Dynamic Data Structures: Orthogonal Range Queries and Update Efficiency

Konstantinos Athanasiou Tsakalidis

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

Dynamic Data Structures: Orthogonal Range Queries and Update Efficiency

A Dissertation
Presented to the Faculty of Science
of Aarhus University
in Partial Fulfilment of the Requirements for the
PhD Degree

by Konstantinos A. Tsakalidis
August 11, 2011

Abstract (English)

We study dynamic data structures for different variants of orthogonal range reporting query problems. In particular, we consider (1) the planar orthogonal 3-sided range reporting problem: given a set of points in the plane, report the points that lie within a given 3-sided rectangle with one unbounded side, (2) the planar orthogonal range maxima reporting problem: given a set of points in the plane, report the points that lie within a given orthogonal range and are not dominated by any other point in the range, and (3) the problem of designing fully persistent B-trees for external memory. Dynamic problems like the above arise in various applications of network optimization, VLSI layout design, computer graphics and distributed computing.

For the first problem, we present dynamic data structures for internal and external memory that support planar orthogonal 3-sided range reporting queries, and insertions and deletions of points efficiently over an average case sequence of update operations. The external memory data structures find applications in constraint and temporal databases. In particular, we assume that the coordinates of the points are drawn from different probabilistic distributions that arise in real-world applications, and we modify existing techniques for updating dynamic data structures, in order to support the queries and the updates more efficiently than the worst-case efficient counterparts for this problem.

For the second problem, we present dynamic data structures for internal memory that support different variants of planar orthogonal range maxima reporting queries, and insertions and deletions of points efficiently in the worst case. The presented structures for the pointer machine model improve upon their previous counterparts either by a logarithmic factor in the query or update complexity, or by the fact that they support all the operations efficiently in the worst case. The presented structure for the word-RAM model is the first structure that supports all operations in sublogarithmic worst case time. The results are obtained by a new technique for updating dynamic data structures that is based on partial persistence. The technique is interesting in its own right, as it alleviates significantly the design of algorithms that handle deletions.

For the third problem, we present the currently most efficient fully persistent B-trees, which comprises the implementation of an efficient multi-versioned indexing database. We first present a generic method for making data structures fully persistent in external memory. This method can render any database multi-versioned, as long as its implementation abides by our assumptions. We obtain the result by presenting an implementation of B-trees with worst case efficient update time, and applying our method for full persistence to it.

Abstract (Danish)

Vi studerer dynamiske data strukturer for forskellige varianter af ortogonale range rapporterings query problemer. Specifikt, overvejer vi (1) det planfladet ortogonale 3-sidet range rapporterings problem: given en punktmængde i planen, rapporter punkterne, der ligger indenfor et givet 3-sidet rektangel med en ubundet side, (2) det planfladet ortogonale range maxima rapporterings problem: given en punktmængde i planen, rapporter punkterne, der ligger indenfor en given ortogonale range og der ikke er domineret af andre punkter i ranget, og (3) problemet for at designe fuldt-vedvarende B-træer for ekstern hukommelse. Dynamiske problemer som de ovennævnte opstår i forskellige anvendelser i netværksoptimering, VLSI design, computer grafik og distributeret regning.

I det første problem, præsenterer vi dynamiske data strukturer for intern og ekstern hukommelse, der støtter planflade ortogonale 3-sidet range rapporterings querier, og indsættelser of slettelser af punkter effektivt over gennemsnitstilfælds sekvenser af opdateringsoperationer. Data strukturen for ekstern hukommelse har anvendelser i afgrænsede og tidsmæssige databaser. Specifikt, antager vi at punkternes kooordinater er udtrukket fra forskellige probabilistiske distributioner, som forekommer i dagligdags anvendelser, og vi modificer eksisterende teknikker for at opdatere dynamiske data strukturer med det mål at støtte operationerne mere effektivt end problemets værste-tilfælds modstykker.

I det andet problem, præsenterer vi dynamiske data strukturer for intern hukommelse, der støtter planflade ortogonale range maxima rapporterings querier, og indsættelser og slettelser af punkter effektivt i det værste tilfælde. De præsenterede strucktrurer for pointer-maskin modelen forbedrer de tidligere løsninger enten ved en logaritmisk faktor i queriets eller opdaterings kompleksiteten, eller ved at de støtte alle operationer effektivt i det værste tildfælde. De præsenterede strukturer for word-RAM modelen er de første strukturer, der støtter alle operarioner i underlogaritmiske værste tildfældes tid. Resultaterne opnåes ven en ny teknik for at opdatere dynamiske data strukturer, der er baseret pådelvis vedvarenhed. Teknikken er i sig selv interessant, fordi den væsentligt letter designet af algoritmer, der behandler slettelser.

I det tredige problem, præsenterer vi de nuværende mest effektive fuldt-vedvarende B-træer, der består implementationen af en effektiv multi-version indeks database. Først præsenter vi en general metode for at lave data strukturer fuldt-vedvarende i ekstern hukommelse. Metoden gør hver database multiversioneret, sålænge den tilfredsstiller vores antagelser. Vi opnår resultatet ved at præsenter en implementation af B-træer med effektiv værste-tilfælds opdateringstid, og ved at anvende vores metode for fuldt-vedvarenhed i den.

Abstract (Greek)

Μελετούμε δυναμικές δομές δεδομένων για διάφορες παραλλαγές προβλημάτων ερωτήσεων αναφοράς ορθογώνιου εύρους. Συγκεκριμένα, θεωρούμε (1) το πρόβλημα ερωτήσεων αναφοράς δισδιάστατου ορθογώνιου 3-πλευρου εύρους: έστω ένα σύνολο σημείων στο επίπεδο, ανάφερε τα σημεία που βρίσκονται εντός ενός δεδομένου 3-πλευρου ορθογώνιου παραλληλόγραμμου με μια απεριόριστη πλευρά, (2) το πρόβλημα ερωτήσεων αναφοράς μεγίστων δισδιάστατου ορθογώνιου εύρους: έστω ένα σύνολο σημείων στο επίπεδο, ανάφερε τα σημεία που βρίσκονται εντός ενός δεδομένου ορθογώνιου παραλληλόγραμμου εύρους τα οποία δεν είναι υπερκείμενα από άλλα σημεία στο εύρος, και (3) το πρόβλημα του σχεδιασμού πλήρως διαχρονικών B-δέντρων για την εξωτερική μνήμη. Δυναμικά προβλήματα όπως τα προαναφερθέντα εμφανίζονται σε εφαρμογές βελτιστοποίησης δικτύων, σχεδιασμού VLSI, computer-γραφικής και κατανεμημένου υπολογισμού.

Για το πρώτο πρόβλημα, παρουσιάζουμε δυναμικές δομές δεδομένων για εσωτερική και εξωτερική μνήμη, που υποστηρίζουν αποδοτικά ερωτήσεις αναφοράς δισδιάστατου ορθογώνιου 3-πλευρου εύρους, και εγγραφές και διαγραφές σημείων υπό αλληλουχίες ενημέρωσης μέσης περίπτωσης. Οι δομές δεδομένων εξωτερικής μνήμης βρίσκουν εφαρμογές σε περιοριστικές και χρονικές βάσεις δεδομένων. Συγκεκριμένα, υποθέτουμε ότι οι συνισταμένες των σημείων ακολουθούν διάφορες πιθανοτικές κατανομές που εμφανίζονται σε πραγματικές εφαρμογές, και μεταβάλλουμε υπάρχουσες τεχνικές ενημέρωσης δυναμικών δομών δεδομένων, ώστε να υποστηρίζονται οιερωτήσεις και οιενημερώσεις πιο αποδοτικά από τις δομές χειρότερης περίπτωσης για αυτό το πρόβλημα.

