contains (q_i, t'_i) for $0 \le i \le k'$. Then, $k' < k+1$. In words, P^* contains at most the $k+1$ lowest points of G_{k+4}.

Proof. Let $P_1, \ldots, P_k$ be the decomposition of P^* into k blocks. We claim that each block P_j can contain at most one point from $\mathcal{P}$. First, by the minimality of P^*, none of the blocks P_j can contain two access points from $\mathcal{P}$ from different sides of B. Furthermore, P_j cannot contain two points from $\mathcal{P}$ from the same side of B, because otherwise, assuming w.l.o.g. that P_j contains two points from $\mathcal{P}$ from the left of B, it would follow that there is another access point $(a, t) \in \mathcal{P}$ on the right side of B, outside of P_j, such that $(P_j).ymin < t < (P_j).ymax$, contradicting that P_j is a block.

Furthermore, observe that except for the block containing (p_1, t_1), none of the blocks P_j can overlap with B. Since we have at most k such blocks in the decomposition, and because $t'_i \ge t_i$ for all i, it follows that the top margin of P^* must be below t'_{k+1}. ∎

Since the bounding box of P^* contains (at best) the first $k+1$ points of G_{k+4}, the three remaining points of G_{k+4} need to be touched after the last access in P^*. In the following we show that this is impossible.

Let (L, t_L) be the topmost access point inside P^* to the left of B, such that there is a touched point inside B at time t_L. Let (R, t_R) be the topmost access point inside P^* to the right of B, such that there is a touched point inside B at time t_R. Let L' be the rightmost key inside B touched at time t_L, and let R' be the leftmost key inside B touched at time of t_R.

Lemma 4.23. After time t_L, within the interval $[B.xmin, L']$, only L' can be touched. Similarly, after time t_R, within the interval $[R', B.xmax]$, only R' can be touched.

Proof. Let L'' be the rightmost touched key in $[L, B.xmin)$ at time t_L, and let R'' be the leftmost touched key in $(B.xmax, R]$ at time t_R.

We have that $\tau(L'', t_L), \tau(L', t_L) \geq \tau(x, t_L)$, for any $x \in [B.xmin, L')$. Thus, any $x \in [B.xmin, L')$ is hidden in (L'', L') after t_L.

Above P^* there can be no access to a key in $[L'', L']$, since that would contradict the fact that P^* is a block. Within P^* after t_L there can be no access to a key in $(L'', B.xmin)$, as the first such access would necessarily cause the touching of a point inside B, a contradiction to the choice of (L, t_L). An access within B is ruled out by our initial assumption.

Thus, there is no access in $(L'', L') \times (t_L, n]$, and from Lemma 4.9(iii) it follows that any $x \in [B.xmin, L')$ can not be touched after time t_L.

We argue similarly on the right side of B. ∎

As a corollary of Lemma 4.23, note that above P^* only elements within $[L', R']$ can be touched within B. If $L' = R'$, then only one element can be touched, and we are done. Therefore, we assume that L' is strictly to the left of R'.

We assume w.l.o.g. that $t_R > t_L$ (the other case is symmetric). Let (Q, t_Q) be the first access point left of B with $t_Q > t_R$ such that there is a touched point inside B at time t_Q. Observe that (Q, t_Q) is outside of P^* by our choice of (L, t_L). If no such Q exists, we are done, since we contradict Lemma 4.21 about the structure of points forced by G_{k+4}.

Let (Z, t_Z) denote the last touched point with the property that $Z \in [(P^*).xmin, L')$, and $t_Z \in [t_L, t_Q)$. If there are more such points in the same row, pick Z to be the leftmost. Note that (Z, t_Z) might coincide with (L, t_L). We have $t_Z \leq t_R$ because otherwise (Q, t_Q) cannot exist. Note than $t_Z > t_R$ would imply that keys in $[L', R']$ are hidden in (Z, R) after t_R.

Next, we make some easy observations.

Lemma 4.24. The following boxes are empty in Y:

1. $[B.xmin, B.xmax] \times [t_R + 1, t_Q - 1]$,
2. $[1, Z - 1] \times [t_Z + 1, t_R]$,
3. $[Q, Z - 1] \times [t_Z + 1, t_Q - 1]$,
4. $[Q, R' - 1] \times [t_R + 1, t_Q - 1]$.

Proof. 1. It is clear that there can be no access point inside B by assumption. Suppose there is a touched point in $[B.xmin, B.xmax] \times [t_R + 1, t_Q - 1]$, and let $(Q', t_{Q'})$ be the first (lowest) such point. Denote the access point at time $t_{Q'}$ as $(Q'', t_{Q'})$. Clearly $t_{Q'} > (P^*).ymax$ must hold, otherwise the choice of (L, t_L) or (R, t_R) would be contradicted. Also $Q'' > (P^*).xmax$ must hold, otherwise the choice of (Q, t_Q) would be contradicted.

But this is impossible, since $Q' \in [L', R']$, and $\tau(Q', t_{Q'} - 1) \leq t_R$, and thus the rectangle with corners $(Q'', t_{Q'})$ and $(Q', \tau(Q', t_{Q'} - 1))$ contains the touch point (R, t_R), contradicting the claim that Greedy touches Q' at time $t_{Q'}$.

2. All keys in $[1, Z-1]$ are hidden in $[1, Z-1]$ after t_Z. There can be no access point on the left of P^*, due to the structure of the block decomposition, and there is no access point in $[(P^*).xmin, Z-1] \times [t_Z+1, t_Q-1]$ due to the choice of Z. Hence there cannot be a touch point in $[1, Z-1] \times [t_Z+1, t_R]$.

3. First there is no access point in $[Q, Z-1] \times [t_Z+1, (P^*).ymax]$ due to the choice of (Z, t_Z) and the structure of the block decomposition. Also, there is no access point in $[Q, Z-1] \times ((P^*).ymax, t_Q-1]$: Assume there were such an access point $(Q', t_{Q'})$. Then it must be the case that $Q < Q' < (P^*).xmin$ and $(P^*).ymax < t_{Q'} < t_Q$. Any rectangle formed by (Q, t_Q) and a point in $P^* \cap B$ would have contained $(Q', t_{Q'})$, a contradiction to the fact that Greedy touches a point inside B at time t_Q.

   Next, we argue that there is no non-access touch point $(a, t) \in [Q, Z-1] \times [t_Z+1, t_Q-1]$. There are three cases.

   - $(a, t) \in [(P^*).xmin, Z-1] \times [t_Z+1, t_Q-1]$ contradicts the choice of (Z, t_Z).
   - $(a, t) \in [Q, (P^*).xmin) \times [t_Z+1, (P^*).ymax]$ contradicts the fact that all elements in $[1, Z)$ are hidden in $[1, Z)$ after t_Z, and there is no access in $[1, Z)$ in the time interval $[t_Z+1, (P^*).ymax]$, since P^* is a block.
   - $(a, t) \in [Q, (P^*).xmin) \times ((P^*).ymax, t_Q-1]$, contradicts the claim that at time t_Q Greedy touches a point inside B, since any rectangle formed by (Q, t_Q) and a point in $P^* \cap B$ would have contained (a, t).

4. Given the previous claims, it remains only to prove that there is no touched point $(a, t) \in [Z, B.xmin) \times t \in [t_R+1, t_Q-1]$. There cannot be such a touched point for $t \le (P^*).ymax$ due to the choice of (Z, t_Z). For $t > (P^*).ymax$, there cannot be souch an access point due to the structure of the block decomposition. Remains the case when (a, t) is a non-access touched point in $[Z, B.xmin) \times ((P^*).ymax, t_Q-1$. This is also impossible, as any rectangle formed by (Q, t_Q) and a point in $P^* \cap B$ would have contained (a, t), contradicting the choice of (Q, t_Q). ∎

Lemma 4.25. At time t_Q we touch R'. Let L^* be the leftmost key touched in $[L', R']$ at time t_Q (it is possible that $L^* = R'$). After time t_Q, only L^* and R' can be touched within $[L', R']$.

Proof. The first part follows from the emptiness of rectangles in Lemma 4.24. That is, the lemma implies that the rectangle formed by (Q, t_Q) and (R', t_R) is empty, so Greedy touches R' at time t_Q.

The keys $a \in (L^*, R')$ cannot be touched because they are hidden in (L^*, R') after t_Q, and there is no access point in this range after t_Q, due to the structure of the block decomposition. Suppose that some key $a \in [L', L^*)$ is touched (for the first time) at some time $t > t_Q$ by accessing element x. So the rectangle formed by (x, t) and $(a, \tau(a, t-1))$ is empty, and we know that $\tau(a, t-1) < t_R$. Notice that $x < (P^*).xmin$, for otherwise (R, t_R) would have been in the rectangle formed by (x, t) and $(a, \tau(a, t-1))$, a contradiction. Furthermore, $\tau(a, t-1) > t_Z$, for otherwise, (Z, t_Z) would have been in the rectangle formed by (x, t) and $(a, \tau(a, t-1))$, a contradiction. Now, since $\tau(a, t-1) > t_Z$, Lemma 4.24 implies that the rectangle formed by (Q, t_Q) and $(a, \tau(a, t-1))$ must be empty, therefore a is touched at time t_Q. This contradicts the choice of L^*. ∎

Lemma 4.22, 4.23, and 4.25 together imply that in total at best the $k+3$ lowest points of G_{k+4} can be touched (out of the $k+4$ needed). This means that our assumption that B is free of access points was false. Therefore G_{k+4} is an input-revealing gadget, finishing the proof of Lemma 4.20. See Figure 4.13 for an illustration.

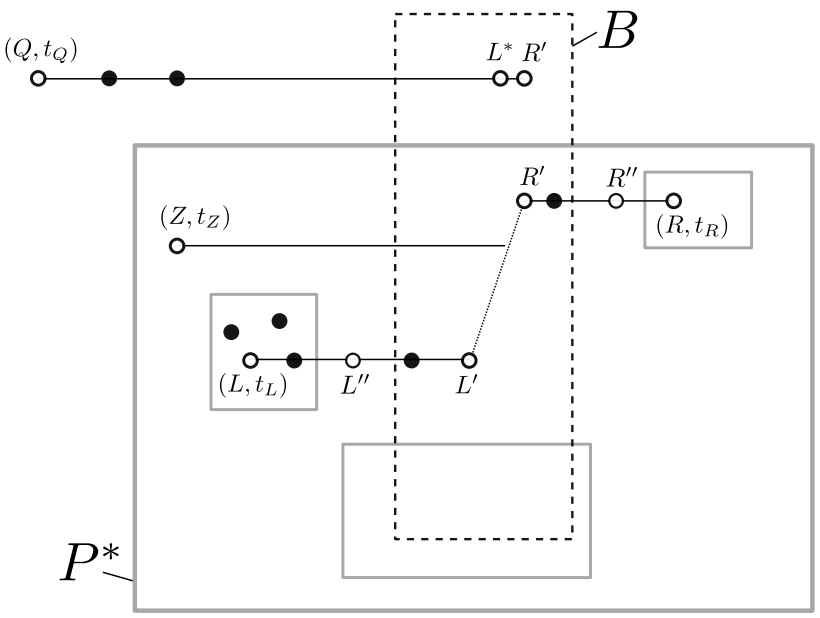

Figure 4.13: Proof of Lemma 4.20

4.7 Upper bounds on the lower bounds

4.7.1 Bound on MIR

In this section we show that the techniques developed in the previous sections can be used to prove an upper bound on the quantity $\mathsf{MIR}(X)$ for pattern-avoiding X. Recall from § 2.7.3 that $\mathsf{MIR}(X)$ is the strongest known lower bound for $\mathsf{OPT}(X)$, conjectured to asymptotically match $\mathsf{OPT}(X)$. We show that if X avoids an arbitrary fixed pattern, then $\mathsf{MIR}(X)$ is linear. The proof has the same structure as the earlier proofs, but the gadget is simpler.

Recall from § 2.7.3 that $\mathsf{MIR}(X)$ equals (up to a constant factor) $\mathrm{cost}_{G^{\boxslash}}(X) + \mathrm{cost}_{G^{\boxbslash}}(X)$, where the two terms are the costs of the Greedy$^{\boxslash}$, respectively Greedy$^{\boxbslash}$ algorithms with input X.

Greedy$^{\boxslash}$ and Greedy$^{\boxbslash}$ turn out to be easier to analyze than Greedy. We show the following.

Theorem 4.26. The cost of Greedy$^{\boxslash}$, resp. Greedy$^{\boxbslash}$ with an arbitrary initial tree, for an access sequence $X \in S_n$ that avoids an arbitrary pattern $\pi \in S_k$ is $n \cdot 2^{O(k)}$.

The same asymptotic upper bound on $\mathsf{MIR}(X)$ follows immediately from Theorem 4.26. We make use of the following observation.

Lemma 4.27. $G^{\boxslash} = (\bullet\ \bullet)$ is an input-revealing gadget for Greedy$^{\boxbslash}$ with arbitrary initial tree. $G^{\boxbslash} = (\bullet\ \bullet)$ is an input-revealing gadget for Greedy$^{\boxslash}$ with arbitrary initial tree.

Proof. We prove the first claim only, the other is symmetric. Let Y be the output of Greedy$^{\boxbslash}$ for input X, and let $(a, t_a), (b, t_b) \in Y$ order-isomorphic to $G^{\boxslash}$, such that $a < b$. Let B be the box with corners $(a, t_a), (b, t_b)$. As $\tau(a, t_a) \geq \tau(b, t_a)$, element b is hidden in $(a, b]$ after t_a. If there is no access point in B, then (b, t_b) cannot be touched, a contradiction. ∎

Proof of Theorem 4.26. Observe that $\pi \otimes G^{\boxslash}$ and $\pi \otimes G^{\boxbslash}$ are permutations of size $O(k)$, and by Lemma 4.27 they are avoided in the output of Greedy$^{\boxbslash}$, resp. Greedy$^{\boxslash}$. An application of Lemma 4.11(ii) finishes the proof. ∎

4.7.2 Bound on Wilber's bound

In this section we use a different argument to bound from above a lower bound on $\mathsf{OPT}(X)$, in this case Wilber's second lower bound $\mathcal{W}^2(X)$. More precisely, we show that if $X \in S_n$ is $\bigvee_k$-avoiding, i.e. it avoids some $\bigvee$-type permutation of size k, then $\mathcal{W}^2(X) = n \cdot O(k)$. Assuming that the conjectured $\mathcal{W}^2(X) = \Theta\big(\mathsf{OPT}(X)\big)$ holds, we obtain that for all X that avoid a constant-sized $\bigvee$-type permutation, $\mathsf{OPT}(X)$ is linear.

We use the definition of $\mathcal{W}^2$ from Figure 2.7 in § 2.6.2. Consider the access sequence $X = (x_1, \ldots, x_n) \in S_n$. For a given entry x_j, we define the offline stair $\mathrm{off}(\mathrm{x_j})$ as the subsequence of entries x_i for $i < j$ such that x_i and x_j form an unsatisfied pair in X. The *number of alternations* for x_j, denoted n_j is the number of consecutive entries in $\mathrm{off}(\mathrm{x_j})$ that are on different sides of x_j. Recall that $\mathcal{W}^2(X) = n + \sum_i n_i$.

Theorem 4.28. If $X \in S_n$ is a $\bigvee_k$-avoiding permutation, then $\mathcal{W}^2(X) = n \cdot O(k)$.

Proof. By the above definition of $\mathcal{W}^2(X)$, we observe that if $n_i = t$, for an arbitrary i, then X contains an "alternating $\bigvee$-type permutation" of size t, as a subsequence of $\mathrm{off}(\mathrm{x_i})$. By "alternating $\bigvee$-type permutation" of size t we mean the permutation $P_t = \big(\lceil t/2 \rceil, \lceil t/2 \rceil + 1, \lceil t/2 \rceil - 1, \ldots\big)$. For instance,

$$P_7 = \left(\begin{array}{ccccccc} \bullet & & & & & & \bullet \\ & \bullet & & & & \bullet & \\ & & \bullet & & \bullet & & \\ & & & \bullet & & & \end{array} \right).$$

Observe that P_{2k} contains every $\bigvee$-type permutation of size k. Therefore, if X is $\bigvee_k$-avoiding, then $n_i < 2k$, for all i, therefore $\mathcal{W}^2(X) = n \cdot O(k)$. ∎

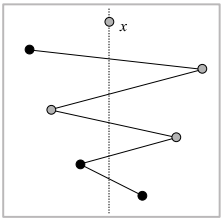

Figure 4.14: Proof of Lemma 4.29. Observe that $\{x\} \cup \mathrm{off}(x)$ contains $\left(\begin{array}{ccc} \bullet & & \bullet \\ & \bullet & \end{array} \right)$ and $\left(\begin{array}{ccc} & \bullet & \\ \bullet & & \bullet \end{array} \right)$ (highlighted).

For $\bigwedge_k$-free permutations we can obtain the same bound for Wilber's second bound computed in reverse. If $\mathcal{W}^2$ is, as conjectured, tight, then the forward and reverse variants are asymptotically equal. It appears to be open however, how big the gap between the two $\mathcal{W}^2$ variants can be.

A similar argument can also be used to bound $\mathcal{W}^2(X)$ for some other types of sequences. We illustrate this with an example. Recall that 2-decomposable (a.k.a. separable) permutations are exactly those that avoid both 2413 and 3142. In § 4.6 we showed that the cost of Greedy is linear for this type of access sequences. Now we bound $\mathcal{W}^2$ for the more general family of sequences that avoid *at least one* of 2413 and 3142.

Theorem 4.29. Let $X \in S_n$ avoiding 2413 or 3142. Then $\mathcal{W}^2(X) \leq n \cdot O(1)$.

Proof. Consider the previous definition of $\mathcal{W}^2$ and observe that if $n_i \geq 5$, then $\mathrm{off}(\mathrm{x_i})$ together with x_i contain both 2413 and 3142. The claim follows. (See Figure 4.14.) ∎

4.8 Comparisons with other bounds

We argue that the bounds for pattern-avoiding sequences shown in this section are not subsumed by previously known bounds (see e.g. § 2.4). We illustrate this by exhibiting two sequences X_1 and X_2, both of which have linear cost.

The first example sequence is defined as $X_1 = \big(1, n, 2, n-1, \ldots\big)$. It is easy to verify that X_1 is 2-decomposable. Even more strongly, X_1 avoids 231 and 213. Therefore, the fact that $\mathsf{OPT}(X) \leq n \cdot O(1)$ follows from the results of this chapter. However, the dynamic finger

bound for X_1 is easily seen to be $\Omega(n \log n)$. (Most consecutive entries have rank-difference $\Omega(n)$.) This shows that, in general, the dynamic finger bound is not stronger than the pattern-avoiding bounds. See Figure 4.15(i) for illustration. The second example shows that the reverse is also not true.

Sequence $X_2 \in S_n$ consists of a grid-like construction of size $f(n)$-by-$f(n)$ where $f(n) = \sqrt{n}/\log n$. Then, the sequence continues with a sequential access of length $n - f(n)^2$. The structure of the grid-like construction is illustrated in Figure 4.15(ii). The construction contains all patterns of size up to $\Omega\big(f(n)\big)$, therefore the pattern-avoiding results cannot show any non-trivial bound on $\mathsf{OPT}(X_2)$. However, the dynamic finger bound is for this sequence $O(n)$. To see this, observe that in the "grid" part of the sequence the differences between consecutive entries are $O\big(f(n)\big)$, except for at most $f(n)$ pairs, which have difference $O(f(n)^2)$. In the second part, the differences are all constant. Thus, the contribution of the first part to the dynamic finger bound is $o(n)$, whereas the second part contributes an $O(n)$ term only. See Figure 4.15(ii) for illustration. We refer to [24] for further observations, and to [26] for stronger separations between various bounds.

We have seen that k-decomposable sequences have linear cost (for constant k). It is tempting to conjecture that the only permutations with nonlinear cost are those that can not be recursively decomposed (i.e. simple permutations). As it is known that the vast majority of permutations is simple [21], this would be consistent with Theorem 2.20. This conjecture is, however, false: there are examples of simple permutations that have linear cost, even for Greedy, such as $X_3 = (\frac{n}{2}, n, 1, n-1, 2, \ldots)$; see Figure 4.15(iii).

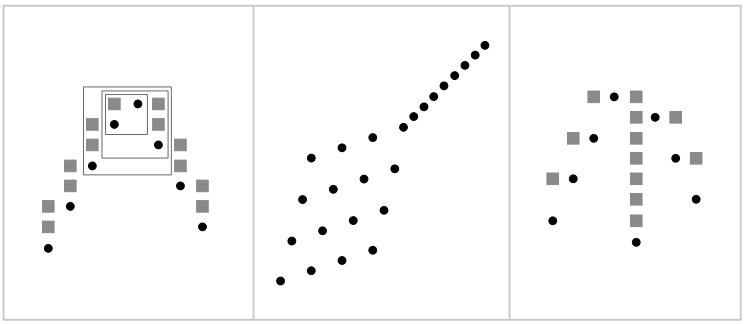

Figure 4.15: From left to right: (i) example with low pattern-avoidance parameter and high dynamic finger bound, with Greedy execution, (ii) example with low dynamic finger bound, and high pattern-avoidance parameter, (iii) non-decomposable permutation with linear cost Greedy execution.

4.9 Discussion and open questions

As discussed in § 1 and § 2, "easiness" of an access sequence in the BST model (i.e. linear cost) is more the exception than the rule. This makes it important to understand the structures that make an input sequence easy. One could even hope for an eventual "proof by exhaustion": a list of all reasons for which access sequences can have small cost, and an algorithm that matches the optimum in all these cases. This motivated our study of pattern-avoidance in access sequences.

From a theoretical point of view, we find interesting the prospect that an *arbitrary* avoided pattern may make the access sequence easy, and find such a phenomenon analogous to statements in other mathematical domains (e.g. the Erdős-Hajnal conjecture on graphs that have an arbitrary forbidden induced subgraph.)

The understanding of pattern-avoiding inputs with respect to the BST optimum and with respect to the cost of online algorithms is far from complete. The arguments using forbidden submatrices appear quite clean, but loose with the parameters, therefore, we expect that the bounds can be improved significantly. It would be interesting to extend the results obtained for special classes of permutations to more general classes.

There are limits on how far the bounds can be theoretically improved. Such barriers can be derived from enumerative results from combinatorics (i.e. how many permutations are there that avoid a certain pattern). We refer to [26] for observations in this direction.

Perhaps most interesting is the question of how well Splay does on pattern-avoiding permutations. Showing an example where Splay (or some other online algorithm) has high cost would disprove the dynamic optimality of the algorithm in question (Problem 46).

Finally, let us ponder what exactly makes pattern-avoidance a useful property in the BST model. For other structural bounds, such as dynamic finger or working set, the intuitive reason is "locality of reference". By contrast, the avoidance of a fixed pattern seems a rather global, as well as a rather finicky property – the addition of a single entry may create a pattern that was previously avoided. It is perhaps intuitive that the *offline optimum* can take advantage of pattern-avoidance, as it "knows" which pattern is missing. But why should we expect a simple online algorithm like Greedy or Splay to learn such a property on the fly?

This intuition may be misleading. A single additional entry may indeed destroy the pattern-avoidance property of a sequence, but the modified sequence is still easy. An eventual theory of access sequences should be robust to such perturbations. In the static BST model, a search sequence is easy if its elements are drawn from a low entropy distribution. In the dynamic BST model, one could also describe a family of access sequences through the distribution of the next access, when the entire sequence is picked at random from a family of sequences. For classical "easy families", such as those with low dynamic finger or working set bound, the "next-access distribution" is still a very low entropy distribution: for dynamic finger, most of the probability mass is concentrated around the last accessed key; for working set, most of the mass is on the last few accesses. In this model, the "next-access distribution" is no longer stationary as in the static case, but for both dynamic finger and working set, the distribution appears to change quite slowly from one access to the next. It is therefore reasonable to expect a BST to be able to track the change in distribution.

How does pattern-avoidance fit into this picture? Can we gain more insight from viewing it as a particular way to restrict "next-access distributions" and the step-by-step change of this distribution? It remains to be seen whether and how the easiness of pattern-avoiding sequences can be reconciled with the other, more "quantitative" structures into a coherent understanding of the BST model.

Chapter 5

Binary search trees and rectangulations

Strangers passing in the street
By chance two separate glances meet
And I am you and what I see is me.
— PINK FLOYD, *Echoes* (1970)

In this chapter we present a correspondence between sequences of rotations that serve an access sequence in the dynamic BST model and sequences of flips that transform one rectangulation into another. Rectangulations are well-studied combinatorial objects, also known as mosaic floorplans. Of particular interest to us are rectangulations constrained by points, also studied in the literature (see [3, 1] and references therein). A flip is a local operation that transforms a rectangulation by a minimal amount (analogously to rotations in trees).

The resulting view of BST executions in terms of flips in rectangulations is closely related to the geometric view of Demaine et al. [31], with some marked differences. Recall that in the geometric view of Demaine et al. we represent nodes of the tree that are touched at a certain time as points in the plane. The elegance and usefulness of the model lies in the fact that the exact details of the rotations are hidden. Online algorithms emerge in the geometric view as a natural class of geometric algorithms (those that operate in a sweepline-fashion). Particularly Greedy appears as a natural online algorithm in this view. (In fact, Greedy is so natural in the geometric view that it is not clear why any algorithm should deviate from it.)

In our rectangulation-view the online/offline distinction seems less relevant. The flip-sequence between rectangulations constructs the execution trace of a BST algorithm not line-by-line, but in an order that is constrained by the internal structure of the access sequence. Greedy still has a natural interpretation in this view (in fact, several equivalent interpretations) – but the order in which the Greedy output is constructed in our rectangulation view is neither the temporal-, nor the keyspace-order. Various concepts of the BST model can be reinterpreted in rectangulation-view. In some sense, the rectangulation-view feels more "algorithm-friendly" than the geometric view. We think that it may prove useful for the design and analysis of offline BST algorithms.

5.1 Rectangulations and Manhattan networks

A *rectangulation*[1] of an axis-parallel rectangle R is a subdivision of R into rectangles by axis-parallel line segments, no two of which may cross. A rectangulation is called *slicing* (or *guillotine*) if it can be obtained by recursively cutting a rectangle with a horizontal or vertical line into two smaller rectangles. See Figure 5.1 for illustration.

[1] Alternative names include rectangular layout, subdivision, dissection, tesselation, or mosaic floorplan

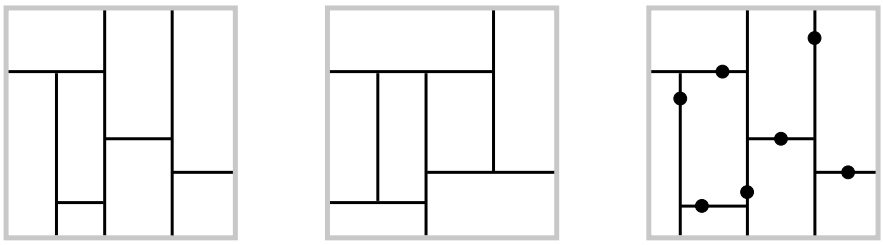

Figure 5.1: Rectangulations. From left to right: (i) slicing rectangulation, (ii) non-slicing rectangulation, (iii) rectangulation constrained by points.

The study of rectangulations in the literature is motivated by several applications. For geometric problems such as *point location*, *nearest neighbor*, or *range searching*, the commonly used data structures rely on spatial subdivisions such as trapezoidations or rectangulations [14, 87]. The popular k-d tree corresponds (in the planar case) exactly to a slicing rectangulation [12]. Rectangulations also appear in geometric approximation algorithms [74]. In communication complexity, a comparison protocol [64] for a bivariate function f corresponds to a slicing rectangulation in which every rectangle is f-monochromatic. Rectangulations are also used to model problems in VLSI circuit design [73, § 53]. In data visualization, "cartograms" based on rectangulations have been used for almost a century to represent both quantitive and relational information [38, 63], or even as a form of art.

Several theoretical aspects of rectangulations have been studied in the combinatorics literature. In particular, it is known that the number of combinatorially different rectangulations with n rectangles is given by the nth Baxter number [107, 95], and the number of combinatorially different *slicing* rectangulations with n rectangles is given by the nth Schröder number [107, 94]. Explicit bijections from rectangulations (general, respectively, slicing) have been given to natural classes of permutations counted by Baxter, respectively, Schröder numbers [2]. Felsner [40] considers various ways in which rectangulations can represent certain classes of (planar) graphs.

Most relevant to our work is the recent paper of Ackerman et al. [1] that studies certain local operations (called *flip* and *rotate*) that transform one rectangulation into another. Following the definition of Ackerman et al. we additionally constrain a rectangulation with a set P of points (no two points on the same vertical or horizontal line), requiring that every point in P is contained in the interior of a segment of the rectangulation (Figure 5.1). The flip and rotate operations in rectangulations were first introduced in [3].

Ackerman et al. [1] study sequences of rectangulations constrained by the same set P of points. In particular, they are interested in the *flip diameter* of rectangulations: the maximum number of flip and rotate operations that may be required to transform one rectangulation into another.

The main result of this chapter can be summarized as follows. Let $X \in S_n$ be a permutation, viewed at the same time as a permutation point set $X \subset [n] \times [n]$. The "trace" of a flip-sequence between two particular rectangulations constrained by X is a satisfied superset of X (and by the results of Demaine et al. described in § 2.7, it can be viewed as the execution trace of a BST algorithm serving X). Thus, the problem of efficiently serving an access sequence is reduced to the problem of finding a short flip sequence between two rectangulations. (We postpone to § 5.2 a more formal statement of results.) The source and target rectangulations are the two that are (intuitively) furthest apart: the rectangulation consisting of only vertical lines and the rectangulation consisting of only horizontal lines.

A converse of the result also holds: every sequence of BST re-arrangements that serves the access sequence (i.e. every satisfied superset) encodes a valid sequence of flips between the all-vertical and the all-horizontal rectangulations.

Our definition of the flip operation (§ 5.2) is slightly different from the definition used by Ackerman et al. [1]. Nevertheless, there is a close connection between the two definitions, which allows us to answer an open question raised by Ackerman et al. concerning the flip diameter of rectangulations. The equivalence also leads to a simplified proof of a result shown by Ackerman et al., which arises now as an immediate corollary of known results for the BST problem. We explore this topic in § 5.3.

We further transform the obtained rectangulation-view, to arrive at a particularly simple formulation of the BST problem, as a problem resembling edge relaxations in a shortest path tree. The interpretation of Greedy in these models is simple and natural. We make some preliminary observations about the BST problem in rectangulation-view in § 5.5.

Small Manhattan Networks. We also describe *Small Manhattan Network*, a problem with known connections to the BST problem (see e.g. Harmon [52]), in terms of (modified) flips between rectangulations.

A *Manhattan-path* of length k between two points $x, y \in A$ with respect to $B \supseteq A$, is a sequence of distinct points $(x = x_1, x_2, \ldots, x_k = y) \in B^k$, such that for all $i = 1, \ldots, k-1$ the two neighboring points x_i, x_{i+1} are on the same horizontal or vertical line, and both the x-coordinates and the y-coordinates of $(x_1, \ldots, x_k)$ form a monotone sequence.

Recall the definition of a *satisfied point set* from § 2.7, as a set of points without pairs forming the corners of an empty rectangle. An alternative definition of a satisfied point set given by Harmon [52] is the following.

Lemma 5.1 ([52])**.** A point set $Y \subseteq [n] \times [n]$ is satisfied, if for any two points $a, b \in Y$, there is a Manhattan-path between a and b with respect to Y.

Verifying the equivalence between the two definition is an easy exercise.

Based on the previous paragraphs and the results described in § 2.7, the BST problem of serving access sequence X can equivalently be formulated as finding a point set $Y \supseteq X$ of small cardinality in which there are Manhattan-paths with respect to Y between every pair of points in Y. An obvious easier problem is to find a point set $Y \supseteq X$, such that there are Manhattan-paths with respect to Y between every pair of points in X. As shown by Harmon [52], the optimum of this easier problem asymptotically equals the MIR lower bound on OPT (see § 2.7.3). We call the set Y of points a *Manhattan-network* of X. Such a construction is also known as an L_1*-spanner*.

The problem of connecting a given set of points in the plane by a Manhattan network is a classical network design problem that has received significant attention. The variant which we describe, i.e. where the number of added points is minimized, was studied in 2007 by Gudmundsson, Klein, Knauer, and Smid [51].

The connection between Manhattan networks and the BST problem seems not widely known. In particular, using this connection, some of the results of Gudmundsson et al. arise as corollaries of known facts about BSTs. We make this connection explicit, and we formulate further properties of small Manhattan networks, following from BST results. We also interpret Manhattan networks in the rectangulation-view outlined earlier. As mentioned (§ 2.7.3, Problem 33), the relation between MIR and OPT is mysterious. Our hope is that

reinterpreting both quantities in a similar setting will lead to new insight about the possible gap between them.