Για το δεύτερο πρόβλημα, παρουσιάζουμε δυναμικές δομές δεδομένων για εσωτερική μνήμη που υποστηρίζουν αποδοτικά διάφορες παραλλαγές ερωτήσεων αναφοράς μεγίστων δισδιάστατου ορθογώνιου εύρους, και εγγραφές και διαγραφές σημείων στην χειρότερη περίπτωση. Οι παρουσιαζόμενες δομές για το μοντέλο pointer machine βελτιώνουν τις προϋπάρχουσες δομές είτε κατά ενα λογαριθμικό παράγοντα στην πολυπλοκότητα ερωτήσεων ή ενημερώσεων, είτε επειδή υποστηρίζουν όλες τις λειτουργίες αποδοτικά στην χειρότερη περίπτωση. Οι παρουσιαζόμενες δομές για το μοντέλο word-RAM είναι οι πρώτες που επιτυγχάνουν υπολογαριθμική πολυπλοκότητα χειρότερης περίπτωσης. Τα αποτελέσματα επιτυγχάνονται μέσω μιας νέας τεχνικής ενημερώσεως δυναμικών δομών δεδομένων που βασίζεται στην μερική διαχρονικότητα. Η τεχνική είναι ενδιαφέρουσα από μόνη της, καθώς διευκολύνει σημαντικά τον σχεδιασμό αλγορίθμων διαγραφής.

Για το τρίτο πρόβλημα, παρουσιάζουμε τα επί του παρόντος πιο αποδοτικά πλήρως διαχρονικά B-δέντρα, που αποτελούν υλοποίηση μιας αποδοτικής βάσης δεδομένων πολλών εκδοχών. Πρώτα παρουσιάζουμε μία γενική μέθοδο που καθιστά τις δομές δεδομένων πλήρως διαχρονικές στην εξωτερική μνήμη. Η μέθοδος αυτή καθιστά κάθε βάση δεδομένων πολυ-εκδοχική, όταν η υλοποίηση τησικανοποιεί τις προϋποθέσεις μας. Επιτυγχάνουμε το αποτέλεσμα παρουσιάζοντας μια υλοποίηση B-δέντρων με αποδοτική ενημέρωση χειρότερης περίπτωσης, και εφαρμόζοντας την μέθοδό μας σε αυτήν.

Preface

This dissertation is devoted to three aspects of dynamic data structures and algorithms: efficient update operations, persistent data structures and orthogonal range queries. Motivated by two different problems that arise in the field of Dynamic Geometric Orthogonal Range Searching, we study how to use existing techniques and we propose new techniques for updating dynamic data structures efficiently over average and worst case sequences of update operations. Motivated by a problem that arises in the field of Multi-Versioned Databases, we study how to make data structures for external memory persistent.

Structure of the dissertation

- **Chapter 1 (Introduction).** This chapter provides a detailed overview of the subject of this dissertation. In particular, the three problems considered in the following chapters are formally defined, the relevant literature is surveyed, and our contributions are briefly stated. Moreover, Section 1.2 provides the necessary preliminaries and the terminology for the remainder of the dissertation.

- **Chapter 2 (Dynamic Planar 3-Sided Range Reporting Queries)** The content of this chapter is based on two papers. In particular, Sections 2.2 and 2.3 are part of a paper published in the Proceedings of the 13th International Conference on Database Theory (ICDT 2010) [KPS+10]. It is entitled "Efficient Processing of 3-Sided Range Queries with Probabilistic Guarantees", and it is joint work with Alexis Kaporis, Apostolos Papadopoulos, Spyros Sioutas and Kostas Tsichlas. Section 2.4 is part of a paper published in the Proceedings of the 20th International Symposium on Algorithms and Computation (ISAAC 2009) [BKS+09]. It is entitled "Dynamic 3-Sided Planar Range Queries with Expected Doubly Logarithmic Time", and it is joint work with Gerth Stølting Brodal, Alexis Kaporis, Spyros Sioutas and Kostas Tsichlas.

  In this chapter, we study dynamic data structures for internal and external memory that support planar orthogonal 3-sided range reporting queries and updates efficiently over an average case sequence of update operations. In Section 2.2 we present a dynamic data structure for internal memory and its variant for external memory, that support update operations more efficiently than the previous worst-case efficient structures, under the assumption that the coordinates of the points are drawn from a continuous but unknown probabilistic distribution. In Section 2.3 we present a dynamic data structure for internal memory and its variant for external memory, that support both queries and updates more efficiently than previous worst-case efficient structures, under the assumption that the x-coordinates of the points are drawn from a smooth probabilistic distribution, and the y-coordinates are drawn from a class of probabilistic distributions that exhibit unbounded density. In Section 2.4 we waive the assumption on the y-coordinates of the points. We present a dynamic data structure for internal memory and its variant for external memory, that support both queries and updates more efficiently than previous worst-case efficient structures, only under the assumption that the x-coordinates of the points are drawn from a smooth probabilistic distribution. Finally, we propose some open problems.

- **Chapter 3 (Dynamic Planar Range Maxima Queries)** The content of this chapter is based on a paper published in the Proceedings of the 38th International Colloquium on Automata, Languages and Programming (ICALP 2011) [BT11]. It is entitled "Dynamic Planar Range Maxima Queries", and it is joint work with Gerth Stølting Brodal.

  In this chapter, we study dynamic data structures for internal memory that support different variants of planar orthogonal range maxima reporting queries and updates efficiently in the worst case. In Section 3.2 we present a dynamic data structure for the pointer machine model that maintains the maximal points of a planar pointset, and supports insertions and deletions in logarithmic worst case time. We obtain the result by a new technique for updating dynamic data structures that is based on partial persistence, which for our problem allows for deleting a point in $O(\text{insertion})$ time. Moreover, we show that the structure supports in the the same complexity, the more general query of reporting the maximal points among the points that lie within a given 3-sided orthogonal rectangle unbounded from above. In Section 3.3 we adapt the above the structure to the word-RAM model, and obtain the first dynamic data structure for this model that supports all the operations in sublogathmic worst case time. In Section 3.4 we present a dynamic data structure for the pointer machine model that supports reporting the maximal points among the points that lie within a given 4-sided orthogonal rectangle. This improves previous results by a logarithmic factor. Finally, we propose some open problems.

- **Chapter 4 (Fully Perstistent B-Trees)** The content of this chapter is based on a paper very recently submitted to the ACM-SIAM Symposium on Discrete Algortithms 2012 (SODA 2012) [BSTT12]. It is entitled "Fully Persistent B-Trees", and it is joint work with Gerth Stølting Brodal, Spyros Sioutas and Kostas Tsichlas.

  In this chapter, we study the problem of making B-trees fully persistent in external memory. In Section 4.2 we present a generic method that makes any ephemeral pointer-based data structure fully persistent in the I/O model and is efficient over a sequence of worst case update operations, as long as every node of the ephemeral structure occupies a constant number of blocks on disk. We present the result in form of an interface of fundamental operations to be executed by the implementation of the ephemeral structure. In Section 4.3 we present an implementation of (a, b)-trees, where every update operation causes at most a constant number of modifications to the tree. In Section 4.4 we obtain the currently most efficient implementation of a fully persistent B-trees, by applying our method for full persistence in external memory to our implementation of (a, b)-trees. Finally, we propose some open problems.

Acknowledgements I would like to thank my advisor Gerth Stølting Brodal and my professor Lars Arge for giving me the opportunity to study Computer Science in the most ideal environment. I especially thank Gerth, whose strong intuition in the field of data structures and algorithms taught me how to abstract and simplify complicated knowledge. A life's lesson that does not only pertain to science.