5.2 The main equivalences

Recall the definition of the dynamic BST problem described in § 2.2. In this chapter we are concerned with offline BST algorithms (defined in either the first or the second model). In fact, we will ignore trees, and focus only on the geometric view of the BST problem, i.e. the problem of finding a small satisfied superset of a set of points (§ 2.7). We refer to this problem simply as Satisfied Superset. By Theorem 2.24, all statements in this view can be translated to statements about trees.

We consider only permutation point sets $X \subset [n] \times [n]$ as inputs. For any point p, we denote by $p.x$ and $p.y$ the x-coordinate and the y-coordinate of p respectively.

5.2.1 Rectangulation problem

Variants of this problem have been studied in the literature. The formulation we describe here is new, but closely related to the problem studied by Ackerman et al. [1]. The exact difference between our model and the model of Ackerman et al., and the implications of this difference are explored in § 5.3.

Let n be an arbitrary integer (the problem size). We define the set of planar points

$$S = \{0, 1, \ldots, n+1\} \times \{0, 1, \ldots, n+1\}.$$

The set C of *corner points* is:

$$C = \{(0,0), (0,n+1), (n+1,0), (n+1,n+1)\}.$$

The set M of *margin points* is:

$$M = \{(i,0), (i,n+1), (0,i), (n+1,i) : i \in [n]\}.$$

Corner-points will not be used in any way. The remaining points of S, i.e. those in $S \setminus (M \cup C) = [n] \times [n]$, are called *non-margin points*.

A *state* (P, L) of the Rectangulation problem consists of a set $P \subseteq (S \setminus C)$ of points and a set L of horizontal and vertical line segments (in the following, simply segments) with endpoints in P.

A state (P, L) is *valid* iff it fulfills the following conditions (see Figure 5.2):

(i) **(completeness)** Each segment in L contains exactly two points from P, namely its two endpoints. (This implies that no point from P is in the interior of a segment in L.)

(ii) **(non-crossing)** No two segments in L intersect each other (except possibly at endpoints).

(iii) **(elbow-free)** Each non-margin point in P is contained in at least two segments of L, and if it is contained in exactly two segments, then the two segments must either be both vertical or both horizontal. (We cannot have the shapes ⌟, ⌞, ⌝, ⌜.)

The *initial state* (P_0, L_0) is defined with respect to an input permutation point set X of size n. The set P_0 is equal to $X \cup M$. The set L_0 contains for each non-margin point $(x, y) \in P_0 \setminus M$

two vertical segments: the one between $(x,0)$ and (x,y), and the one between (x,y) and $(x,n+1)$. It is easy to see that the initial state is valid.

An *end state* (P^*,L^*) is a valid state that consists of a point set $P^* \supseteq P_0$, and a set of segments L^*, all of them horizontal, such that they cover every point $\{0,1,\ldots,n+1\} \times [n]$. See Figure 5.2 for illustration.

Given a permutation point set X, an algorithm $\mathcal{A}$ for the Rectangulation problem transforms the initial state (P_0,L_0) determined by X into an end state (P^*,L^*), through a sequence of *valid flips*, defined below. The cost of the algorithm, denoted $\text{cost}_{\mathcal{A}}(X)$, is the number of flips in this sequence. In words, the goal is to go from the all-vertical state to the all-horizontal state through a minimum number of valid flips.

Let (P,L) be a valid state. Two points $a,b \in S \setminus C$ define a *flip*, denoted $\langle a,b\rangle$. A flip $\langle a,b\rangle$ transforms the state (P,L) into a new state (P',L') as follows. First, we let $P' = P \cup \{a,b\}$, and $L' = L \cup \{[a,b]\}$. If there exists some segment $[x,y] \in L$ that contains a in its interior, we remove $[x,y]$ from L', and add the segments $[x,a]$, and $[a,y]$ to L'. Similarly, if there exists some segment $[z,t] \in L$ that contains b in its interior, we remove $[z,t]$ from L', and add the segments $[z,b]$, and $[b,t]$ to L'.

For $\langle a,b\rangle$ to be a valid flip, it must hold that the resulting state (P',L') is a valid state. In particular, we can only add a segment $[a,b]$ if it is horizontal or vertical, and if it does not intersect existing segments (except at a or b). After every flip we can remove from L' an arbitrary number of segments. By removing segments we must not violate the elbow-free property. For instance, we can only remove a vertical segment if its non-margin endpoints are contained in two horizontal segments of L' (in other words, the endpoints have extensions both to the left and to the right).

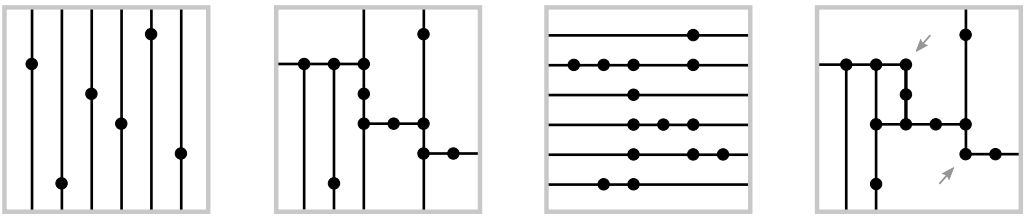

Figure 5.2: Rectangulation problem. From left to right: (i) initial state corresponding to $X = (2,6,4,3,1,5)$, (ii) a valid intermediate state, (iii) a valid end state, (iv) an invalid intermediate state (observe that the state is not elbow-free). Margin points are not shown.

The following theorem captures the connection (in one direction) between Rectangulation and Satisfied Superset.

Theorem 5.2. Any algorithm $\mathcal{A}$ for the Rectangulation problem can be transformed (in polynomial time) into an algorithm $\mathcal{B}$ for the Satisfied Superset problem, such that for all inputs X, we have $\text{cost}_{\mathcal{B}}(X) = O\big(\text{cost}_{\mathcal{A}}(X)\big)$.

Proof. Consider an algorithm $\mathcal{A}$ for Rectangulation, executed from initial state (P_0,L_0), defined by an input permutation X. As we run $\mathcal{A}$, we construct a set $Y \supseteq X$, that is a solution for Satisfied Superset (this is our new algorithm $\mathcal{B}$). The process is straightforward: Initially we let $Y = X$. Whenever $\mathcal{A}$ performs a flip $\langle a,b\rangle$, we let $Y = Y \cup (\{a,b\} \setminus M)$. In words, we construct a superset of X by adding every non-margin endpoint created while flipping from the all-vertical to the all-horizontal state in Rectangulation. The cost of $\mathcal{A}$ is equal to the

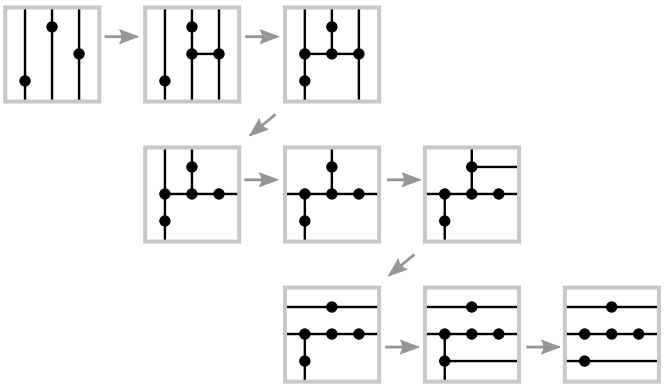

Figure 5.3: A sequence of valid flips from initial state to end state. Margin points are not shown.

number of flips. Since each flip adds at most two points to Y, the claim on the cost of $\mathcal{B}$ is immediate.

It remains to show that the point set Y thus constructed is satisfied. Suppose otherwise, that in the end there are two points $a, b \in Y$, that are not on the same horizontal or vertical line, and the rectangle with corners a, b contains no other point of Y. Without loss of generality, assume that a is above and to the left of b. Let $\langle a, a' \rangle$ be the last flip in the execution of $\mathcal{A}$ such that a' is on the same horizontal line as a and to the right of a. Let $\langle b', b \rangle$ be the last flip such that b' is on the same horizontal line as b and to the left of b. (There have to be such flips, otherwise $\mathcal{A}$ would not produce a valid end state.) Since the rectangle with corners a, b is empty, b' must be to the left of a, and a' must be to the right of b.

Suppose that the flip $\langle b', b \rangle$ occurs earlier than the flip $\langle a, a' \rangle$ (the other case is symmetric), and consider the state before the flip $\langle a, a' \rangle$. In that state there must be a vertical segment with top endpoint at a, otherwise a would be contained in at most two segments, not both horizontal or vertical, contradicting the elbow-free property. Let a^* be the bottom endpoint of the vertical segment with top endpoint a. The point a^* must be strictly below b, for otherwise the rectangle with corners a, b would contain it. This means that $[a, a^*]$ intersects $[b', b]$, contradicting that we are in a valid state. We conclude that Y is a satisfied superset of X. ∎

The following converse of Theorem 5.2 also holds.

Theorem 5.3. Any algorithm $\mathcal{B}$ for the Satisfied Superset problem can be transformed (in polynomial time) into an algorithm $\mathcal{A}$ for the Rectangulation problem, such that for all inputs X, we have $\text{cost}_{\mathcal{A}}(X) = O\big(\text{cost}_{\mathcal{B}}(X)\big)$.

Proof. Consider an algorithm $\mathcal{B}$ for Satisfied Superset that for input X outputs a point set $Y \supseteq X$. We construct a sequence of flips that transform the initial state (P_0, L_0) of the Rectangulation problem determined by X into an end state (P^*, L^*), such that $P^* \setminus M = Y$ (this is our new algorithm $\mathcal{A}$). We define $\mathcal{A}$ such that every flip creates a new horizontal segment whose endpoints are in $Y \cup M$, and no horizontal segment is ever removed during the course of the algorithm. The claim on the cost of $\mathcal{A}$ is immediate, since each flip can be charged to one of its (non-margin) endpoints, and each point in Y has at most two flips charged to it. The removal of vertical segments does not contribute to the cost.

We run algorithm $\mathcal{A}$ until we reach an end state, maintaining the invariant that in every state (P, L), we have $(P \setminus M) \subseteq Y$. The invariant clearly holds in the initial state (P_0, L_0) determined by X. Algorithm $\mathcal{A}$ consists of two types of greedy steps, executed in any order: (1) if at any point, some valid flip $\langle a, b \rangle$ is possible, such that $a, b \in Y \cup M$, then execute it,

and (2) if at any point, some vertical segment $[a,b] \in L$ that contains no point from Y (except possibly its endpoints) can be removed, then remove it.

It remains to be shown that the algorithm does not get stuck, i.e. that there is always an operation of type (1) or (2) that can be executed, unless we have reached a valid end state. Consider an intermediate state (P,L) during the execution of $\mathcal{A}$ and suppose for contradiction that there is no available operation of either type.

Consider two points $q, q' \in Y \cup M$ on the same horizontal line, q to the left of q', such that $[q,q']$ is not in L, and the segment $[q,q']$ contains no point of Y in its interior. (Note that q and q' are not necessarily in P.) If there is no such pair of points, then we are done, since all horizontal lines are complete, and all remaining vertical segments can be removed. Among such pairs, consider the one where q is the rightmost, in case of a tie, choose the one where q' is the leftmost.

Call a point $q \in P$ *left-extensible* if it is not the right endpoint of a segment in L, and *right-extensible* if it is not the left endpoint of a segment in L.

Observe that throughout the execution of $\mathcal{A}$, for any state (P,L), every point in Y is contained in some segment of L. Since $\langle q,q'\rangle$ is not a valid flip, $[q,q']$ must intersect some vertical line $[z,z'] \in L$ (assume w.l.o.g. that z is strictly above, and z' is strictly below $[q,q']$). Observe that $[z,z']$ cannot contain a point of Y in its interior. If it were to contain such a point z^*, then z^* would be the left endpoint of some segment missing from L, contradicting the choice of q. Thus, since removing $[z,z']$ is not a valid step, it must be that one of z and z' is a non-margin point that is left- or right-extensible. If z or z' were right-extensible, that would contradict the choice of q. Therefore, one of them must be left-extensible. Assume w.l.o.g. that z is left-extensible.

Since Y is satisfied, by Lemma 5.1 there has to be a point $w \in Y \setminus \{z,q\}$ either on the horizontal segment $[(q.x,z.y),z]$, or on the vertical segment $[z,(z.x,q.y)]$. Since $[z,z']$ cannot contain a point of Y in its interior, it must be the case that w is on $[(q.x,z.y),z]$, and choose w to be closest to z. But then the segment $[w,z]$ is missing from L, contradicting the choice of q,q' because $q.x \le w.x \le z.x < q'.x$. ∎

Theorem 5.2 and Theorem 5.3 state that the Rectangulation and Satisfied Superset problems are polynomial-time equivalent. Observe that the proofs, in fact, show something stronger: for an arbitrary permutation point set X, a point set $Y \supseteq X$ is a solution for Satisfied Superset exactly if $Y \cup M$ is the point set of a valid (and reachable) end state for Rectangulation.

5.2.2 A tree relaxation problem

Consider again a permutation point set $X = \{(x_i,i) : i \in [n]\}$ as input. We define a *monotone tree* on X to be a rooted tree that has X as the set of vertices, and whose edges are all going away from the root according to the vertical ordering of the points. That is, if two points $a = (a.x,a.y)$, $b = (b.x,b.y)$, with $a,b \in X$ are the endpoints of an edge in a monotone tree on X, then a is closer to the root than b (in graph-distance) iff $a.y < b.y$. Recall that all points in X have distinct x- and y-coordinates. It follows that $(x_1,1)$ is the root of every monotone tree on X.

We are concerned with two special monotone trees on X. The *treap* on X is the binary search tree with the x-coordinates as keys, and the y-coordinates as heap-priorities. That is, the lowest point $(x_1,1) \in X$ is the root of the tree, and the points left of $(x_1,1)$ form its left subtree, and the points right of $(x_1,1)$ form its right subtree, defined in a recursive fashion. The *path* on X is a tree that connects all elements through a path by increasing y-coordinate, i.e. in the order $(x_1,1),\ldots,(x_n,n)$. It is easy to verify that both the treap and the path defined

on X are unique and that they form monotone trees on X. Observe that the definition of a monotone tree does not require the tree to fulfill the search tree property or even to be binary. See Figure 5.4 for illustration.

Given a permutation point set X, an algorithm $\mathcal{A}$ for the Tree Relaxation problem transforms the treap on X to the path on X through a sequence of *valid edge-flips*, defined below. The cost of the algorithm, denoted $\text{cost}_{\mathcal{A}}(X)$, is the number of edge-flips in this sequence.

Let T be a monotone tree on X. A valid edge-flip in T is defined as follows. Consider a vertex r of T that has at least two children. Sort the children of r by their x-coordinate, and let a and b be two children that are neighbors in this sorted order, such that a is below b (the y-coordinate of a is smaller than the y-coordinate of b). Then the edge-flip $(a \to b)$ adds the edge (a,b) to T and removes the edge (r,b) from T. It is easy to verify that a valid edge-flip maintains the monotone tree property of T. The edge-flip operation is reminiscent of an edge-relaxation in shortest-path algorithms (performed in reverse). See Figure 5.5 for illustration.

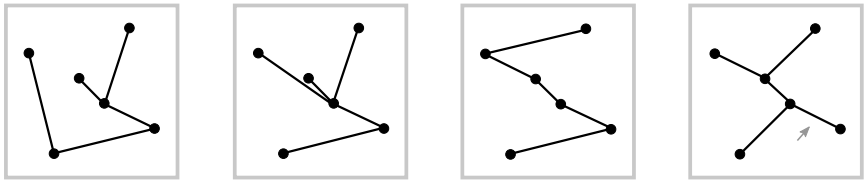

Figure 5.4: Tree relaxation problem. From left to right: (i) treap on $X = (2,6,4,3,1,5)$, (ii) an intermediate monotone tree on X, (iii) path on X, (iv) an invalid intermediate state (tree is not monotone).

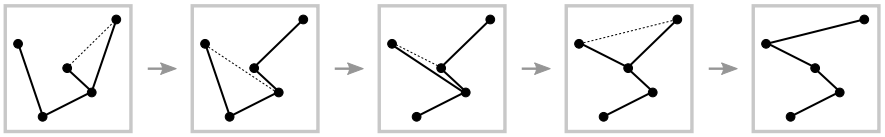

Figure 5.5: A sequence of valid edge-flips from initial (treap) state to end (path) state.

The Tree Relaxation problem is closely related to the Rectangulation problem (and as a consequence, to the BST problem), as shown by the following theorem.

Theorem 5.4. Any algorithm $\mathcal{A}$ for the Tree Relaxation problem can be transformed (in polynomial time) into an algorithm $\mathcal{B}$ for the Rectangulation problem, such that for all inputs X of size n, we have $\text{cost}_{\mathcal{B}}(X) = O\big(\text{cost}_{\mathcal{A}}(X) + n\big)$.

Proof. We start $\mathcal{B}$ with an *initial phase*, then we simultaneously run algorithm $\mathcal{A}$ for Tree Relaxation on X, and output the operations of $\mathcal{B}$ for Rectangulation on X, such that we output at most two flips in $\mathcal{B}$ for every edge-flip in $\mathcal{A}$. We finish $\mathcal{B}$ with a *cleanup phase*. Both the initial phase and the cleanup phase consist of $O(n)$ flips, to be specified later. In the end we show that $\mathcal{B}$ reaches a valid end state of Rectangulation. Since the total number of flips performed is at most $O(n) + 2 \cdot \text{cost}_{\mathcal{A}}(X)$, the claim on the cost follows.

After the initial phase, we maintain three invariants, denoted I_1, I_2, and I_3. Informally, the role of the invariants is to enforce that the current monotone tree of $\mathcal{A}$ is, at all times, the *segment visibility graph* of the horizontal segments in the current rectangulation of $\mathcal{B}$.

In any given state (P,L) during the execution of $\mathcal{B}$, let H_i denote the set of horizontal segments in L at height i.

I_1 **(contiguity):** For all i, the union of the horizontal segments in H_i form a contiguous horizontal segment, which we denote h_i.

I_2 **(nesting):** For all i, denote the x-coordinate of the left (resp. right) endpoints of h_i as ℓ_i (resp. r_i). Let $x_{i_1}, \ldots, x_{i_k}$ be the children of x_t in the current monotone tree on X, sorted by x-coordinate (i.e. $x_{i_1} < \cdots < x_{i_k}$). We have that the endpoints of the segments $h_{i_1}, \ldots, h_{i_k}$ are aligned, and not overhanging the parent segment h_t. More precisely, $\ell_t \le \ell_{i_1}$, and $r_{i_k} \le r_t$, and for all $j = 1, \ldots, k-1$, we have $r_{i_j} = \ell_{i_{j+1}}$. For the root x_1 of the tree, we have $\ell_1 = 0$, and $r_1 = n+1$.

I_3 **(visibility):** Let $x_{i_1}, \ldots, x_{i_k}, x_t$ be defined as before. For $j = 1, \ldots, k$, let us denote by $\mathcal{R}_j$ the axis-aligned rectangle with corners (ℓ_{i_j}, i_j), (r_{i_j}, t). Let $\mathcal{R}_0$ be the rectangle with corners (ℓ_t, t), $(\ell_{i_1}, n+1)$, and let $\mathcal{R}_{j+1}$ be the rectangle with corners $(r_{i_k}, n+1)$, (r_t, t). We have that the interiors of the rectangles $\mathcal{R}_0, \mathcal{R}_1, \ldots, \mathcal{R}_{j+1}$ are not intersected by any segment in L in the current state of $\mathcal{B}$. Furthermore, the vertical sides of the rectangles $\mathcal{R}_0, \mathcal{R}_1, \ldots, \mathcal{R}_{j+1}$ are either touching the margin, or fully covered by segments in L.

It remains to describe the steps of the algorithm $\mathcal{B}$.

Initial phase. For each ($i = n, \ldots, 1$), flip $\langle L_i, x_i \rangle$ and $\langle x_i, R_i \rangle$, where L_i (R_i) is the leftmost (rightmost) point such that the corresponding flip is valid. After every flip, remove all possible vertical segments from the current state before proceeding to the next i. In particular, remove the vertical segment with endpoints $(x_i, 0)$, (x_i, i).

Recall that in the beginning, $\mathcal{A}$ is in the state that the current tree is the treap on X. Let T be the treap on X. We show that the invariants hold after the initial phase. Observe that I_1 holds trivially: after the initial phase we have the horizontal contiguous segments $h_i = [L_i, R_i]$.

We prove I_2 and I_3 by induction. Clearly, if $|X| = 1$, the invariants hold. Consider the last step, when we flip $\langle L_1, x_1 \rangle$ and $\langle x_1, R_1 \rangle$, for suitable L_1, R_1. Denote by v the vertical segment with x-coordinate equal to x_1. Observe that for all $i > 1$, the segments h_i can intersect v only at their endpoints (L_i and R_i). Since none of these points are extended both to the left and to the right, no portion of v has been removed before this step. This means that the rectangulations on the two sides of v are independent of each other, i.e. they would have been the same even if the input on the other side of v were different. In particular, this means that, by induction I_2 and I_3 hold for the rectangulations on the two sides of v, corresponding to the left and right subtrees of x_1 in the treap T.

Let x_i and x_j be the left, respectively right child of x_1 in T (one of the two might be missing, in case $x_1 = 1$ or $x_1 = n$). By I_2, we have that h_i extends horizontally from 0 to x_1, and h_j extends horizontally from x_1 to $n+1$. Since the vertical segments below every point x_i have been removed, we can execute the flips $\langle (0,1), x_1 \rangle$ and $\langle x_1, (n+1, 1) \rangle$. Finally, since x_1 is complete, we can remove the vertical segment with top endpoint x_1.

Both I_2 and I_3 are established for the full tree T, completing the induction.

Flips during the execution of $\mathcal{A}$. Let x_i and x_j be neighboring children of x_t in the current tree, such that $i < j$, and the valid edge-flip $(x_i \to x_j)$ is executed in $\mathcal{A}$. Assume w.l.o.g. that $x_i < x_j$. By I_1, there exist contiguous horizontal segments h_i and h_j. Observe that h_i is below h_j. Let ℓ_i, ℓ_j, and ℓ_t (r_i, r_j, and r_t) denote the x-coordinates of the left (right) endpoints

of h_i, h_j, and h_t. By I_2, we have $\ell_t \le \ell_i < r_i = \ell_j < r_j \le r_t$. Let $L = (r_i, i)$, and let $R = (r_j, i)$. Then the flip $\langle L, R\rangle$ is executed in $\mathcal{B}$.

Let us verify that the flip is valid. Due to invariant I_3, the rectangle with corners (r_i, i) and (r_j, j) has empty interior, therefore the flip intersects no vertical segment. Moreover, by I_3, the right side of the rectangle is covered by segments. Therefore, the flip creates no crossing, elbow, or point of degree one, it is therefore a valid flip.

After the flip, x_j is the child of x_i, and x_i is the child of x_t. It is easy to verify that the invariants are maintained, except for the following case: Let x_k be the rightmost child of x_i *before the flip*, with endpoints ℓ_k and r_k. After the flip, x_k and x_j are neighboring siblings, but it may happen that their endpoints are not aligned, i.e. $r_k < \ell_j$. If this is the case, we need to perform an additional flip. Suppose that h_k is lower than h_j. Then execute in $\mathcal{B}$ the flip $\langle (r_k, j), (\ell_j, j)\rangle$. In the case when h_j is lower than h_k, execute the flip $\langle (r_k, k), (\ell_j, k)\rangle$. The flips are valid by I_3 before the edge-flip in $\mathcal{A}$, and by the flip, I_3 is re-established using at most two flips.

In the end, if $\mathcal{A}$ is a correct algorithm for Tree Relaxation, it will end with a path tree. Let $h_1, \dots, h_n$ be the horizontal lines corresponding to the current state in the execution of $\mathcal{B}$.

Cleanup phase. From invariants I_1, I_2, and I_3, it follows that $\ell_1 \le \ell_2 \le \cdots \le \ell_n < r_n \le \cdots \le r_1$. We can transform this state to a valid end state for Rectangulation, with the valid flips (in this order): $\langle 0, \ell_1\rangle, \langle 0, \ell_2\rangle, \dots, \langle 0, \ell_n\rangle, \langle r_1, n+1\rangle, \dots, \langle r_n, n+1\rangle$. ∎

From the previous proof it seems that algorithms for Tree Relaxation correspond to algorithms for Rectangulation of a very restricted kind. Whether the restriction is really significant, remains an intriguing open question. Therefore, we ask the following.

Problem 49. Does a converse of Theorem 5.4 hold?

5.2.3 Signed satisfied supersets and rectangulations

We define the "one-sided" (alternatively: "signed") variants of the studied problems.

Signed Satisfied Superset. Recall the definition of ◪- and ◹-type pairs of points (§ 2.7.3). A point set $Y \subseteq [n] \times [n]$ is ◪-*satisfied* if it has no unsatisfied ◪-type pair, and it is ◹-*satisfied* if it has no unsatisfied ◹-type pair. Note that Y is satisfied if and only if it is both ◪-satisfied and ◹-satisfied.

Given a permutation point set X of size n, an algorithm $\mathcal{A}$ for the ◪- (◹-) Satisfied Superset problem outputs a point set Y with $X \subset Y \subseteq [n] \times [n]$, such that Y is ◪-satisfied (◹-satisfied). The cost of $\mathcal{A}$ is the size of the set Y, denoted $\mathrm{cost}_{\mathcal{A}}(X)$. We generally call the ◪- and ◹-Satisfied Superset problems *Signed* Satisfied Superset.

Let $\mathsf{OPT}_{◪}(X)$ and $\mathsf{OPT}_{◹}(X)$ be the optimum cost for the ◪-, resp. ◹-Satisfied Superset problem for X. Demaine et al. [31] show that Greedy$^{◪}$ and Greedy$^{◹}$ (described in § 2.7.3) compute the optimum solution, i.e.

$$\mathrm{cost}_{G^{◪}}(X) = \mathsf{OPT}_{◪}(X),$$

$$\mathrm{cost}_{G^{◹}}(X) = \mathsf{OPT}_{◹}(X).$$

Recall from § 2.7.3 that $\mathsf{SG}(X)$ denotes the sum of the two quantities, and that $\mathsf{SG}(X) = \Theta\big(\mathsf{MIR}(X)\big)$.

Signed Rectangulation. Let $[p,q]$ be a vertical segment, and $[q,r]$ be a horizontal segment. We say that $[p,q]$ and $[q,r]$ form a *(¬⌞)-elbow* if (i) p is above q and r is on the right of q, or (ii) p is below q and r is on the left of q. Symmetrically, we say that $[p,q]$ and $[q,r]$ form a *(⌟⌜)-elbow* if (i) p is above q and r is on the left of q, or (ii) p is below q and r is on the right of q.

A state (P,L) of the Rectangulation problem is *(¬⌞)-elbow-free*, respectively *(⌟⌜)-elbow-free* if each non-margin point in P is contained in at least two segments of L, and if it is contained in exactly two segments, then they must not form a (¬⌞)-elbow, resp. (⌟⌜)-elbow.

We define the (¬⌞)-Rectangulation problem the same way as Rectangulation, except that we only require that each state (P,L) of the (¬⌞)-Rectangulation problem is *(⌟⌜)-elbow-free* instead of elbow-free (i.e. the (¬⌞)-elbows are allowed). We similarly define the (⌟⌜)-Rectangulation problem. We generally call the (¬⌞)- and (⌟⌜)-Rectangulation problems *Signed* Rectangulation.

Given any set S of allowed elbows, we can similarly define the S-Rectangulation problem in an obvious way, e.g. (¬⌜)-Rectangulation problem or (⫬⌞)-Rectangulation problem.

Theorem 5.5. Any algorithm $\mathcal{A}$ for the (⌟⌜)- or (¬⌞)-Rectangulation problem can be transformed (in polynomial time) into an algorithm $\mathcal{B}$ for the ◪-, respectively ◹-Satisfied Superset problem, such that for all inputs X, we have $\text{cost}_{\mathcal{B}}(X) = O\big(\text{cost}_{\mathcal{A}}(X)\big)$.

Proof. We only show the case of (¬⌞)-Rectangulation. The other case is symmetric. The argument is similar as in Theorem 5.2. Initially, let $Y = X$. We construct an algorithm $\mathcal{B}$ from $\mathcal{A}$ by adding to Y every non-margin endpoint created while flipping from the all-vertical to the all-horizontal state in Rectangulation. The cost of $\mathcal{A}$ is equal to the number of flips. Since each flip adds at most two points to Y, the claim on the cost of $\mathcal{B}$ is immediate.

We claim that Y is ◹-satisfied. Suppose otherwise, that there are two points $a,b \in Y$ where a is above and to the left of b. Let $\langle a,a'\rangle$ be the last flip in the execution of $\mathcal{A}$ such that a' is on the same horizontal line as a and to the right of a. Let $\langle b',b\rangle$ be the last flip such that b' is on the same horizontal line as b and to the left of b. (There have to be such flips, otherwise $\mathcal{A}$ would not produce a valid end state.) Since the rectangle with corners a,b is empty, b' must be to the left of a, and a' must be to the right of b.

Suppose that the flip $\langle b',b\rangle$ occurs earlier than the flip $\langle a,a'\rangle$ (the other case is symmetric), and consider the state before the flip $\langle a,a'\rangle$. In that state there must be a vertical segment with top endpoint at a, otherwise a would be contained in (⌟⌜)-elbow. (This is the only difference from the proof of Theorem 5.2.) Let a^* be the bottom endpoint of the vertical segment with top endpoint a. The point a^* must be strictly below b, for otherwise the rectangle with corners a,b would contain it. This means that $[a,a^*]$ intersects $[b',b]$, contradicting that we are in a valid state. We conclude that Y is a ◹-satisfied superset of X. ∎

Theorem 5.6. Any algorithm $\mathcal{B}$ for the ◪-Satisfied Superset problem can be transformed (in polynomial time) into an algorithm $\mathcal{A}$ for the (⌜)-Rectangulation problem, such that for all inputs X, we have $\text{cost}_{\mathcal{A}}(X) = O\big(\text{cost}_{\mathcal{B}}(X)\big)$.

Proof. The argument is similar as in Theorem 5.3. Let Y be a ◪-satisfied set constructed by $\mathcal{B}$. We construct an algorithm $\mathcal{A}$ that maintains the state (P,L) with the following operations in a greedy manner: (1) if some valid flip $\langle a,b\rangle$ is possible where $a,b \in Y \cup M$, then execute it, and (2) if some vertical segment $[a,b] \in L$ containing no point from Y (except possibly its endpoints) can be removed, then remove it. Here, the valid flip is defined according to the (⌜)-Rectangulation problem. We claim that $\mathcal{A}$ reaches an end state. The cost of $\mathcal{A}$ follows with the same argument as in Theorem 5.3. Suppose for contradiction that $\mathcal{A}$ gets stuck at an intermediate state (P,L).

As in Theorem 5.3, consider two points $q, q' \in Y \cup M$ on the same horizontal line, q to the left of q', such that $[q, q']$ is not in L, and the segment $[q, q']$ contains no point of Y in its interior. If there is no such pair of points, then we are done, since all horizontal lines are complete, and all remaining vertical segments can be removed. Among such pairs, consider the one where q is the rightmost, in case of a tie, choose the one where q' is the leftmost. Left-extensibility and right-extensibility are defined as in the proof of Theorem 5.3.

Observe that throughout the execution of $\mathcal{A}$, for any state (P, L), every point in Y is contained in some segment of L. Since $\langle q, q' \rangle$ is not a valid flip, $[q, q']$ must intersect some vertical line $[z, z'] \in L$ (assume w.l.o.g. that z is strictly above, and z' is strictly below $[q, q']$). Observe that $[z, z']$ cannot contain a point of Y in its interior. If it were to contain such a point z^*, then z^* would be the left endpoint of some segment missing from L, contradicting the choice of q. Thus, since removing $[z, z']$ is not a valid step according to $(\ulcorner)$-Rectangulation problem, it must be that either z is left-extensible, z is right-extensible or z' is right-extensible. If z or z' were right-extensible, that would contradict the choice of q. Therefore, z is left-extensible.

Since Y is $\boxslash$-satisfied, by the statement analogous to Lemma 5.1 there has to be a point $w \in Y \setminus \{z, q\}$ either on the horizontal segment $[(q.x, z.y), z]$, or on the vertical segment $[z, (z.x, q.y)]$. Since $[z, z']$ cannot contain a point of Y in its interior, it must be the case that w is on $[(q.x, z.y), z]$, and choose w to be closest to z. But then the segment $[w, z]$ is missing from L, contradicting the choice of q, q' because $w.x \geq q.x$. ∎

Theorems analogous to Theorem 5.6 for $(\lrcorner)$,$(\llcorner)$,$(\neg)$-Rectangulation can be shown similarly. By Theorems 5.5 and 5.6 we have that (i) $\boxslash$-Satisfied Superset problem, $(\lrcorner\ulcorner)$,$(\ulcorner)$,$(\lrcorner)$-Rectangulation problems are equivalent, and (ii) $\boxbslash$-Satisfied Superset problem, $(\neg\llcorner)$,$(\neg)$,$(\llcorner)$-Rectangulation problems are equivalent.