I especially thank Lars Arge and Else Magård, who are MADALGO's soul, for making my stay in Denmark so easy. I also thank all the Ph.D. students and Post-docs that were in MADALGO during my studies, especially Lasse Deleuran, Casper Kejlberg-Rasmussen, Freek van Walderveen, Kasper Green Larsen, Nodari Sitchinava, Mohammad Ali Abam, Srinivasa Rao and Peyman Afshani. My stay in Denmark could not have been so exciting, without my best friends Pooya Davoodi, Dimitris Katsikas and Rikke Jordal Hansen.

I would like to thank Ian Munro, Patrick Nicholson, Meng He, Matthew Skala and Shahin Kamali for the fruitful collaboration we had during my short stay in the University of Waterloo, as well as Brad Nickerson, Alex Lopez Ortiz, Stephane Durocher, Timothy Chan, and the other members of the lively Computer Science community of Canada that I had the opportunity to meet.

I thank Peter Bro Miltersen, Rolf Fagerberd and Jeff Erickson for accepting to be in my Ph.D. committee, as well as Anders Møller for auditoring my midterm evaluation.

I especially thank Athanasios Tsakalidis, my father, and Paul Spirakis, my former advisor in the University of Patras, for introducing me to Computer Science and for giving me early on the right directions to pursue a scientific carreer of such a demanding level. Also special thanks to my colleagues and former tutors Kostas Tsichlas, Spyros Sioutas and Christos Makris. Interacting with them is always prolific and didactic. Also I would like to thanks my coauthors Alexis Kaporis and Apostolos Papadopoulos.

Finally, I thank my mother Aglaia Liopa-Tsakalidi and my brother Dimitris Tsakalidis who have encouraged and supported me all these years.

Konstantinos A. Tsakalidis,
Aarhus, Denmark, August 11, 2011.

Contents

Chapter 1

Introduction

This dissertation studies the problems of designing average case efficient fully dynamic data structures for Planar 3-Sided Range Reporting in the internal and the external memory (Chapter 2), designing worst case efficient fully dynamic data structures for Planar Range Maxima Reporting in the internal memory (Chapter 3), and designing I/O-efficient Fully Persistent B-trees for the external memory (Chapter 4). In the following, we introduce the field of dynamic data structures in order to define the terminology.

Research on design and analysis of *dynamic data structures* studies how to process and maintain volatile data efficiently. Volatile data arises in applications from network optimization, VLSI layout design, computer graphics, distributed computing and other fields, is usually expressed as objects that have some combinatorial relationship to each other. A dynamic data structure stores a set of input objects in memory and supports the *update operation*. This operation modifies the stored data either by modifying an already stored object, or by *inserting* a new object in the stored set, or by *deleting* a stored object from the memory. The data structure moreover supports the *query operation* that processes the stored objects in order to answer questions about them. For example, the objects may be points on the plane, where the relationship of two points is their distance, and the query asks for the two points in the set with minimum distance. An efficient dynamic data structure minimizes the space that the objects occupy in memory and the time to support the query and the update operations.

Consider the outcome of an update operation to a dynamic data structure to be a new *version* of the structure. The new version contains the unmodified objects, the modified objects and the inserted objects, and does not contain the objects removed by the operation. A dynamic data structure is called *persistent* if all the versions of the structure are maintained as update operations are performed to it. Ordinary data structures, which do not have this capability are called *ephemeral*. The problem of making an ephemeral dynamic data structure persistent can be seen as the problem of designing another dynamic data structure, called *persistent mechanism*, that assumes the ephemeral structure as an input object. The supported query and update operations, respectively query and update a particular version of the structure.

The field of Computational Geometry interprets the input objects as geometric objects, such as points, lines, polygons, hyper-planes, etc., that lie in the d-dimensional Euclidean space $\mathbb{R}^d$. Problems in this field usually ask for computing a geometric property of objects with high complexity. For example the problem of Line Segment Intersection, asks to report all the intersections that occur in a given set of line segments. The problem is called *static* when the objects cannot be modified. Dynamic data structures with lower complexity are useful tools for algorithms that solve such static problems. Moreover, persistent data structures are usually used for preprocessing the input objects. In this setting the sequence of update operations is known in advance, namely the updates are *offline*, in contrast to an online setting. In many applications, the update operations are often only insertions or only deletions, in which case it suffices to use a *semi-dynamic* data structure.

Problems that arise in Geometric Range Searching, a fundamental subfield of computational geometry, ask to store and preprocess a set of geometric objects, such that *range queries* can be supported efficiently. Range queries ask for the some information about the objects that intersect a given query range. The range may be a line, a hyper-rectangle, a simplex, a sphere, etc. The solution to geometric range searching problems is given by a static data structure that supports efficiently the range query, occupies minimal space and can be preprocessed efficiently. Static data structures do not support update operations. However, the fact that the data is immutable does not mean that the static data structure may not be modified itself. Problems that arise in Dynamic Geometric Range Searching moreover ask for insertions and deletions of objects. Solutions to such problems are given by *fully dynamic data structures* that support range queries and operate in an *online setting*, namely when the update operations are not known in advance and they are executed upon request.

A database stores and processes large amounts of data, while maintaining them under frequent updates. In practice, large amounts of data means that the data is too big to be stored in *internal memory*, and thus it is stored in *external memory*, which may be a hard disk or some other means of storage with slow data access. Problems that arise in the field of Databases can be usually expressed as geometric range searching problems. Therefore *I/O-efficient data structures*, namely data structures optimized for external memory, that solve such problems, comprise also an implementation of databases that support range queries. A database is updated by means of *transactions* that modify the stored data. Dynamic I/O-efficient data structures implement databases that can be updated in an online setting. Moreover, persistent I/O-efficient data structures implement databases that maintain the history of modifications that the transactions impose to the stored data, or also the history of modifications that occur to the database itself. This finds applications in the subfield of Multi-Versioned and Time-Evolving Databases.

1.1 Dynamic Range Searching

Here we present the problems considered in this dissertation, survey the relevant literature, and state our contributions. All considered problems are of variants of Dynamic Orthogonal Range Reporting problems. The input is a set of n points in d-dimensional Euclidean space $\mathbb{R}^d$. A fundamental problem in this field asks for dynamic data structures that support *d-dimensional orthogonal range reporting queries*. The query reports the points that lie within a given d-dimensional axis-aligned orthogonal hyper-rectangle $[\ell_1, r_1]\times[\ell_2, r_2]\times\ldots\times[\ell_d, r_d]$, where $\forall i \in [1, d]\ell_i \leq r_i$. For $d = 2$, the queries are *planar orthogonal 4-sided rectangles* $[x_\ell, x_r]\times[y_b, y_t]$.

1.1.1 Planar Orthogonal 3-Sided Range Reporting Queries

In Chapter 2 we consider the problem of designing dynamic data structures for internal and external memory that support planar orthogonal 3-sided range reporting queries, and are efficient in the average case. Worst case sequences of operations do not usually arise in real-word applications, which motivates us to study the average case complexity of the problem. For this reason we assume that the x- and y-coordinates of the points are drawn from a probabilistic distribution. *Planar orthogonal 3-sided range reporting queries* ask to report the points that lie within a given orthogonal axis-aligned rectangle $[x_\ell, x_r]\times]-\infty, y_t]$ where $x_\ell \leq x_r$. 3-sided queries are also called *grounded*, since one side of the rectangle extends to infinity. We refer to these queries simply as *3-sided range queries*. Figure 1.1 shows an example of a 3-sided range query. Planar orthogonal 3-sided range reporting queries find applications in constraint and temporal indexing databases [KRVV96], and in databases for uncertain data [CXP+04].

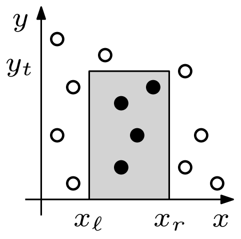

Figure 1.1: The planar orthogonal 3-sided range reporting query $[x_\ell, x_r]\times]-\infty, y_t]$ is shown in gray. Black points are reported.