Let us also consider the case of allowing two types of elbows that are neighbors in the clockwise ordering of the four possible elbows, i.e. the $(\neg\ulcorner)$, $(\urcorner\llcorner)$, $(\lrcorner\llcorner)$, $(\urcorner\lrcorner)$-Rectangulation problems. Together with $(\neg\llcorner)$ and $(\lrcorner\ulcorner)$, these are all possible cases with two types of allowed elbows. We argue that $(\neg\ulcorner)$-Rectangulation is trivial: for every input of size n there is a flip sequence of length $O(n)$. Due to the symmetries of the problem, the same holds for the $(\urcorner\llcorner)$, $(\lrcorner\llcorner)$, $(\urcorner\lrcorner)$ cases, and consequently, also for Rectangulation with three or four types of allowed elbows.

The algorithm for obtaining a linear sequence of flips for $(\neg\ulcorner)$-Rectangulation is as follows. First execute an inital phase as in the proof of Theorem 5.4, then complete the horizontal rectangulation line by line, from top to bottom. At step k, assume that the horizontal lines $k, \ldots, n$ are completed. Remove every vertical segment whose top endpoint is at height k (observe that this can only create $(\neg\ulcorner)$-elbows). Then, complete the horizontal line at height $k-1$, by flipping horizontal segments at height $k-1$ to the maximum extent possible (this can not create crossings, since we removed vertical segments in the previous step).

5.3 Consequences for rectangulations

In this section we study the Flip Diameter problem introduced by Ackerman et al. [1]. Ackerman et al. study the distance between two rectangulations constrained by the same set of points, where distance refers to the shortest sequence of local operations that transform one rectangulation into the other.

The concept of rectangulation studied by Ackerman et al. is the same as the one we defined in § 5.2.1, apart from the fact that we keep track of all intersection points that are created in the sequence of transformations from one rectangulation to another, whereas in the problem studied by Ackerman et al. this is not explicitly needed. Their definition of a

rectangulation is essentially the union of all segments in L, for a given state (P, L). In this section we adopt the same convention, i.e. we consider two states (P, L) and (P', L') equivalent if the union of all segments in L equals the union of all segments in L'.

The Flip Diameter problem asks, given a set of points constraining rectangulations, to find the largest possible distance between two rectangulations. This is in contrast to the problem described in § 5.2.1, where we are concerned with the distance between two *particular* rectangulations, namely the all-horizontal, and the all-vertical one.

Finally, the local operations used by Ackerman et al. are slightly different from the flip operation defined in § 5.2.1. In the following, we describe the rotate and flip operation used by Ackerman et al. in the context of our Rectangulation problem. We call these two operations *A-rotate* and *A-flip*.

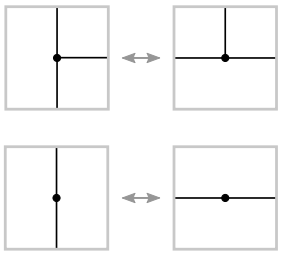

Figure 5.6: (above) A-rotate operations. (below) A-flip operations.

Given a valid state (P, L) of Rectangulation, an *A-rotate* operation consists of removing a segment $[x, y]$ from L, and adding a new segment $[x, z]$ to L. If z is contained in the interior of some segment $[a, b] \in L$, we remove $[a, b]$ from L, and add $[a, z]$ and $[z, b]$ to L. We denote the resulting set of segments L' and we let $P' = P \cup \{z\}$. The A-rotate operation is valid, if $[x, y]$ and $[x, z]$ have different orientations (i.e. one of them horizontal, the other vertical), and if the resulting state (P', L') is a valid state of Rectangulation.

Given a valid state (P, L) of Rectangulation, an *A-flip* operation consists of removing two segments $[x, y]$ and $[y, z]$ from L and adding new segments $[v, y]$ and $[y, w]$ to L. If v is contained in the interior of some segment $[a, b] \in L$, we remove $[a, b]$ from L and add $[a, v]$ and $[v, b]$. Similarly, if w is contained in the interior of some segment $[c, d] \in L$, we remove $[c, d]$ from L and add $[c, w]$ and $[w, d]$. We denote the resulting set of segments L' and we let $P' = P \cup \{v, w\}$. The A-flip operation is valid, if $[x, y]$ and $[y, z]$ have the same orientiation (i.e. both horizontal or both vertical), $[v, y]$ and $[y, w]$ have the same orientation (i.e. both horizontal or both vertical), different from the orientation of $[x, y]$, and if the resulting state (P', L') is a valid state of Rectangulation. We illustrate the A-rotate and A-flip operations in Figure 5.6.

We make the simple observation that both an A-rotate and an A-flip can be simulated with one, respectively two flip operations as defined in § 5.2.1.

Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be two valid states of Rectangulation given by a permutation point set X. Let $d(\mathcal{R}_1, \mathcal{R}_2)$ denote the shortest number of A-rotate and A-flip operations that transform $\mathcal{R}_1$ to $\mathcal{R}_2$, and let $d'(\mathcal{R}_1, \mathcal{R}_2)$ be the shortest number of flip operations (according to the definitions in § 5.2.1) that transform $\mathcal{R}_1$ to $\mathcal{R}_2$. The Flip Diameter problem studied by Ackerman et al. asks for the quantity $\mathsf{diam}(X) = \max_{\mathcal{R}_1, \mathcal{R}_2} \{d(\mathcal{R}_1, \mathcal{R}_2)\}$, where the maximum is over all valid states of Rectangulation determined by X. Let $\mathsf{OPT}^{\mathsf{R}}(X) = \min_{\mathcal{A}} \mathrm{cost}_{\mathcal{A}}(X)$, i.e. the smallest cost of any algorithm for the Rectangulation problem with input X, as defined in § 5.2.1. We make the following easy observation.

Theorem 5.7. For an arbitrary permutation X, we have

$$\mathsf{OPT}^{\mathsf{R}}(X) \leq 2 \cdot \mathsf{diam}(X).$$

Proof. By the observation that two flip operations can simulate an A-rotate or an A-flip, we have that $d'(\mathcal{R}_1, \mathcal{R}_2) \leq 2 \cdot d(\mathcal{R}_1, \mathcal{R}_2)$, for all $\mathcal{R}_1, \mathcal{R}_2$.

Let $\mathsf{diam}'(X) = \max_{\mathcal{R}_1, \mathcal{R}_2} \{d'(\mathcal{R}_1, \mathcal{R}_2)\}$, where the maximum is over all valid states of Rectangulation reachable from the initial state determined by X. By the previous observation, we have $\mathsf{diam}'(X) \leq 2 \cdot \mathsf{diam}(X)$. Furthermore, $\mathsf{diam}'(X) \geq \mathsf{OPT}^{\mathsf{R}}(X)$, since $\mathsf{OPT}^{\mathsf{R}}(X)$ refers to the shortest number of flips between two particular Rectangulations. The claim follows. ∎

We give a new interpretation of a result of Ackerman et al. They prove the following.

Theorem 5.8 ([1, § 3])**.** There exists a permutation X of size n such that $\mathsf{diam}(X) = \Omega(n \log n)$.

The proof of Ackerman et al. uses the bitwise reversal permutation R_n (§ 2.6) and argues about certain geometric constraints that hold for any possible sequence of A-rotate and A-flip operations on rectangulations constrained by this permutation.

We give a very simple alternative proof: from Theorem 2.21 we know that for every BST algorithm $\mathcal{A}$ it holds that $\mathrm{cost}_{\mathcal{A}}(R_n) = \Omega(n \log n)$. Using the equivalences between the BST problem and Satisfied Superset (Theorem 2.24), respectively, between Satisfied Superset and Rectangulation (Theorem 5.2), it follows that $\mathsf{OPT}^{\mathsf{R}}(R_n) = \Omega(n \log n)$. The application of Theorem 5.7 finishes the proof.

Ackerman et al. raise the open question of computing the average of $\mathsf{diam}(X)$ over all permutation point sets X of size n. A simple argument shows that this value is $\Omega(n \log n)$.

Theorem 5.9. For a random permutation X of size n, we have $\mathbb{E}_X[\mathsf{diam}(X)] = \Omega(n \log n)$.

Proof. It is known (Theorem 2.20) that $\mathbb{E}_X[\mathrm{cost}_{\mathcal{A}}(X)] = \Omega(n \log n)$ for any BST algorithm $\mathcal{A}$. For any algorithm $\mathcal{B}$ for Rectangulation, there is a BST algorithm $\mathcal{A}$ such that $\mathrm{cost}_{\mathcal{A}}(X) = O\big(\mathrm{cost}_{\mathcal{B}}(X)\big)$ for all X (Theorems 2.24 and 5.2). Thus, $\mathbb{E}_X[\mathrm{cost}_{\mathcal{B}}(X)] = \Omega(n \log n)$ for every Rectangulation algorithm $\mathcal{B}$. Since $\mathrm{cost}_{\mathcal{B}}(X) \le 2 \cdot \mathsf{diam}(X)$ for some algorithm $\mathcal{B}$ for Rectangulation (Theorem 5.7), the claim follows. ∎

It would be of interest to describe natural classes of inputs X of size n, for which the value of $\mathsf{diam}(X)$ is small, i.e. linear in n. For the BST problem there has been extensive research on query sequences that can be served with linear total cost (e.g. § 2.4, § 4). For any such sequence X we obtain (via Theorems 2.24 and 5.3) that $d'(V, H) = O(n)$, where V is the Rectangulation initial state determined by X, and H is any Rectangulation valid end state reachable from V.

The claim that easy (linear cost) permutations for the BST problem are also easy (linear cost) for the Flip Diameter problem follows immediately, if the following two conjectures hold.

Problem 50. Prove or disprove:

- For any two Rectangulation states $\mathcal{R}_1$ and $\mathcal{R}_2$, we have $d(\mathcal{R}_1, \mathcal{R}_2) = O\big(d'(\mathcal{R}_1, \mathcal{R}_2)\big)$.
- For any two Rectangulation states $\mathcal{R}_1$ and $\mathcal{R}_2$, we have $d'(\mathcal{R}_1, \mathcal{R}_2) = O\big(d'(H, V)\big)$.

The first conjecture claims that A-rotate and A-flip operations are essentially equivalent with our flip operation, and the second conjecture claims that the distance between the all-vertical and any all-horizontal state is asymptotically the largest among all distances. For instance, Ackerman et al. state the open question of whether $\mathsf{diam}(X)$ is linear, if X is a *separable permutation*. Recall Theorem 4.19, stating that separable permutations are linear-cost for the BST problem. Thus, if the above conjectures hold, then we get an affirmative answer to this question.

Based on Theorems 2.24, 5.2, and 5.3, we know that our flip operation between rectangulations captures every possible BST algorithm. It would be interesting to give a characterization of the class of BST algorithms that are captured by the A-flip and A-rotate operations.

Ackerman et al. [1, § 2] show that $\mathsf{diam}(X) = O(n \log n)$, for all permutations X of size n. The proof is constructive (an algorithm with worst-case $O(n \log n)$ operations). The proposed algorithm and its analysis are quite sophisticated, for instance, the proof relies on the Four-color theorem. However, if we interpret this algorithm in the special case of transforming the all-vertical rectangulation to the all-horizontal rectangulation (i.e. for our Rectangulation

problem), the output of the algorithm is rather simple: it corresponds to a static balanced binary search tree (whose cost for serving X is clearly $O(n \log n)$).

We know that A-flip and A-rotate operations can capture non-trivial BST algorithms (i.e. other than static trees), since Ackerman et al. show that the flip diameter of a diagonal point set is $O(n)$. In the language of binary search trees, this means that the sequence $S = (1, 2, \ldots, n)$ is accessed in time $O(n)$, i.e. the sequential access condition. Recall that no static BST can achieve this bound. It is instructive to interpret the algorithm of Ackerman et al. [1, § 4] given for this particular input in terms of BST rotations (the algorithm corresponds to a straightforward offline BST algorithm tailored for serving the access sequence S). As we have seen, in the BST world, such a bound is achieved by several general-purpose algorithms, including Splay and Greedy.

5.4 Consequences for Manhattan networks

We stated that given an input permutation point set X, a set $Y \supseteq X$ is a solution for the Satisfied Superset problem, if and only if for all $a, b \in Y$, there is a Manhattan path between a and b with respect to Y (Lemma 5.1).

As mentioned in § 5.1, a straightforward relaxation of the Satisfied Superset problem is to require Manhattan paths only between pairs of points from the input point set X. This is the Small Manhattan Network problem, which is of independent interest.

More precisely, an algorithm $\mathcal{A}$ for Small Manhattan Network outputs, given a permutation point set X, a point set $Y \supseteq X$, such that for all $a, b \in X$, there is a Manhattan path between a and b with respect to Y. The cost of $\mathcal{A}$, denoted $\mathrm{cost}_{\mathcal{A}}(X)$ is the size of the set Y. In the context of the BST problem, Small Manhattan Network was defined by Harmon [52] as a lower bound for the BST optimum. In a geometric setting, the problem was studied by Gudmundsson, Klein, Knauer, and Smid [51].

Let $\mathsf{OPT}^{\mathsf{M}}(X) = \min_{\mathcal{A}} \mathrm{cost}_{\mathcal{A}}(X)$, i.e. the optimum Small Manhattan Network solution for input X.

From the above definition of Small Manhattan Network as a less restricted Satisfied Superset problem, the following result is immediate.

Theorem 5.10 ([52])**.** For an arbitrary permutation X we have

$$\mathsf{OPT}^{\mathsf{M}}(X) = O\big(\mathsf{OPT}(X)\big).$$

In § 5.2.3 we described the Signed Satisfied Superset problem and we denoted by $\mathsf{SG}(X)$ the Signed Satisfied Superset optimum for X, i.e. the sum of the ◪- and ◹-Satisfied Superset optima:

$$\mathsf{SG}(X) \;=\; \mathsf{OPT}_{◪}(X) \;+\; \mathsf{OPT}_{◹}(X).$$

Further, if $\mathsf{OPT}^{\mathsf{R}}_{(⌟⌜)}(X)$ and $\mathsf{OPT}^{\mathsf{R}}_{(⌝⌞)}(X)$ denote the optima for (⌟⌜)- and (⌝⌞)-Rectangulation (defined in § 5.2.3), let us denote by $\mathsf{OPT}^{\mathsf{sR}}(X)$ the Signed Rectangulation optimum for X, i.e.

$$\mathsf{OPT}^{\mathsf{sR}}(X) \;=\; \mathsf{OPT}^{\mathsf{R}}_{(⌟⌜)}(X) \;+\; \mathsf{OPT}^{\mathsf{R}}_{(⌝⌞)}(X).$$

The following statement extends Theorems 2.30 and 2.31. The equivalences were shown by Demaine et al. [31] and Harmon [52], except for the new quantity $\mathsf{OPT}^{\mathsf{sR}}(X)$, whose relation with the other quantities follows from Theorems 5.5 and 5.6.

Theorem 5.11. For an arbitrary permutation X we have (up to constant factors):

$$\mathsf{SG}(X) = \mathsf{OPT}^{\mathsf{sR}}(X) = \mathsf{MIR}(X) = \mathsf{OPT}^{\mathsf{M}}(X).$$

For completeness, we give an alternative proof of the statement $\mathsf{OPT}^{\mathsf{M}}(X) = \Omega\big(\mathsf{MIR}(X)\big)$. (The other direction $\mathsf{OPT}^{\mathsf{M}}(X) = O\big(\mathsf{MIR}(X)\big)$ follows from the fact that the union of the Greedy$^{\boxslash}$ and Greedy$^{\boxbslash}$ outputs for X is on one hand, a feasible Manhattan Network solution, on the other hand, a constant-approximation for $\mathsf{MIR}(X)$.) The following proof is inspired by a proof of a similar flavor given by Demaine et al. [31] for a different statement, and is somewhat simpler than the proof given by Harmon [52].

Proof. Recall from § 2.7.3 the definitions of $\mathcal{R}_{\boxslash} \subseteq \mathcal{I}_X$ and $\mathcal{R}_{\boxbslash} \subseteq \mathcal{I}_X$ as the largest $\boxslash$-independent, respectively $\boxbslash$-independent sets of rectangles defined by X. Let $\mathsf{OPT}^{\mathsf{M}}_{\boxslash}(X)$ be the size of the smallest point set $Y \supseteq X$ such that for all pairs of points $a, b \in X$ such that a is above and to the right of b, there is a Manhattan path between a and b in Y. Let $\mathsf{OPT}^{\mathsf{M}}_{\boxbslash}(X)$ the size of the smallest point set $Y \supseteq X$ such that for all pairs of points $a, b \in X$ such that a is above and to the left of b, there is a Manhattan path between a and b in Y. The proof relies on the following lemma.

Lemma 5.12. For all permutations X, we have

$$\mathsf{OPT}^{\mathsf{M}}_{\boxslash}(X) \geq |X| + |\mathcal{R}_{\boxslash}(X)|,$$

$$\mathsf{OPT}^{\mathsf{M}}_{\boxbslash}(X) \geq |X| + |\mathcal{R}_{\boxbslash}(X)|.$$

As we have $\mathsf{OPT}^{\mathsf{M}}(X) \geq \max\{\mathsf{OPT}^{\mathsf{M}}_{\boxslash}(X), \mathsf{OPT}^{\mathsf{M}}_{\boxbslash}(X)\}$, using Lemma 5.12, we obtain:

$$\mathsf{OPT}^{\mathsf{M}}(X) \geq |X| + \frac{1}{2}|\mathcal{R}_{\boxslash}(X)| + \frac{1}{2}|\mathcal{R}_{\boxbslash}(X)| = \theta\big(\mathsf{MIR}(X)\big).$$

It remains to prove Lemma 5.12. We prove the first statement only, as the other statement is entirely symmetric.

Let $\mathcal{R}$ be a maximally wide rectangle in $\mathcal{R}_{\boxslash}(X)$, and let v be a vertical line segment with endpoints on the opposite horizontal sides of $\mathcal{R}$, such that none of the other rectangles in $\mathcal{R}_{\boxslash}(X)$ intersect v. (Such a v exists by the maximality of $\mathcal{R}$, and the independence-property of $\mathcal{R}_{\boxslash}(X)$.) To simplify the argument, take v such that the x- coordinate of v is fractional. Let a and b be the corners of $\mathcal{R}$, such that a is above and to the right of b. Consider a Manhattan path $P_{ab} = (a = x_1, \ldots, x_k = b)$, where $x_i \in Y$, and Y is the solution achieving $\mathsf{OPT}^{\mathsf{M}}(X)$. Let p and q be the unique neighboring points in P_{ab} such that p is to the left of v, and q is to the right of v. (Observe that p and q are on the same horizontal line, and there is no point of Y in the interior of $[p, q]$.) Charge the cost of the rectangle (a, b) to the pair (p, q). Remove $\mathcal{R}$ from $\mathcal{R}_{\boxslash}(X)$, and continue the process. Observe that the pair (p, q) can not be charged again in the future (since no other rectangle intersects v). Furthermore, the number of pairs to which the rectangles can be charged is at most $\mathsf{OPT}^{\mathsf{M}}_{\boxslash} - |X|$. The claim follows. ∎

Equipped with these observations, we revisit the Small Manhattan Network problem studied by Gudmundsson et al. [51] and reinterpret some of their results. Gudmundsson et al. show the following.

Theorem 5.13 ([51, Thm. 1])**.** For every point set X of size n we have $\mathsf{OPT}^{\mathsf{M}}(X) = O(n \log n)$.

The solution given by Gudmundsson et al. is constructive, i.e. an algorithm that constructs a Manhattan network with $O(n \log n)$ points. We can sketch it as follows: Split X with a vertical line v into two equal subsets, and add the projection of all points in X to v to the solution. Repeat the process recursively on the subsets of X on the two sides of v. It is straightforward to verify both that the resulting point set is a valid Manhattan Network solution, and that its size is $O(n \log n)$.

We observe that an alternative way to prove $\mathsf{OPT}^{\mathsf{M}}(X) = O(n \log n)$ is simply to note that $\mathsf{OPT}(X) = O(n \log n)$, and apply Theorem 5.10. The upper bound on $\mathsf{OPT}(X)$ follows from the observation that a BST access sequence can be served with logarithmic cost per access. In fact, it is not hard to see that the algorithm given by Gudmundsson et al. corresponds to the execution trace of a static balanced BST that serves access sequence X. (The vertical line v corresponds to the root of the tree, that is touched by every access, and the same holds at every recursive level.)

Gudmundsson et al. further show the following result.

Theorem 5.14 ([51, Thm. 4])**.** For some point set X of size n we have $\mathsf{OPT}^{\mathsf{M}}(X) = \Omega(n \log n)$.

The instance used to show this is (essentially) the bitwise reversal sequence R_n described in § 2.6. Again, the result can be shown in an alternative way, observing that $\mathsf{MIR}(R_n) = \Omega(n \log n)$ (Theorem 2.21), and using the correspondence of Theorem 5.11.

The correspondence between Small Manhattan Network and BST yields further results for the Small Manhattan Network problem. In particular, for the BST problem we have several fine-grained bounds on the cost of the optimum solution, such as dynamic finger, working set, lazy finger, or the traversal bound (§ 2.4). Results of this type give immediate upper bounds on the complexity of the Small Manhattan Network solution for inputs with particular structure. (Although some of these structures may seem unusual in a geometric setting.)

More importantly, by computing the union of the Greedy$^{\boxslash}$ and Greedy$^{\boxbslash}$ outputs for a point set X we obtain a polynomial time $O(1)$-approximation for the Small Manhattan Network problem.

Furthermore, similarly to the result for Flip Diameter, we obtain the following.

Theorem 5.15. For a random point set X of size n, the complexity of the Small Manhattan Network optimum is $\Theta(n \log n)$.

Since only the relative ordering of the points matters for Small Manhattan Network (and not the distances between points), by random point set we mean a point set in general position whose relative ordering corresponds to a random permutation. Again, the proof only needs the result of Wilber [106] or its stronger form by Blum et al. (Theorem 2.20) and its variant that holds for permutations [24].

In § 4.7.1 we showed that for a point set X that avoids an arbitrary constant-size permutation pattern, it holds that $\mathsf{MIR}(X) = O(n)$. This yields the following observation.

Theorem 5.16. Every planar point set that avoids a fixed permutation pattern admits a Manhattan network of linear complexity.

5.5 Consequences for BSTs

The "flip" and "tree-relax" models of the BST problem (described in § 5.2) give new interpretations of several well-studied concepts in the BST world (e.g. upper and lower bounds). We list some preliminary observations and questions in this direction.

Upper bounds. For an arbitrary pair (u, v) of points, where $u = (u.x, u.y)$, and $v = (v.x, v.y)$, let us define the *height* of (u, v) as $h(u, v) = |u.y - v.y|$, and the *width* of (u, v) as $w(u, v) = |u.x - v.x|$. For an arbitrary monotone tree T, let $h(T)$ be the sum of heights of all edges in T, and let $w(T)$ be the sum of widths of all edges in T.

Consider a permutation access sequence $X \in S_n$ and the corresponding treap T on X, as well as the path P on X (both are defined in § 5.2.2). Consider an edge-flip operation $(a \to b)$ in some tree T' that adds the edge (a,b) and removes the edge (r,b), where r is the parent of b in T'. Let the resulting tree be T''. We make the following two observations:

$$h(T') - h(T'') = h(r,b) - h(a,b) = h(r,a) \geq 1, \quad \text{and}$$
$$w(T') - w(T'') = w(r,b) - w(a,b) = -w(r,a) \leq -1.$$

In words, the total height strictly decreases, and the total weight strictly increases in every edge-relax operation. We also observe that $h(P) = n-1$ (in the end, every edge is of height 1). It follows that the quantities $H = h(T) - h(P) = h(T) - n + 1$, and $W = w(P) - w(T)$ are upper bounds on the cost of *every* algorithm for the Tree Relaxation problem. Given X, both W and T can be easily computed. The bounds are however, not very strong, as both W and T can be as large as $\Theta(n^2)$. Nevertheless, for certain highly structured inputs, such as for permutations close to the sequential access $(1,\ldots,n)$, the quantities can yield asymptotically tight bounds for OPT.

We may conjecture that both bounds can be strengthened, by summing the *logarithms* of the heights, respectively weights. More precisely, we define for an arbitrary monotone tree T the quantities

$$H'(T) = \sum_{(u,v)\in T} \log\big(h(u,v)\big),$$
$$W'(T) = \sum_{(u,v)\in T} \log\big(w(u,v)\big).$$

If T is the initial treap on X, and P is the path on X, then $W'(P)$ is the classical *dynamic finger* bound, and $H'(T)$ is (essentially) the classical *working set* bound [91]. (Working set is typically defined in the literature with respect to the last occurrence of the *same element* in an access sequence. However, for permutation access sequences it is natural to consider the occurrence of the nearest successor or predecessor among the already seen elements, which is exactly what the quantity $H'(T)$ captures.) The bounds $W'(P)$ and $H'(T)$ no longer hold for *every* algorithm, but we know that $W'(P)$ is asymptotically matched by Splay [29, 28] and by Greedy [54]. Is $H'(T)$ a valid upper bound on $\mathsf{OPT}(X)$?

A different upper bound on the cost of every Tree Relaxation algorithm can be computed by summing for all vertices in the monotone tree, the distance to the root. (By distance we mean the number of edges on the path to the root.) Again, it can be seen that this quantity strictly increases with ever edge-flip operation, reaching in the end the value $n(n-1)/2$.

New heuristics. The above quantities suggest natural greedy heuristics for Tree Relaxation. For instance, in every step we may perform the edge-flip that decreases the total edge height the most, or that increases the total edge width the most, or that increases the total distance-from-the-root the most. We leave for further research the question of how efficient (and how natural) the corresponding BST algorithms are.

In the Rectangulation problem we flip from the all-vertical to the all-horizontal state. Natural measures of quality for any intermediate state include the total length of remaining vertical segments, the total length of horizontal segments, or the difference between the two quantities. It would seem natural to perform flips that greedily optimize any of these quantities. It remains open whether the resulting BST algorithms are efficient.

Interpretations of Greedy. Refer to § 2.7.1 for the description of GeometricGreedy, and let $\text{cost}_{GG}(X)$ denote its cost for input permutation X. Let us define the following algorithms for the Rectangulation and Tree Relaxation problems.

GreedyRectangle is an algorithm for Rectangulation defined as follows. In the initial phase, for every $i = n, \ldots, 1$, execute the flips $\langle L_i, x_i \rangle$, and $\langle x_i, R_i \rangle$ where L_i (R_i) is the leftmost (rightmost) point such that the corresponding flip is valid. After every flip remove as many vertical segments as possible.

Afterwards, in every step of the algorithm (until we reach an end state), let k be the largest value such that the horizontal lines with y-coordinates $1, \ldots, k$ are fully covered by segments in the current state (P, L). Let q be the highest point in P such that (i) q is visible from below (i.e. the vertical segment between q and the kth horizontal line does not contain any points of P in its interior and its interior is not intersected by any horizontal segment in L), and (ii) q is left- or right-extensible. Observe that there must be such a point q, unless we are in an end state. If q is left- (right-) extensible, then execute the flip $\langle L, q \rangle$ (resp. $\langle q, R \rangle$), where L (R) is the leftmost (rightmost) point such that the flip is valid. Note that q is either left- or right-extensible, but not both. After the flip, remove as many vertical segments as possible. Let $\text{cost}_{GR}(X)$ denote the cost of GreedyRectangle.

GreedyRelax is an algorithm for Tree Relaxation defined as follows. In every step of the algorithm (until we reach the path monotone tree), let T be the current monotone tree (initially the treap on X). Let r be the nearest node to the root (in graph-distance) that has at least two children. Possibly r is the root itself. Let b be the child of r with highest y-coordinate, and let a be the sibling of b that is its neighbor in the ordering by x-coordinates. If there are two such neighboring siblings, let a be the one with higher y-coordinate. Perform the edge-flip $(a \to b)$. Let $\text{cost}_{GT}(X)$ denote the cost of GreedyRelax.

Theorem 5.17. For every permutation X of size n the quantities $\text{cost}_{GG}(X)$, $\text{cost}_{GR}(X)$, and $\text{cost}_{GT}(X)$ are equal, up to a constant factor and an additive term $O(n)$.

Proof sketch. $\text{cost}_{GG}(X) = \text{cost}_{GR}(X)$.

Let Y be the output of Greedy for X. We run GreedyRectangle and maintain a number of invariants similar to those in the proof of Theorem 5.4.

I_1 **(contiguity):** For all i, the union of the horizontal segments at height i form a contiguous horizontal segment, which we denote h_i.

I_2' **(modified nesting):** For all i, denote the left (resp. right) endpoints of h_i as ℓ_i (resp. r_i). Two segments h_i, h_j do not "overhang", i.e. assuming $i < j$, either $\ell_j \geq r_i$ or $\ell_i \geq r_j$ (avoidance) or $\ell_i \leq \ell_j < r_j \leq r_i$ (containment) holds. Thus, the segments h_i for $i = 1, \ldots, n$ induce a tree over $\{x_1, \ldots, x_n\}$ by the visibilities between the segments h_i (two horizontal segments see each other if there is a vertical segment properly intersecting both of them and does not touch any of the other horizontal segments). Denote this tree as T, and observe that it is a monotone tree on X.

I_2'' **(modified nesting):** Let $x_{i_1}, \ldots, x_{i_k}$ be the children of x_t in the current tree T, sorted by x-coordinate (i.e. $x_{i_1} < \cdots < x_{i_k}$). We have that the endpoints of the segments $h_{i_1}, \ldots, h_{i_k}$ are not overlapping, and not overhanging the parent segment h_t. More precisely, $\ell_t \leq \ell_{i_1}$, and $r_{i_k} \leq r_t$, and for all $j = 1, \ldots, k-1$, we have $r_{i_j} \leq \ell_{i_{j+1}}$. For the root x_1 of the tree, we have $\ell_1 = 0$, and $r_1 = n+1$.

(Observe that we allow gaps between the siblings.)

I_3' **(modified visibility):** Let $x_{i_1}, \ldots, x_{i_k}, x_t$ be defined as before. For $j = 1, \ldots, k$, let us denote by $\mathcal{R}_j$ the axis-aligned rectangle with corners (ℓ_{i_j}, i_j), (r_{i_j}, t). Let $\mathcal{R}_0$ be the rectangle with corners (ℓ_t, t), $(\ell_{i_1}, n+1)$, and let $\mathcal{R}_{j+1}$ be the rectangle with corners $(r_{i_k}, n+1)$, (r_t, t). Also for $j = 1, \ldots, k-1$ let $\mathcal{R}'_j$ be the rectangle with corners (r_{i_j}, t), $(\ell_{i_{j+1}}, n+1)$.

We have that the interiors of the rectangles $\mathcal{R}_0, \mathcal{R}_1, \ldots, \mathcal{R}_{j+1}$ and $\mathcal{R}'_1, \ldots, \mathcal{R}'_{k-1}$ are not intersected by any segment in L in the current state of GreedyRectangle. Furthermore, the vertical sides of the rectangles $\mathcal{R}_0, \mathcal{R}_1, \ldots, \mathcal{R}_{j+1}$ and $\mathcal{R}'_1, \ldots, \mathcal{R}'_{k-1}$ are either touching the margin, or fully covered by segments in L.

(The novelty is that the gaps between siblings also define empty rectangles.)

I_4 **(equivalence):** For every segment h_i, the points in $h_i \cap P$ are exactly those in $h_i \cap Y$ (i.e. what Greedy would add). Since the h_is cover everything in the end, this invariant leads to the proof.

Finally, a technical invariant.

I_5 **(satisfaction):** For any i and j such that x_i is the ancestor of x_j in the current monotone tree, there are no points $x \in P \cap h_i$ and $y \in P \cap h_j$, such that x and y form an unsatisfied pair (with respect to the current point set P).

We need to show that all invariants hold after the initial phase, and that they are maintained through every flip. The claim follows from invariant I_4 and the observation that GreedyRectangle terminates, i.e. it reaches a valid end state, since it can always pick a valid flip.

We omit the details, except for the maintenance of invariant I_4.

Consider a step of GreedyRectangle when the first k lines are completed and we flip $\langle L, q \rangle$. Suppose $q \in h_i$. By choice of q, we know that x_i is the child of x_k, and $x_k, \ldots, x_1$ form a path to the root.