Table 1.1: Asymptotic bounds for planar orthogonal 3-sided planar range reporting in the pointer machine model. The update time is not stated for static structures. The number of points in the structure is n, the universe of the x-coordinates is $[U]$, and the size of the query output is t.

	Space	Query Time	Update Time
[FMNT87]	$n + U$	$\log\log n + t$	-
[McC85]	n	$\log n + t$	$\log n$

Table 1.2: Asymptotic bounds for planar orthogonal 3-sided planar range reporting in the word-RAM model. The update time is not stated for static structures. The number of points in the structure is n, the universe of the x-coordinates is $[U]$, and the size of the query output is t.

	Space	Query Time	Update Time
[ABR00]	$n+U$	$1+t$	-
[SMK$^+$04]	$n+U$	$1+t$	-
[Wil00]	n	$\frac{\log n}{\log\log n}+t$	$\frac{\log n}{\log\log n}$
[Wil00] [a]	n	$\frac{\log n}{\log\log n}+t$	$\sqrt{\log n}$ [f]
[Mor06]	n	$\frac{\log n}{\log\log n}+t$	$\log^{\delta} n, 0 \leq \delta < 1$
Section 2.2 [b] [e]	n	$\log n + t$	$\log\log n$ [g]
Section 2.3 [c] [e]	n	$\log\log n + t$ [g]	$\log\log n$ [g]
Section 2.4 [d] [e]	n	$\log\log n + t$ [g]	$\log\log n$ [h]

[a]No assumption is posed on the input.
[b]x- and y-coordinates from an unknown μ-random distribution.
[c]x-coordinates from a smooth distribution, y-coordinates from restricted distribution.
[d]x-coordinates from a smooth distribution, y-coordinates from arbitrary distribution.
[e]Deletions are uniformly random over the inserted points.
[f]Expected time.
[g]Expected time with high probability.
[h]Expected amortized time.

Table 1.3: Asymptotic bounds for planar orthogonal 3-sided planar range reporting in the I/O model. The update I/Os is not stated for static structures. The number of points in the structure is n, the size of the query output is t and the size of the block is B.

	Space	Query I/Os	Update I/Os
[IKO88]	n/B	$\log n + t/B$	-
[BG90]	n/B	$\log_B n + t/B$	-
[KRVV96]	n/B	$\log_B n + t/B + \log B$	-
[RS94]	$\frac{n \log B}{B} \log\log B$	$\log_B n + t/B$	$\log n \log B$
[SR95]	n/B	$\log_B n + t/B + \log^* B$	$\log_B n + \frac{\log_B^2 n}{B}$ [h]
[ASV99]	n/B	$\log_B n + t/B$	$\log_B n$ [h]
Section 2.2 [a] [d]	n/B	$\log_B n + t/B$	$\log_B \log n$ [f]
Section 2.3 [b] [d]	n/B	$\log_B \log n + t/B$ [e]	$\log_B \log n$ [f]
Section 2.4 [c] [d]	n/B	$\log \log_B n + t/B$ [e]	$\log_B \log n$ [g]

[a]x- and y-coordinates from an unknown μ-random distribution.
[b]x-coordinates from smooth distribution, y-coordinates from restricted distribution.
[c]x-coordinates from smooth distribution, y-coordinates from arbitrary distribution.
[d]Deletions are uniformly random over the inserted points.
[e]Expected I/Os with high probability.
[f]Expected amortized I/Os with high probability.
[g]Expected amortized I/Os.
[h]Amortized I/Os.

Internal Memory By n we denote the number of stored points and by t the size of the query's output. The implementations of the data structure for internal memory presented in Sections 2.2 and 2.3 contain *priority search trees* [McC85] that support 3-sided range queries in $O(\log n{+}t)$ worst case time and updates in $O(\log n)$ worst case time, using $O(n)$ space. In the word-RAM model, Willard [Wil00] presents a dynamic data structure, based on fusion trees [FW93], that supports 3-sided range queries in $O\left(\log n/\log\log n + t\right)$ worst case time, using $O(n)$ space. Updates are supported in $O\left(\log n/\log\log n\right)$ worst case time, or $O(\sqrt{\log n})$ randomized time, where the randomization comes from the data structure and not from the distribution of the points. In [Mor06] the worst case update time is further improved to $O(\log^{\delta} n)$, for constant $\delta{<}1$.

Fries et al. [FMNT87] present a static data structure for the pointer machine that supports 3-sided range queries in $O(\log\log n{+}t)$ time and uses $O(n{+}U)$ space and preprocessing time. They assume that the the x-coordinates of the points lie on a *grid*, namely they are integers from the set $[U]{=}\{0,\ldots,U-1\}$. In the word-RAM model with word size $w{\geq}\log n$, the x-coordinates can be considered as lying on the grid. Alstrup et al. [ABR00, Theorem 4] present a static data structure for the word-RAM model that supports 3-sided range queries in $O(1{+}t)$ worst case time, using $O(n{+}U)$ space and preprocessing time, when the x-coordinates lie on the grid and the y-coordinates are from $\mathbb{R}$. The data structure for internal memory presented in Section 2.3 uses the *modified priority search tree* of Sioutas et al. [SMK$^+$04], that supports the same operations in the same complexities as [ABR00]. Tables 1.1 and 1.2 summarize the data structures for internal memory that support 3-sided range queries.

External Memory By B we denote the size of a block in the I/O model. Path-caching [RS94] is a generic technique to externalize data structures for internal memory. Applying it to the priority search tree yields a data structure that supports 3-sided range queries in $O(\log_B n + t/B)$ I/Os and updates in $O(\log n \log B)$ I/Os, using $O(n\frac{\log B \log\log B}{B})$ blocks. *P-range trees* [SR95] support 3-sided range queries in $O(\log_B n{+}t/B{+}\log^* B)$ I/Os and updates in $O(\log_B n{+}\log^2_B n/B)$ amortized I/Os, using $O(n/B)$ blocks. The structures for external memory in Sections 2.2 and 2.3 contain *external priority search trees* [ASV99] that support 3-sided range queries in $O(\log_B n{+}t/B)$ I/Os and updates in $O(\log_B n)$ amortized I/Os, using $O(n/B)$ blocks. I/O-efficient static data structures support 3-sided range queries in $O(\log n{+}t/B)$ I/Os [IKO88], $O(\log_B n{+}t/B)$ I/Os [BG90], and $O(\log_B n{+}t/B{+}\log B)$ I/Os [KRVV96], respectively, using $O(n/B)$ blocks. The structure for external memory in Section 2.3 uses *modified external priority search trees* [KPS$^+$10] that supports 3-sided range queries in $O(t/B{+}1)$ expected I/Os with high probability, using $O(n/B)$ blocks and preprocessing I/Os. Table 1.3 summarizes the data structures for external memory that support 3-sided range queries.

Our Contributions In Chapter 2 we present three dynamic data structures for the word-RAM model that support 3-sided range reporting queries and updates efficiently in the average case, using linear space. We moreover adapt the three structures to the I/O model. We assume that the coordinates of the inserted points are drawn continuously from a μ-random distribution. Moreover we assume that every stored point is deleted with equal probability [Knu77]. All presented data structures are deterministic and the expectation is with respect to the input distribution.

We achieve different bounds by assuming different distributions and adapting the structures appropriately. In particular, in Section 2.2 we present a dynamic data structure for the word-RAM model that supports 3-sided range queries in $O(\log n + t)$ worst case time, using $O(n)$ space. It supports updates in $O(\log \log n)$ expected time with high probability, when both the x- and y-coordinates are drawn continuously from a continuous unknown μ-random distribution. In Section 2.2 we also present the I/O-efficient variant of the above data structure that supports 3-sided queries in $O(\log_B n+t/B)$ worst case I/Os, using $O(n/B)$ blocks of space. It support update operations in $O(\log_B \log n)$ expected amortized I/Os with high probability, under the same assumptions.