If L is at the margin, no new point in P is created. If L is not at the margin, then $L.x$ is the right boundary of some h_j where x_j is a sibling of x_i to the left of x_i. (Either x_j and x_i are neighboring siblings and there was a gap between them, or there is one sibling between them). Observe that $j < i$.

To show that I_4 is maintained, we need to prove that: (1) L is part of the Greedy output Y, and (2) there is no point of Y in the interior of $[L, q]$.

(2) Suppose for contradiction that Greedy would add some point in $[L, q]$. We use the following observation: running Greedy on the original X produces the same output as running Greedy on the current P. If we run Greedy on current P and it adds something in $[L, q]$, it means that the stair of q contains some point below with x coordinate between $L.x$ and $q.x$. But such a point cannot be in the "empty rectangle", so it must be in an ancestor segment of x_i in T, contradicting I_5.

(1) Suppose for contradiction that Greedy would not add point L. This would mean that some other point is added by Greedy in the rectangle between q and $(L.x, r_j)$ (that would hide $(L.x, r_j)$). But then I_5 is contradicted, the same way as before.

The $\text{cost}_{GR}(X) = \text{cost}_{GT}(X)$ claim of the theorem follows from the simulation of GreedyRelax by a Rectangulation algorithm, according to Theorem 5.4, and observing that the Rectangulation algorithm maintains the invariants. We omit the details. ∎

We remark that the correspondence to Greedy can be maintained even if we choose the next flip differently from the above description of GreedyRectangle. As long as we perform a flip from a left- or right-extensible point q visible from below (conditions (i) and (ii)), we need not pick the highest such q. A sufficient condition is that the endpoint of the flip (L or R) is either a margin point, or a non-margin point that is not visible from below. This yields a family of algorithms, all producing the same solution as Greedy, but in different orders. We omit the details.

5.6 Discussion

There exist known connections between rectangulations and BSTs. In particular, slicing rectangulations have a straightforward BST-representation, which is useful in geometric applications such as planar point location. For general rectagulations more complex BST-based representations are known, such as the twin binary tree structure given by Yao, Chen, Cheng, and Graham [107].

Intriguingly, we are not aware of any connection between these connections and the connection described in this chapter. Despite the fact that we relate a sequence of rectangulations with a sequence of BSTs, we stress that we do not directly match rectangulations to trees and flips to rotations. An intermediate rectangulation in our proposed model corresponds to an abstract state of a BST algorithm, in which some partial structure of the intermediate trees has been committed to, while other structure is still left undecided. We find it an interesting question, whether the known BST-based representations of rectangulations have any relevance to the connection discussed here.

A further – as of yet – unrelated correspondence is the one between slicing rectangulations and separable permutations, both counted by the Schröder numbers [2]. (We studied separable permutations in § 4 as a particularly easy class of access sequences.)

It remains interesting to explore the analogies between various concepts in the BST, respectively, rectangulation models. For instance, what is the class of BST algorithms that correspond to rectangulation flip sequences restricted to slicing rectangulations? What is the class of algorithms for Rectangulation that correspond to BST algorithms with the search-path-only restriction? Which BST algorithms are captured by the tree-relaxation view (Problem 49)? We expect such questions to yield further insight into the combinatorial structures in question. Ultimately, however, the value of the presented connections should be judged depending on whether they lead to progress on the fundamental questions about binary search trees.

List of open problems

Chapter 1

Problem 1. Can the rotation-distance between two BSTs be computed in polynomial time?

Chapter 2

Problem 2. Given the access frequencies of n elements, can the optimum static BST be computed in time $o(n^2)$?

Problem 3. Given $X \in [n]^m$ is it possible to compute $\mathsf{OPT}(X)$ in polynomial time? What is the best approximation computable in polynomial time?

Problem 4. Does it hold for all X that $\mathsf{OPT}^{\mathrm{sp,root}}(X) = O\big(\mathsf{OPT}^{\mathrm{sp}}(X)\big)$?

Problem 5. Do any of the following statements hold for all X?

- $\mathsf{OPT}^{\mathrm{str,root}}(X) = O\big(\mathsf{OPT}^{\mathrm{str}}(X)\big)$,
- $\mathsf{OPT}^{\mathrm{str}}(X) = O\big(\mathsf{OPT}^{\mathrm{sp}}(X)\big)$,
- $\mathsf{OPT}^{\mathrm{sp}}(X) = O\big(\mathsf{OPT}(X)\big)$.

Problem 6. Is there a strict online algorithm $\mathcal{A}$ such that for all X we have $\mathrm{cost}_{\mathcal{A}}(X) = O\big(\mathsf{OPT}(X)\big)$?

Problem 7. Is there a strict online algorithm $\mathcal{A}$ such that for all X we have $\mathrm{cost}_{\mathcal{A}}(X) = O\big(\mathsf{OPT}^{\mathrm{str,root}}(X)\big)$?

Problem 8. Do any of the following statements hold for all X?

- $\mathsf{OPT}^{\mathrm{str}}(X) = O\big(\mathsf{OPT}^{\mathrm{len}}(X)\big)$,
- $\mathsf{OPT}^{\mathrm{len}}(X) = O\big(\mathsf{OPT}(X)\big)$.

Problem 9. Is there a lenient online algorithm $\mathcal{A}$ such that for all X we have $\mathrm{cost}_{\mathcal{A}}(X) = O\big(\mathsf{OPT}(X)\big)$?

Problem 10. Is there an unlimited online algorithm $\mathcal{A}$ such that for all X we have $\mathrm{cost}_{\mathcal{A}}(X) = O\big(\mathsf{OPT}(X)\big)$?

Problem 11. Is there an $o(\log n)$-competitive strict online algorithm? Is there an $o(\log\log n)$-competitive lenient or unlimited online algorithm?

Problem 12. Characterize the sequences $X \in [n]^m$ (or at least $X \in S_m$) for which $\mathsf{OPT}(X) = O(m)$.

Problem 13. Is there a strict online algorithm (with a fixed initial tree of its choice) that has cost $O(n)$ on every traversal sequence of length n?

Problem 14. Is there a strict online algorithm that matches the lazy finger bound?

Problem 15. Is there a simpler proof that Splay matches the dynamic finger bound?

Problem 16. Does Splay match the lazy finger bound?

Problem 17.

- Is the cost of Splay (with every initial tree) $O(n)$ for every traversal sequence of length n?
- Is the cost of Splay (with some fixed initial tree) $o(n \log n)$ for every traversal sequence of length n?

Problem 18. Is Splay c-competitive for some $c = o(\log n)$?

Problem 19. Is there some $c = o(\log n)$, such that the cost of Splay for X is at most $c \cdot \mathsf{OPT}^{\mathrm{str,root}}(X)$?

Problem 20. Is there some $c = o(\log n)$, such that the cost of GreedyFuture for X is at most $c \cdot \mathsf{OPT}(X)$?

Problem 21. Is there some $c = o(\log n)$, such that the cost of GreedyFuture for X is at most $c \cdot \mathsf{OPT}^{\mathrm{str,root}}(X)$? Even more specifically, can we bound the cost of GreedyFuture as c times the cost of Splay?

Problem 22. Is the cost of GreedyFuture $O(n)$ for every traversal sequence of length n with every initial tree?

Problem 23. Is there a strict online algorithm $\mathcal{A}$, and some $c = o(\log n)$, such that the cost of $\mathcal{A}$ for every (sufficiently long) input sequence X is at most c times the cost of GreedyFuture for X? In particular, is Splay such an algorithm?

Problem 24. Prove or disprove that $\mathcal{W}^1(X) = \Theta\big(\mathsf{OPT}(X)\big)$ for all $X \in [n]^m$.

Problem 25. Let T be a balanced BST over $[n]$. Prove or disprove that $\mathcal{W}^1_T(X) = \Theta\big(\mathsf{OPT}(X)\big)$ for all $X \in S_n$.

Problem 26. Prove or disprove that $\mathcal{W}^2(X) = \Theta\big(\mathsf{OPT}(X)\big)$ for all $X \in [n]^m$.

Problem 27. Prove or disprove that $\mathcal{W}^1(X) = \Theta\big(\mathcal{W}^2(X)\big)$ for all $X \in [n]^m$.

Problem 28. Is $\mathrm{cost}_{GG}(X) \le \frac{4}{3} \cdot \mathsf{OPT}(X)$ for every X?

Problem 29. Is the cost of GeometricGreedy asymptotically the same on X and X^{rot}, where X^{rot} is X rotated by 90 degrees?

Problem 30. Is $\mathrm{cost}_{GG}(X) \le \mathsf{OPT}(X) + O(m)$ for every $X \in [n]^m$?

Problem 31. Is minimum satisfied superset NP-hard for point sets with one point in every row?

Problem 32. Prove or disprove that $\mathcal{W}^1(X) = \Theta\big(\mathsf{MIR}(X)\big)$ or $\mathcal{W}^2(X) = \Theta\big(\mathsf{MIR}(X)\big)$ hold for all $X \in [n]^m$.

Problem 33. Prove or disprove that $\mathsf{MIR}(X) = \Theta\big(\mathsf{OPT}(X)\big)$ for all $X \in [n]^m$.

Chapter 3

Problem 34. Is the cost of PathBalance $m \cdot O(\log n)$ for every (sufficiently long) access sequence $X \in [n]^m$?

Problem 35. Can Theorem 3.20 be strengthened in any of the following ways?

1. Involving in the statement (instead of the sequential access condition) the balance condition, the access lemma, or some other measure of efficiency.

2. Involving in the statement the quantity z (number of zigzags).
3. Relaxing the condition that *every* transformation must create only few leaves.
4. Relaxing the dependence on the monotonicity condition.
5. Relaxing the bound $n^{o(1)}$ to, say, $o(n)$ or $o(|Q|)$.

Problem 36. Does every weakly depth-halving strict online algorithm satisfy the balance condition? More strongly, does the access lemma hold for every such algorithm? How about the sequential access, dynamic finger, lazy finger, and $O(1)$-competitiveness properties?

Problem 37. Find a natural definition of depth-halving that Splay satisfies, and show that it implies the access lemma (or at least the balance condition).

Problem 38. Does every algorithm that satisfies the conditions of Theorem 3.4 have any of the sequential access, dynamic finger, lazy finger, and O(1)-competitiveness properties?

Problem 39. Is there a (natural) non-monotone strict online BST algorithm with access-to-root property that satisfies the balance condition?

Problem 40. If every transformation creates $\Omega(|P|)$ leaves, is the balance condition guaranteed? How about the sequential, dynamic finger, lazy finger, $O(1)$-competitiveness properties?

Problem 41. If every transformation creates $\Omega(|P| - z)$ leaves, where z is the number of zigzags in P, is the balance condition guaranteed? How about the other properties?

Problem 42. Characterize the class of strict online algorithms that satisfy the sequential access property.

Problem 43. Characterize the class of strict online algorithms that satisfy the dynamic finger, lazy finger (if any), or $O(1)$-competitiveness (if any) properties.

Chapter 4

Problem 44. Is there some fixed function $f(\cdot)$ such that for every access sequence $X \in [n]^m$ that avoids an arbitrary fixed pattern $\pi \in S_k$, we have $\mathsf{OPT}(X) \le m \cdot f(k)$? More strongly, does such an upper bound hold for the cost of some online algorithm?

Problem 45. Is the cost of Greedy, Splay, or some other online BST algorithm $O(n)$ on all $\bigwedge$-type permutations of size n with every initial tree?

Problem 46. Find an upper bound on the cost of Splay for every access sequence $X \in S_n$ that avoids an arbitrary pattern $\pi \in S_k$.

Problem 47. Is there a sorting algorithm that can sort in linear time every permutation that avoids an arbitrary fixed pattern?

Problem 48. Is there some function $f(\cdot)$ such that for every $\bigvee_k$-avoiding or $\bigwedge_k$-avoiding sequence $X \in S_n$, it holds that $\mathsf{OPT}(X) \le n \cdot f(k)$? More strongly, can we bound the cost of Splay or Greedy (with or without initial tree) for such inputs?

Chapter 5

Problem 49. Does a converse of Theorem 5.4 hold?

Problem 50. Prove or disprove:

- For any two Rectangulation states $\mathcal{R}_1$ and $\mathcal{R}_2$, we have $d(\mathcal{R}_1, \mathcal{R}_2) = O\big(d'(\mathcal{R}_1, \mathcal{R}_2)\big)$.
- For any two Rectangulation states $\mathcal{R}_1$ and $\mathcal{R}_2$, we have $d'(\mathcal{R}_1, \mathcal{R}_2) = O\big(d'(H, V)\big)$.

List of figures

Bibliography

[1] Eyal Ackerman, Michelle M. Allen, Gill Barequet, Maarten Löffler, Joshua Mermelstein, Diane L. Souvaine, and Csaba D. Tóth. “The Flip Diameter of Rectangulations and Convex Subdivisions”. In: *LATIN 2014*, pp. 478–489 (pages 99–102, 110, 112, 113).

[2] Eyal Ackerman, Gill Barequet, and Ron Y. Pinter. “A bijection between permutations and floorplans, and its applications”. In: *Discrete Applied Mathematics* 154.12 (2006), pp. 1674 –1684 (pages 100, 119).

[3] Eyal Ackerman, Gill Barequet, and Ron Y. Pinter. “On the number of rectangulations of a planar point set”. In: *Journal of Combinatorial Theory, Series A* 113.6 (2006), pp. 1072 –1091 (pages 99, 100).

[4] G. M. Adelson-Velskiĭ and E. M. Landis. “An algorithm for organization of information”. In: *Dokl. Akad. Nauk SSSR* 146 (1962), pp. 263–266 (page 8).

[5] Oswin Aichholzer, Wolfgang Mulzer, and Alexander Pilz. “Flip Distance Between Triangulations of a Simple Polygon is NP-Complete”. In: *Discrete & Computational Geometry* 54.2 (2015), pp. 368–389 (page 9).

[6] Susanne Albers and Marek Karpinski. “Randomized splay trees: Theoretical and experimental results”. In: *Information Processing Letters* 81.4 (2002), pp. 213 –221 (page 26).

[7] Brian Allen and Ian Munro. “Self-Organizing Binary Search Trees”. In: *J. ACM* 25.4 (Oct. 1978), pp. 526–535 (pages 3, 23, 29).

[8] Sanjeev Arora, Elad Hazan, and Satyen Kale. “The Multiplicative Weights Update Method: a Meta-Algorithm and Applications”. In: *Theory of Computing* 8.6 (2012), pp. 121–164 (page 24).

[9] David Arthur. “Fast Sorting and Pattern-avoiding Permutations”. In: *ANALCO 2007*, pp. 169–174 (page 88).

[10] R. Balasubramanian and Venkatesh Raman. “Path Balance Heuristic for Self-Adjusting Binary Search Trees”. In: *FSTTCS 1995*, pp. 338–348 (pages 63, 68).

[11] Jérémy Barbay and Gonzalo Navarro. “On compressing permutations and adaptive sorting”. In: *Theoretical Computer Science* 513 (2013), pp. 109 –123 (pages 79, 87).

[12] Jon Louis Bentley. “Multidimensional Binary Search Trees Used for Associative Searching”. In: *Commun. ACM* 18.9 (Sept. 1975), pp. 509–517 (page 100).

[13] Jon Louis Bentley and Andrew Chi-Chih Yao. “An almost optimal algorithm for unbounded searching”. In: *Information Processing Letters* 5.3 (1976), pp. 82 –87 (page 27).

[14] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. 3rd ed. Springer-Verlag, 2008 (page 100).

[15] Daniel Bienstock and Ervin Győri. “An Extremal Problem on Sparse 0-1 Matrices”. In: *SIAM J. Discrete Math.* 4.1 (1991), pp. 17–27 (page 83).

[16] Avrim Blum, Shuchi Chawla, and Adam Kalai. “Static Optimality and Dynamic Search-optimality in Lists and Trees”. In: *SODA 2002*, pp. 1–8 (pages 25, 26, 37).

[17] Miklós Bóna. “A Survey of Stack-Sorting Disciplines”. In: *Electr. J. Comb.* on.2 (2002) (page 79).

[18] Miklós Bóna. *Combinatorics of Permutations*. CRC Press, Inc., 2004 (page 75).

[19] Prosenjit Bose, Jonathan F Buss, and Anna Lubiw. “Pattern matching for permutations”. In: *Information Processing Letters* 65.5 (1998), pp. 277–283 (pages 75, 80).

[20] Prosenjit Bose, Karim Douïeb, John Iacono, and Stefan Langerman. “The Power and Limitations of Static Binary Search Trees with Lazy Finger”. In: *ISAAC 2014*, pp. 181–192 (pages 26, 28, 31).