In Section 2.3 we present a dynamic data structure for the word-RAM model that supports 3-sided range queries in $O(\log \log n + t)$ expected time with high probability and supports updates in $O(\log \log n)$ expected time with high probability, using $O(n)$ space, when the x-coordinates are drawn continuously from a (n^α, n^δ)-smooth distribution [MT93], for constants $0 < \alpha, \delta < 1$, and the y-coordinates are continuously drawn from a distribution of the restricted class. In Section 2.3 we also present the I/O-efficient variant of the above data structure that supports 3-sided range queries in $O(\log_B \log n + t/B)$ expected I/Os with high probability and supports updates in $O(\log_B \log n)$ expected amortized I/Os with high probability, using $O(n/B)$ blocks of space, when the x-coordinates of the points are continuously drawn from a $(n/(\log \log n)^{1+\epsilon}, n^{1/B})$-smooth distribution, for a constant $0 < \epsilon$, and the y-coordinates are continuously drawn from a distribution of the restricted class.

In Section 2.4 we present a dynamic data structure for the word-RAM model that supports 3-sided range queries in $O(\log \log n + t)$ expected time with high probability and supports updates in $O(\log \log n)$ expected amortized time, using $O(n)$ space, when the x-coordinates of the points are continuously drawn from a (n^α, n^δ)-smooth distribution, for constants $0 < \alpha, \delta < 1$, and when the y-coordinates are continuously drawn from any arbitrary distribution. In Section 2.4 we also present the I/O-efficient variant of this structure that supports 3-sided range queries in $O(\log \log_B n+t/B)$ expected I/Os with high probability and supports updates in $O(\log_B \log n)$ expected amortized I/Os, using $O(n/B)$ blocks of space, when the x-coordinates of the points are continuously drawn from a $(n/(\log \log n)^{1+\epsilon}, n^{1/B})$-smooth distribution, for a constant $0 < \epsilon$, and the y-coordinates are continuously drawn from any arbitrary distribution.

1.1.2 Planar Orthogonal Range Maxima Reporting Queries

In Chapter 3 we consider the problem of designing dynamic data structures for the pointer machine and the word-RAM model that support planar orthogonal 3-sided and 4-sided range maxima reporting queries and updates efficiently in the worst case. Given a set n points in $\mathcal{R}^d$, a point $p = (p_1, \ldots, p_d)$ *dominates* another point $q = (q_1, \ldots q_d)$ if $p_i \geq q_i$ holds for all $i \in [d]$. The m points that are not dominated by any other point in the set are called *maximal*. The problem of computing all maximal point is known in the field of Databases as the *skyline* problem [BKS01]. In database applications, maximal points model objects that are "preferrable" in all their attributes.

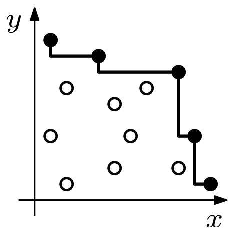

Figure 1.2: The staircase of a planar pointset. Black points are maximal.

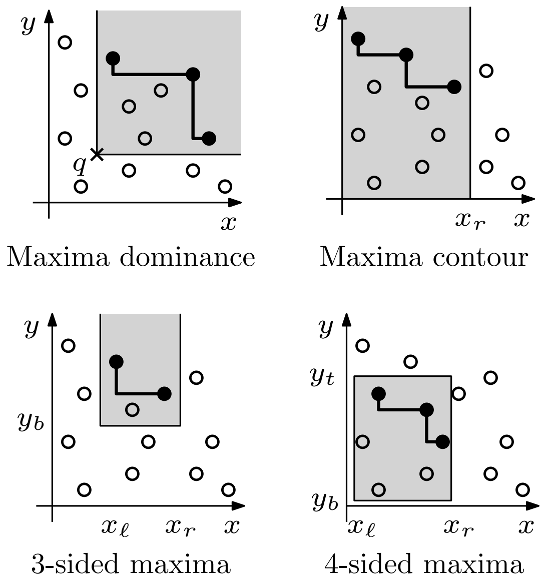

Figure 1.3: Variants of planar orthogonal range maxima reporting queries are shown in gray. Black points are reported.

The maximal points of a static set of n d-dimensional points can be computed in optimal $O(n \log^{d-2} n)$ worst case time for $d \geq 3$ [KLP75]. When $d{=}n$, the maximal points can be computed in $O(n^{2.688})$ worst case time [Mat91]. For $d{=}2$ the maximal points are also called the *staircase* of the set since when they are sorted by increasing x-coordinate they are also sorted by decreasing y-coordinate. Figure 1.2 shows the staircase of a planar pointset. The staircase can be computed in $O(n \log n)$ worst case time [KLP75], and in $O(n)$ worst case time if the points are already sorted by x-coordinate.

The problem of designing dynamic data structures that maintain the set of maximal points under insertions and deletions of points in a set has been considered in the pointer machine model [OvL81,FR90,Jan91,dFGT97,Kap00]. The data structures support several queries, such as reporting all m maximal points (*all maxima reporting queries*), reporting the t maximal points that dominate a given query point (*maxima dominance reporting queries*), reporting the t maximal points among the points that lie to the left of a given vertical line (*maxima contour reporting queries*), and deciding whether a given query point lies above or below the staircase (*maxima emptiness queries*). The variants of range maxima queries are shown in Figure 1.3.

Table 1.4: Asymptotic bounds for range maxima reporting queries in the pointer machine model. “Dominance” queries: Report the maximal points that dominate the query point. “Contour” queries: Report the maximal points among the points that lie to the left of a given vertical query line. All structures occupy $O(n)$ space and report all maximal points in $O(m)$ time. The time every structure decides whether a point is maximal or not, is the time of the dominance query, when $t = 0$.

	Dominance	Contour	Insertion	Deletion
[OvL81]	$\log m + t$	-	$\log^2 n$	$\log^2 n$
[FR90]	$\log n + t$	$\min\{\log^2 n{+}t, (t{+}1)\log n\}$	$\log n$	$\log^2 n$
[Jan91]	$\log n + t$	$\log n + t$	$\log n$	$\log^2 n$
[dFGT97] [a]	$\log m + t$	-	$\log n$	$\log n$
[Kap00]	$\log n + t$ [b]	-	$\log n$	$\log n$
Section 3.2	$\log n + t$	$\log n + t$	$\log n$	$\log n$
Section 3.3 [c]	$\frac{\log n}{\log\log n} + t$	$\frac{\log n}{\log\log n} + t$	$\frac{\log n}{\log\log n}$	$\frac{\log n}{\log\log n}$

[a]Only boundary updates are supported.
[b]Amortized time.
[c]For the word-RAM model.

Table 1.5: Asymptotic bounds for rectangular visibility queries in the pointer machine model. The number of points in the structure is n and the size of the query output is t.

	Space	Query Time	Insertion Time	Deletion Time
[OW88]	$n \log n$	$\log^2 n + t$	$\log^2 n$	$\log^3 n$
[OW88]	$n \log n$	$(1+t)\log n$	$\log^2 n$	$\log^2 n$
Section 3.4	$n \log n$	$\log^2 n + t$	$\log^2 n$	$\log^2 n$

Overmars and van Leeuwen [OvL81] present a dynamic data structure that supports all maxima reporting queries in $O(m)$ worst case time, and maxima emptiness queries in $O(\log m)$ worst case time, using $O(n)$ space. The insertion and deletion of an arbitrary point is supported in $O(\log^2 n)$ worst case time. Frederickson and Rodger [FR90] present a dynamic data structure that supports all maxima reporting queries in $O(m)$ worst case time, maxima emptiness queries in $O(\log n)$ worst case time, maxima dominance reporting queries in $O(\log n+t)$ worst case time and maxima contour reporting queries in $O(\min\{\log^2 n + t, (t + 1) \log n\})$ worst case time, using $O(n)$ space. The worst case insertion time is $O(\log n)$ and the worst case deletion time is $O(\log^2 n)$. Janardan [Jan91] presents a dynamic data structure that supports all maxima reporting queries in $O(m)$ worst case time, maxima emptiness queries in $O(\log n)$ worst case time, maxima dominance reporting queries and maxima contour reporting queries in $O(\log n + t)$ worst case time, using $O(n)$ space. The worst case insertion time is $O(\log n)$, and the worst case deletion time is $O(\log^2 n)$. D'Amore et al. [dFGT97] improve the worst case deletion time by assuming *boundary updates*, namely that the updated point has the maximum or minimum x-coordinate among the points in the set. They present a dynamic data structure that supports insertions and deletions of points in $O(\log n)$ worst case time. It supports all maxima reporting queries in $O(m)$ worst case time, maxima emptiness queries in $O(\log m)$ worst case time, and uses $O(n)$ space. Kapoor [Kap00] presents a dynamic data structure that supports insertions and deletions of arbitrary points in $O(\log n)$ worst case time. The structure supports maxima emptiness queries in $O(\log n)$ worst case time and uses $O(n)$ space. Moreover, the structure supports all maxima reporting queries in $O(m + \mathit{chng} \cdot \log n)$ worst case time, where *chng* is the total number of changes that the update operations have caused to the staircase that was reported by the latest query operation. Kapoor modifies the structure such that a sequence of queries, and n insertions and d deletions of points requires $O(n \log n + d \log n + r)$ worst case time, where r is the total number of reported maximal points. The worst case update time remains $O(\log n)$, however the query needs $O(r)$ amortized time. Table 1.4 summarizes the dynamic data structures for internal memory that maintain the maximal points under insertions and deletions of points.

3-sided range maxima reporting queries report the maximal points among the points that lie within a given 3-sided rectangle $[x_\ell, x_r] \times [y_b, +\infty[$ unbounded from above. They are obtained by composing maxima contour reporting queries with maxima dominance reporting queries. The structure of [Jan91] support 3-sided range maxima reporting queries in $O(\log n + t)$ worst case time.

4-sided range maxima reporting queries report the maximal points among the points that lie within a given 4-sided rectangle $[x_\ell, x_r] \times [y_b, y_t]$. A special case of 4-sided range maxima queries, namely when the query rectangle is $]-\infty, x_r] \times]-\infty, y_t]$, can be used to answer *rectangular visibility queries*, namely to report the points that are rectangularly visible from a given query point. A point $p \in S$ is *rectangularly visible* from a point q if the orthogonal rectangle with p and q as diagonally opposite corners contains no other point in P. Overmars and Wood [OW88] present a dynamic data structure that supports rectangular visibility queries. It is weight-balanced search tree augmented with secondary structures that support 3-sided range maxima queries. It supports the queries in $O(\log^2 n+t)$ worst case time, where t is the size of the output, insertions and deletions of points in $O(\log^2 n)$ worst case time and uses $O(n \log n)$ space, when the structure of [Jan91] is applied. Both the insertion and deletion time can be improved to $O(\log^2 n)$ worst case, using $O(n \log n)$ space, at the expense of $O(t \log n)$ worst case query time [OW88, Theorem 3.5]. Table 1.5 summarizes the dynamic data structure for rectangular visibility queries in the pointer machine model.

Our Contributions In Section 3.2 we present a dynamic data structure for the pointer machine model, that supports 3-sided range maxima reporting queries in $O(\log n + t)$ worst case time, all maxima reporting queries in $O(m)$ worst case time and maxima emptiness queries in $O(\log n)$ worst case time, using $O(n)$ space. It supports insertions and deletions in optimal $O(\log n)$ worst case time. This improves by a logarithmic factor the worst case deletion time of the structure of Janardan [Jan91] for 3-sided range maxima reporting queries. It attains the same update time as the structure of Kapoor [Kap00]. However, more general 3-sided range maxima reporting queries are supported, and the running time are worst case, rather than amortized.

In Section 3.3 we present a dynamic data structure for the word-RAM model that supports 3-sided range maxima reporting queries in optimal $O(\frac{\log n}{\log \log n} + t)$ worst case, all maxima reporting queries in $O(m)$ worst case time and maxima emptiness queries in optimal $O(\frac{\log n}{\log \log n})$ worst case time, using $O(n)$ space. It supports the insertion and deletion of a point in $O(\frac{\log n}{\log \log n})$ worst case time. These are the first sublogarithmic worst case bounds for all operations in the word-RAM model.

In Section 3.4 we present dynamic data structure for the pointer machine model, that supports 4-sided range maxima reporting queries in $O(\log^2 n + t)$ worst case time, updates in $O(\log^2 n)$ worst case time, using $O(n \log n)$ space. This improves by a logarithmic factor the deletion time of the structure of Overmars and Wood [OW88] for *rectangular visibility queries*, when the dynamic data structure of Janardan [Jan91] for 3-sided range maxima reporting queries is applied. This also improves by a logarithmic factor the query time in [OW88, Theorem 3.5].

1.1.3 Persistent B-Trees

In Chapter 4 we consider the problem of making B-Trees fully persistent in the I/O model. B-Trees [BM72, Com79, HM82] are the most common dynamic dictionary data structures used for external memory. They support *one-dimensional range reporting queries*, namely to report the stored one-dimensional points that belong to a given range $[x_\ell, x_r]$ in $O(\log_B n + t/B)$ worst case I/Os, and updates in $O(\log_B n)$ worst case I/Os, where t is the number of points in the range. Persistent B-trees are the basic building block for multi-versioned data bases [ST99]. In Section 1.2.5 we provide the definitions for persistent data structures. By n we denote the number of elements in the accessed version, by m the total number of updates performed to all versions, by t the size of the range query's output, and by B the size of the block in the I/O model.

The most efficient **fully persistent** B-Trees, which can also be used for confluent persistence, are the fully persistent B^+-Trees of Lanka and Mays [LM91]. They support range queries in $O(\log_B m \cdot (\log_B n + t/B))$ I/Os and updates in $O(\log_B^2 n)$ amortized I/Os, using $O(m/B)$ disk blocks of space. Multiple variants of B-Trees have been made **partially persistent** [BM72, Com79, MS81, HM81, HM82]. Salzberg and Tsotras' [ST99] survey on persistent access methods and other techniques for time-evolving data provides a comparison among partially persistent B$^+$-Trees used to process databases on disks. They include the MVBT structure developed by Becker et al. [BGO$^+$96], the MVAS of Varman and Verma [VV97] and the TSB-Trees of Lomet and Salzberg [LS93]. Moreover, the authors in [GTVV93] acquire partially persistent *hysterical* B-Trees [MS81] optimized for offline batched problems. The most efficient implementation of partially persistent B-Trees was presented by Arge et al. [ADT03] in order to solve efficiently the static point location problem in the I/O model. They support range queries in $O(\log_B m + t/B)$ I/Os and updates in $O(\log_B m)$ amortized I/O, using $O(m/B)$ disk blocks. Table 1.6 summarizes the persistent B-Trees that can be used in an online setting.

Table 1.6: Asymptotic bounds for persistent B-Trees used in an online setting. The number of operations is m, the size of the accessed version is n, the size of the block is B and the size of the range query's output is t.