[21] Robert Brignall. “A Survey of Simple Permutations”. In: *CoRR* abs/0801.0963 (2008) (pages 80, 81, 96).

[22] Marie-Louise Bruner and Martin Lackner. “The computational landscape of permutation patterns”. In: *CoRR* abs/1301.0340 (2013) (page 75).

[23] Parinya Chalermsook, Mayank Goswami, László Kozma, Kurt Mehlhorn, and Thatchaphol Saranurak. “Greedy Is an Almost Optimal Deque”. In: *WADS 2015*. Preprint URL: http://arxiv.org/abs/1506.08319, pp. 152–165 (pages 84, 88).

[24] Parinya Chalermsook, Mayank Goswami, László Kozma, Kurt Mehlhorn, and Thatchaphol Saranurak. “Pattern-Avoiding Access in Binary Search Trees”. In: *FOCS 2015*. Preprint URL: http://arxiv.org/abs/1507.06953, pp. 410–423 (pages ix, 37, 75, 84, 85, 87, 96, 115).

[25] Parinya Chalermsook, Mayank Goswami, László Kozma, Kurt Mehlhorn, and Thatchaphol Saranurak. “Self-Adjusting Binary Search Trees: What Makes Them Tick?” In: *ESA 2015*. Preprint URL: http://arxiv.org/abs/1503.03105, pp. 300–312 (pages ix, 48, 62).

[26] Parinya Chalermsook, Mayank Goswami, László Kozma, Kurt Mehlhorn, and Thatchaphol Saranurak. “The landscape of bounds for binary search trees”. In: *CoRR* abs/1603.04892 (2016). Preprint URL: http://arxiv.org/abs/1603.04892 (pages 27, 28, 32, 33, 89, 90, 96, 97).

[27] R. Chaudhuri and H. Höft. “Splaying a Search Tree in Preorder Takes Linear Time”. In: *SIGACT News* 24.2 (Apr. 1993), pp. 88–93 (pages 32, 77).

[28] R. Cole. “On the Dynamic Finger Conjecture for Splay Trees. Part II: The Proof”. In: *SIAM Journal on Computing* 30.1 (2000), pp. 44–85 (pages 30, 116).

[29] 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”. In: *SIAM J. Comput.* 30.1 (Apr. 2000), pp. 1–43 (pages 30, 116).

[30] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. 2nd. McGraw-Hill Higher Education, 2001 (pages 3, 77, 87).

[31] Erik D. Demaine, Dion Harmon, John Iacono, Daniel M. Kane, and Mihai Pătraşcu. “The geometry of binary search trees”. In: *SODA 2009*, pp. 496–505 (pages 5, 13, 34, 35, 41, 42, 45, 47, 49–52, 75, 99, 108, 113, 114).

[32] Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Pătraşcu. “Dynamic Optimality - Almost”. In: *SIAM J. Comput.* 37.1 (2007), pp. 240–251 (pages 13, 14, 37, 38, 40).

[33] Erik D. Demaine, John Iacono, Stefan Langerman, and Özgür Özkan. “Combining Binary Search Trees”. In: *ICALP 2013*, pp. 388–399 (pages 26, 90).

[34] Jonathan C. Derryberry, Daniel D. Sleator, and Chengwen C. Wang. “A lower bound framework for binary search trees with rotations”. In: *Tech. Rep. CMU-CS-05-187, Carnegie Mellon Univ.* (2005) (pages 5, 50).

[35] P. Diaconis, R. Graham, and R.L. Graham. *Magical Mathematics: The Mathematical Ideas that Animate Great Magic Tricks*. Princeton University Press, 2011 (page 78).

[36] C.H.J. Edwards. *The Historical Development of the Calculus*. Springer Study Edition. Springer New York, 2012 (page 1).

[37] Amr Elmasry. “On the sequential access theorem and deque conjecture for splay trees”. In: *Theoretical Computer Science* 314.3 (2004), pp. 459 –466 (page 31).

[38] David Eppstein, Elena Mumford, Bettina Speckmann, and Kevin Verbeek. “Area-Universal and Constrained Rectangular Layouts”. In: *SIAM J. Comput.* 41.3 (2012), pp. 537–564 (page 100).

[39] Vladmir Estivill-Castro and Derick Wood. “A Survey of Adaptive Sorting Algorithms”. In: *ACM Comput. Surv.* 24.4 (Dec. 1992), pp. 441–476 (page 87).

[40] Stefan Felsner. *Rectangle and Square Representations of Planar Graphs , in: Pach, J., Thirty Essays on Geometric Graph Theory*. Algorithms and combinatorics. Springer New York, 2012, pp. 213–248 (page 100).

[41] Jacob Fox. “Stanley-Wilf limits are typically exponential”. In: *CoRR* abs/1310.8378 (2013) (page 84).

[42] Kyle Fox. “Upper Bounds for Maximally Greedy Binary Search Trees”. In: *WADS 2011*, pp. 411–422 (pages 36, 59).

[43] Michael L. Fredman. “How good is the information theory bound in sorting?” In: *Theoretical Computer Science* 1.4 (1976), pp. 355 –361 (page 87).

[44] Zoltán Füredi. “The maximum number of unit distances in a convex n-gon”. In: *J. Comb. Theory, Ser. A* 55.2 (1990), pp. 316–320 (page 83).

[45] Zoltán Füredi and Péter Hajnal. “Davenport-Schinzel theory of matrices”. In: *Discrete Mathematics* 103.3 (1992), pp. 233 –251 (pages 76, 84).

[46] Martin Fürer. “Randomized Splay Trees”. In: *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA 1999, pp. 903–904 (page 26).

[47] George F. Georgakopoulos. “Chain-splay trees, or, how to achieve and prove loglogN-competitiveness by splaying”. In: *Inf. Process. Lett.* 106.1 (2008), pp. 37–43 (page 40).

[48] George F. Georgakopoulos and David J. McClurkin. “Generalized Template Splay: A Basic Theory and Calculus”. In: *Comput. J.* 47.1 (2004), pp. 10–19 (pages 32, 36, 55, 57, 62, 68).

[49] Sally Goldman, Adam Kalai, and Santosh Vempala. “Learning Theory 2003 Efficient algorithms for online decision problems”. In: *Journal of Computer and System Sciences* 71.3 (2005), pp. 291 –307 (page 26).

[50] Navin Goyal and Manoj Gupta. “On Dynamic Optimality for Binary Search Trees”. In: *CoRR* abs/1102.4523 (2011) (page 36).

[51] Joachim Gudmundsson, Oliver Klein, Christian Knauer, and Michiel Smid. “Small manhattan networks and algorithms for the earth mover’s distance”. In: *EWCG 2007*. 2007, pp. 174–177 (pages 101, 113–115).

[52] Dion Harmon. “New Bounds on Optimal Binary Search Trees”. PhD thesis. Massachusetts Institute of Technology, 2006 (pages 41, 50, 101, 113, 114).

[53] John Iacono. “In Pursuit of the Dynamic Optimality Conjecture”. English. In: *Space-Efficient Data Structures, Streams, and Algorithms*. Vol. 8066. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 236–250 (pages 4, 19, 21, 24, 39).

[54] John Iacono and Stefan Langerman. “Weighted dynamic finger in binary search trees”. In: *SODA 2016*. Chap. 49, pp. 672–691 (pages 26, 28, 35, 36, 90, 116).

[55] Karel Culik II and Derick Wood. “A note on some tree similarity measures”. In: *Information Processing Letters* 15.1 (1982), pp. 39 –42 (page 8).

[56] S. Kitaev. *Patterns in Permutations and Words*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2011 (page 75).

[57] Martin Klazar. “The Füredi-Hajnal Conjecture Implies the Stanley-Wilf Conjecture”. English. In: *Formal Power Series and Algebraic Combinatorics*. Ed. by Daniel Krob, AlexanderA. Mikhalev, and AlexanderV. Mikhalev. Springer Berlin Heidelberg, 2000, pp. 250–255 (page 76).

[58] Donald E. Knuth. “Optimum binary search trees”. In: *Acta Informatica* 1.1 (1971), pp. 14–25 (page 12).

[59] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997 (page 75).

[60] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. 1998 (pages 1, 12, 87).

[61] Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 1968 (page 75).

[62] László Kozma and Thatchaphol Saranurak. “Binary search trees and rectangulations”. In: *CoRR* abs/1603.08151 (2016). Preprint URL: http://arxiv.org/abs/1603.08151 (page ix).

[63] Marc J. van Kreveld and Bettina Speckmann. “On rectangular cartograms”. In: *Comput. Geom.* 37.3 (2007), pp. 175–187 (page 100).

[64] E. Kushilevitz and N. Nisan. *Communication Complexity*. Cambridge University Press, 2006 (page 100).

[65] André E. Kézdy, Hunter S. Snevily, and Chi Wang. “Partitioning permutations into increasing and decreasing subsequences”. In: *Journal of Combinatorial Theory, Series A* 73.2 (1996), pp. 353 –359 (page 79).

[66] Christos Levcopoulos and Ola Petersson. “Sorting shuffled monotone sequences”. In: *SWAT 1990*. Ed. by John R. Gilbert and Rolf Karlsson, pp. 181–191 (pages 79, 87).

[67] Anna Lubiw and Vinayak Pathak. “Flip distance between two triangulations of a point set is NP-complete”. In: *Comput. Geom.* 49 (2015), pp. 17 –23 (page 9).

[68] Joan M. Lucas. “Canonical forms for competitive binary search tree algorithms”. In: *Tech. Rep. DCS-TR-250, Rutgers University* (1988) (pages 16, 34).

[69] Heikki Mannila. “Measures of Presortedness and Optimal Sorting Algorithms (Extended Abstract)”. In: *ICALP 1984*, pp. 324–336 (page 87).

[70] Adam Marcus and Gábor Tardos. “Excluded permutation matrices and the Stanley-Wilf conjecture”. In: *Journal of Combinatorial Theory, Series A* 107.1 (2004), pp. 153 –160 (pages 76, 84).

[71] Kurt Mehlhorn. *Data Structures and Algorithms*. Monographs in Theoretical Computer Science - An Eatcs Series v. 1. Springer-Verlag, 1984 (pages 32, 87).

[72] Kurt Mehlhorn. "Nearly optimal binary search trees". In: *Acta Informatica* 5.4 (1975), pp. 287–295 (page 12).

[73] Dinesh P. Mehta and Sartaj Sahni. *Floorplan Representation in VLSI, in: Handbook Of Data Structures And Applications*. Chapman & Hall/CRC, 2004 (page 100).

[74] Joseph S. B. Mitchell. "Guillotine Subdivisions Approximate Polygonal Subdivisions: A Simple Polynomial-Time Approximation Scheme for Geometric TSP, k-MST, and Related Problems". In: *SIAM J. Comput.* 28.4 (Mar. 1999), pp. 1298–1309 (page 100).

[75] Alistair Moffat and Ola Petersson. "An Overview of Adaptive Sorting". In: *Australian Computer Journal* 24.2 (1992), pp. 70–77 (page 87).

[76] J. Ian Munro. "On the Competitiveness of Linear Search". English. In: *ESA 2000*, pp. 338–345 (pages 34, 49).

[77] J. Ian Munro and Philip M. Spira. "Sorting and Searching in Multisets". In: *SIAM J. Comput.* 5.1 (1976), pp. 1–8 (page 87).

[78] Gabriel Nivasch. "Improved bounds and new techniques for Davenport–Schinzel sequences and their generalizations". In: *J. ACM* 57.3 (2010) (page 84).

[79] Igor Pak. "History of Catalan numbers". In: *arXiv preprint arXiv:1408.5711* (2014) (pages 1, 7).

[80] Seth Pettie. "Applications of Forbidden 0-1 Matrices to Search Tree and Path Compression-based Data Structures". In: *SODA 2010*, pp. 1457–1467 (pages 31, 48, 84).

[81] Seth Pettie. "Sharp Bounds on Formation-free Sequences". In: *SODA 2015*, pp. 592–604 (page 84).

[82] Seth Pettie. "Splay Trees, Davenport-Schinzel Sequences, and the Deque Conjecture". In: *SODA 2008*, pp. 1115–1124 (pages 48, 84).

[83] Alexander Pilz. "Flip distance between triangulations of a planar point set is APX-hard". In: *Comput. Geom.* 47.5 (2014), pp. 589–604 (page 9).

[84] Lionel Pournin. "The diameter of associahedra". In: *arXiv preprint arXiv:1207.6296* (2014) (page 9).

[85] Vaughan R. Pratt. "Computing Permutations with Double-ended Queues, Parallel Stacks and Parallel Queues". In: *STOC 1973*, pp. 268–277 (page 75).

[86] James H. Schmerl and William T. Trotter. "Critically indecomposable partially ordered sets, graphs, tournaments and other binary relational structures". In: *Discrete Mathematics* 113.1–3 (1993), pp. 191 –205 (page 81).

[87] Raimund Seidel and Udo Adamy. "On the Exact Worst Case Query Complexity of Planar Point Location". In: *J. Algorithms* 37.1 (2000), pp. 189–217 (page 100).

[88] Raimund Seidel and Cecilia R. Aragon. "Randomized Search Trees". In: *Algorithmica* 16.4/5 (1996), pp. 464–497 (pages 9, 32).

[89] Rodica Simion and Frank W. Schmidt. "Restricted Permutations". In: *European Journal of Combinatorics* 6.4 (1985), pp. 383 –406 (page 75).

[90] Daniel D. Sleator and Robert E. Tarjan. "Amortized Efficiency of List Update and Paging Rules". In: *Commun. ACM* 28.2 (Feb. 1985), pp. 202–208 (page 4).

[91] Daniel D. Sleator and Robert E. Tarjan. "Self-adjusting Binary Search Trees". In: *J. ACM* 32.3 (July 1985), pp. 652–686 (pages 3, 28, 30–34, 53, 61, 68, 116).

[92] Daniel D. Sleator, Robert E. Tarjan, and William P. Thurston. “Rotation distance, triangulations, and hyperbolic geometry”. In: *J. Amer. Math. Soc.* 1.3 (1988), pp. 647–681 (page 9).

[93] Sloane N. J. A. *The On-Line Encyclopedia of Integer Sequences*. http://oeis.org. Sequence A000108 (page 7).

[94] Sloane N. J. A. *The On-Line Encyclopedia of Integer Sequences*. https://oeis.org/A006318. Sequence A006318 (pages 81, 100).

[95] Sloane N. J. A. *The On-Line Encyclopedia of Integer Sequences*. https://oeis.org/A001181. Sequence A001181 (page 100).

[96] Richard P. Stanley. *Catalan Numbers*. Cambridge University Press, 2015 (pages 7, 8).

[97] Richard P. Stanley. *Catalan Numbers*. Cambridge University Press, 2015 (page 9).

[98] Richard P. Stanley. *Enumerative Combinatorics. Volume 2*. Cambridge studies in advanced mathematics. Cambridge University Press (page 7).

[99] Ashok Subramanian. “An Explanation of Splaying”. In: *J. Algorithms* 20.3 (1996), pp. 512–525 (pages 36, 55–57, 62, 68, 73).

[100] R. Sundar. “Twists, turns, cascades, deque conjecture, and scanning theorem”. In: *FOCS 1989*, pp. 555–559 (page 31).

[101] Robert Endre Tarjan. “Sequential access in splay trees takes linear time”. In: *Combinatorica* 5.4 (1985), pp. 367–378 (page 31).

[102] Robert Endre Tarjan. “Sorting Using Networks of Queues and Stacks”. In: *J. ACM* 19.2 (Apr. 1972), pp. 341–346 (page 75).

[103] Vincent Vatter. “Permutation classes”. In: *CoRR* abs/1409.5159 (2014) (page 75).

[104] Jean Vuillemin. “A Unifying Look at Data Structures”. In: *Commun. ACM* 23.4 (Apr. 1980), pp. 229–239 (page 9).

[105] Chengwen C. Wang, Jonathan C. Derryberry, and Daniel D. Sleator. “O(Log Log N)-competitive Dynamic Binary Search Trees”. In: *SODA 2006*, pp. 374–383 (page 40).

[106] R. Wilber. “Lower Bounds for Accessing Binary Search Trees with Rotations”. In: *SIAM Journal on Computing* 18.1 (1989), pp. 56–67 (pages 13, 16, 37–39, 115).

[107] Bo Yao, Hongyu Chen, Chung-Kuan Cheng, and Ronald Graham. “Floorplan Representations: Complexity and Connections”. In: *ACM Trans. Des. Autom. Electron. Syst.* 8.1 (Jan. 2003), pp. 55–80 (pages 81, 100, 119).