	Space	Range Query I/Os	Update I/Os
Partially Persistent B-Trees			
[LS93]	m/B	$\log_B m + t/B$	$\log_B^2 m$
[BGO$^+$96]	m/B	$\log_B m + t/B$	$\log_B^2 n$
[VV97]	m/B	$\log_B m + t/B$	$\log_B^2 n$ [a]
[ADT03]	m/B	$\log_B m + t/B \log_B m$ [a]	
Fully Persistent B-Trees			
[LM91]	m/B	$\log_B m \cdot (\log_B n + t/B)$	$\log_B m \cdot \log_B n$ [a]
Chapter 4	m/B	$\log_B n + t/B$	$\log_B n + \log_2 B$ [a]

[a]Amortized I/Os.

Our Contributions In Chapter 4 we present an implementation of fully persistent B-Trees that supports range queries at any version in $O(\log_B n+t/B)$ I/Os and updates at any version in $O(\log_B n + \log_2 B)$ amortized I/Os, using $O(m/B)$ disk blocks.

We obtain the result by first designing a method for the I/O model that makes pointer-based ephemeral data structures fully persistent (Section 4.2). We assume that a record of the ephemeral structure occupies a constant number of blocks. The method achieves $O(1)$ worst case I/O-overhead per access step. The access step for the I/O model is defined as an access to a block of the ephemeral structure. Moreover, the method achieves $O(d_{in} \log B)$ amortized I/O-overhead and $O(\frac{d_{in}}{B})$ amortized space-overhead per update step, where d_{in} is the maximum indegree of an ephemeral record. The update step in the I/O model is defined a constant number of modifications made to a record of the ephemeral structure.

In Section 4.3 we present the *Incremental B-Trees*, an implementation of B-Trees that supports one-dimensional range reporting queries in $O(\log_B n+t/B)$ I/Os and updates in $O(\log_B n)$ I/Os, using $O(n/B)$ blocks of space, and where each update operation performs in the worst case $O(1)$ modifications to the tree. The desired fully persistent B-Trees are achieved by applying our method for I/O-efficient full persistence to Incremental B-Trees (Section 4.4).

1.2 Preliminaries

In this section we provide the basic preliminary structures and methods that we use or modify appropriately in order to attain our results. This moreover serves the purpose of defining the terminology used to present our contributions in the introduction of each chapter. For every presented result that is used by our structures, we refer to the part of the dissertation that concerns it.

1.2.1 Models of Computation

Here we define the considered models of computation and their measures of complexity. Computation in internal memory is usually studied by the pointer machine and the word-RAM models. Accessing data in external memory is usually studied by the I/O model.

The *Pointer Machine model* assumes that the memory consists of a set of records that contain a constant number of fields, where either a value or a pointer to another record can be stored. Comparisons are the only operation allowed on the values, and they take constant worst case time. When the data structure stores n elements, its space $S(n)$ is measured in terms of the number of occupied records. The query time $Q(n)$ and update time $U(n)$ is measured in terms of value comparisons and pointer traversals.

The *Random Access Machine model with word size* w (*word-RAM model*) assumes that the memory consists of an infinite array of cells, where every cell contains w bits. The elements are considered to be integers from the universe $U = \{0, \dots, 2^w - 1\}$. Every cell can be accessed in constant worst case time (random acces), and bit-wise operations allowed on the elements, such as comparisons, shifts, logical operations, etc., that need constant worst case time. When $w \geq \log n$, n different integers can be stored. The integers belong to *rank-space*, where two sets of integers that may have different values, but have the same permutation, are mapped to the same instance of the problem. *Tabulation* can be used in order to support the update operations in constant worst case time. In particular, *precomputed lookup-tables* store precomputed information about every possible permutation. The space occupied by the tables is not the bottleneck, as long as the number of integers in the permutation is small enough. When the data structure stores n elements, its space $S(n)$ is measured in terms of the number of occupied cells. The query time $Q(n)$ and update time $U(n)$ is measured in terms cell accesses and bit-wise operations.

The *Input/Output model* (*I/O model*) [AV88] assumes that the memory consist of two parts. The *external memory* that is an infinite array of blocks, where every block contains at most B elements, and the *internal memory* that consists of an array of $\frac{M}{B}$ blocks. Complexity is measured in *I/O-operations* (*I/Os*), namely in transfers of blocks between the two memories. Computation occurs only among the elements stored in internal memory, and it is for free. A block in external memory can be accessed in one I/O. When the data structure stores n elements, its space $S(n)$ is measured in terms of the number of occupied blocks in external memory. The query time $Q(n)$ and update time $U(n)$ is measured in terms I/O-operations, respectively called query and update I/Os.

1.2.2 Dynamic Search Trees

The data structures presented in Sections 2.2, 2.3, 2.4, 3.2, 3.3, 3.4 and 4.3 are either based on implementations or are modifications of search trees. *Search trees* are dynamic data structures that solve the Dynamic Predecessor Searching problem, namely they support *predecessor searching queries*. Let S be a set of n elements that belong to a totally ordered universe U. Predecessor searching queries ask to report the predecessor in S of a given element of U. The *predecessor* of an element e in S is the largest element in S that is smaller or equal to e. Similarly, the *successor* of an element e in S is the smallest element in S that is larger or equal to e. If e exists in S, it is itself its predecessor and successor. Therefore, predecessor queries can moreover decide whether a given element belongs in S or not. These queries are called *membership queries*. Search trees moreover support insertions and deletions of elements of U to S.

In order to support the operations efficiently, search trees maintain the elements in S sorted with respect to their total order. In particular, search trees are rooted trees that store the elements of S. Every *internal node* u stores a set $L_u = [p_1, e_1, p_2, \ldots, e_{k-1}, p_k]$ of $k-1$ elements $e_1, \ldots, e_{k-1}$ stored in non-decreasing order and k pointers $p_1, \ldots, p_k$. Every pointer p_i, where $i \in [i, k-1]$, points to the *root node* u_i of a recursively defined search tree on the elements x_i of S, such that $e_i < x_i < e_{i+1}$ holds. Pointer p_i is called a *child pointer* to the *child node* u_i of its *parent node* u. Each element of S is stored exactly once in the tree, either in a *leaf* or in an internal node, thus the space of the search tree is linear to the number of stored elements.

The elements stored in an internal node act as *search keys*. The search for the predecessor of element e in the tree begins at the root. In every internal node u, we search for the predecessor element e_i of e in L_u. If e belongs to L_u, and thus $e = e_i$, the search returns e_i. Otherwise if $e_i \neq e$ and the search recurses to the subtree pointed by p_{i+1}. This defines a *search path* from the root to the node that contains e, if it belongs to S. Otherwise, the search path ends at the leftmost leaf of the subtree pointed by the child pointer that follows in L_u the predecessor of e in S.

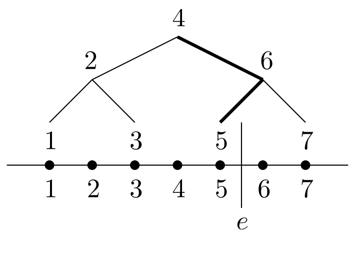

Figure 1.4: Set S is depicted geometrically as one-dimensional points that lie on the infinite line U. The predecessor (resp. successor) of e is 5 (resp. 6). The search tree that stores S is also shown. Thick edges depict the search path.

We define the attributes of search trees that are used in the remainder of the dissertation. A search tree is *node-oriented* when the search keys are stored elements. A search tree is *leaf-oriented* when the search keys are copies of stored elements. A node u is the *ancestor node* of a node v in its subtree. Node v is the *descendant node* of u. Two nodes that have a common parent node are called *sibling nodes*. The *depth* of an internal node u is the number of edges (*length*) in the root-to-u path. The *height* of an internal node u is the maximum length of a u-to-ℓ where ℓ is a leaf in the subtree of u. The *weight* of an internal node is the number of leaves in its subtree. The *degree* of an internal node is the number of its children. A search tree is *binary* when the degree of every internal node is two. The two children are distinguished as *left* and *right*.

Balancing Here we describe how to update search trees efficiently. To insert an element e to S, we search for the node u that contains its predecessor e'. If u is a leaf we insert e in L_u. Otherwise, we insert e in the leaf that contains the successor of e'. Notice that if $e' = e_i$ in L_u, its successor belongs to the leftmost leaf of the subtree pointed by the child pointer p_{i+1}. To delete an element e from S, we search for the node u that contains it. If e does not belong to S, no deletion is required. If u is a leaf, we remove e from L_u. Otherwise, we swap e with its successor element in S, such that e belongs to a leaf, and remove it from the leaf. If the tree is not processed any further after an insertion or deletion of an element, particular sequences of update operations may transform the tree in such a way that the succeeding operations may not have the same asymptotic complexity. For example consider inserting n times an element that is larger than the currently largest element stored in the tree. Searching for the largest element now needs $O(n)$ time. To avoid this situation, the tree has to be "balanced". In particular, a search tree with n leaves and bounded degree is *balanced* if the longest root-to-leaf path is at most $c \log n$, for a constant $c \geq 0$. In order to achieve balance, constraints are imposed to the nodes with respect to some attribute of the tree, such as the height, the weight, the degree of the nodes, etc. As we explained, an insertion or deletion may cause some nodes to violate their constraints and thus render the search tree *unbalanced*. The violating nodes belong to the nodes of the search path for the updated element. The tree is *rebalanced* by restoring the violated constraints.

A search tree is *height-balanced* if there is a constraint on the height of the subtrees rooted at the children of every internal node of the tree. *AVL trees* [AVL62] are height-balanced binary search trees that support predecessor queries, insertions and deletions of elements in $O(\log n)$ worst case, using $O(n)$ space. The balance constraint asserts that the height of the left and the right subtree of every internal node differ by at most one. The tree is rebalanced by performing in the worst case $O(\log n)$ *rotations* over adjacent edges to the unbalanced nodes of the search path. A right rotation over the edge (u, u') where node u is an unbalanced node on the search path and u' is the left child of u, sets the right child of u' as the right child of u, and sets u as the right child of u'. The height the left subtree of u decreases and the height of the right subtree of u' increases by one. Left rotations are symmetric.

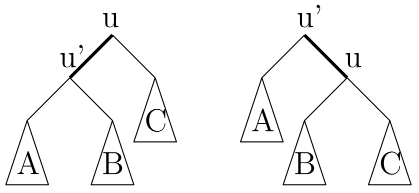

Figure 1.5: From left to right: A right rotation over edge (u, u'). From right to left: A left rotation over edge (u', u).

A search tree is *weight-balanced* if there is a constraint on the weight of every node. The implementation of the data structure presented in Section 3.4 is based on *$BB[\alpha]$-trees* [NR72,BM80]. They are weight-balanced binary search trees that support predecessor queries, insertions and deletions of elements in $O(\log n)$ worst case time, using $O(n)$ space. The balance constraint asserts that the weight of every node differs from the weight of its left child by at most a constant multiplicative factor. The tree is rebalanced using in the worst case $O(\log n)$ rotations along the search path.

A search tree is *path-balanced* if there is a constraint on the length of the longest and the shortest path from every internal node to any leaf of the node's subtree. The implementation of the data structure presented in Section 3.2 is based on *red-black trees* [GS78]. They are path-balanced binary search trees that support predecessor queries, insertions and deletions of elements in $O(\log n)$ worst case time, using $O(n)$ space. The nodes of the tree are assigned *colors* on which constraints are imposed in order to ensure that no path from an internal node u to a leaf ℓ of its subtree is longer than twice the length of any other u-to-ℓ path. In particular, every node can be either red or black. The root, the leaves and the children of every red node are black. Moreover, every path from a node to any leaf of its subtree contains the same number of black nodes. The tree is rebalanced by rotations and by *recolorings*, namely by changing the color assigned to the unbalanced nodes. In the worst case, the tree is rebalanced by $O(1)$ rotations and $O(\log n)$ recolorings after the insertion of a leaf, and by $O(\log n)$ rotations and $O(\log n)$ recolorings after the deletion of a leaf. *BB-trees* [Tar83] is an implementation of red-black trees that are rebalanced in the worst case by $O(\log n)$ recolorings and only $O(1)$ rotations after every insertion and deletion of an element.

A search tree is *degree-balanced* if all the leaves of the tree have the same depth and there is a constraint on the degree of every internal node of the tree. The data structures presented in Sections 2.4 and 4.3 are modifications of *(a, b)-trees* [HM82]. They are degree-balanced search trees that support predecessor queries, insertions and deletions of elements in $O(\log n)$ worst case time, using $O(n)$ space. The amortized update time is $O(1)$ for $2 \leq a$ and $2a \leq b$. The balance constraint asserts that the degree of every internal node, except for the root, is an integer within the range $[a, b]$, and the weight of every leaf is an integer within $[a', b']$, for constants $a' \leq b'$. The degree of the root is within $[2, b]$ when $n \geq 2$. Parameters a and b are constants such that $2 \leq a$ and $2a - 1 \leq b$.

An element is inserted in a leaf of the tree, which may *overflow* the leaf, namely its weight may exceed b'. The leaf is rebalanced by *splitting* it into two balanced leaves ℓ and ℓ'. Leaf ℓ' becomes the new child of the parent node u of ℓ. The leftmost element of ℓ' is moved as the new search key in u, and a new child pointer to ℓ' is inserted to u. This may in turn overflow the internal node u, namely its degree may exceed b. Node u is split recursively, which may cause the splits to cascade up to the root of the tree. The two nodes that occur when splitting the root become the children of a new root. An element is deleted from a leaf of the tree, which may *underflow* the leaf, namely its weight may become smaller than a'. The leaf is rebalanced by *merging* it with one of its sibling leaves ℓ'. The search key in the common parent node u is moved to the resulting leaf, and the child pointer to ℓ' is removed. If the resulting leaf overflows, we perform a *share*, namely we split the leaf again into two balanced leaves. A merge may cause u to undeflow, namely its degree to become smaller than a. Node u is merged recursively, which may cause the merges to cascade up to the root of the tree. When the only two children nodes of the root are merged, the resulting node becomes the new root of the tree.

(a, b)-trees are suitable for the I/O model. By setting $a, b = \Theta(B)$ every node is stored in a constant number of blocks, and thus searching in a node, splitting a node and merging two nodes can be performed in constant worst case I/Os. Predecessor searching, inserting and deleting an element is supported in $O(\log_B n)$ worst case I/Os, using $O(n/B)$ disk blocks.

The Amortized Complexity of Search Trees We study the amortized complexity of rebalancing a search tree, namely the number of rebalancing operations over a sequence of worst case update operations. Two consecutive update operations on $(2, 3)$-trees [AHU74], and more general on $(a, 2a - 1)$-trees or *B-trees*[1] [BM72, Com79], may cause two sequences of $O(\log n)$ rebalancings. For example, consider inserting and deleting repeatedly an element whose insertion rebalances all nodes on the search path. In particular, there exists a sequence of n worst case operations such that the total time for rebalancing the tree is $O(n \log n)$. Namely the amortized rebalancing time is $O(\log n)$. On the other hand, *hysterical* [MS81], *robust* [HM81] and *weak* [HM82] B-trees, namely (a, b)-trees with $b \geq 2a$, have constant amortized rebalancing time. Namely, for every sequence of n worst case operations, the total time for rebalancings is $O(n)$.

For every red-black tree [GS78] there exist an equivalent $(2, 4)$-tree [Bay72]. In particular, a node with degree 4 in a $(2, 4)$-tree corresponds to a black node with two red children in a red-black tree. A node with degree 3 in a $(2, 4)$-tree corresponds to a black node with one red child in a red-black tree. Since the red child can either be the left or the right child, the correspondence between $(2, 4)$-tree and red-black trees is not one to one. In effect, a red node and its parent node in a red-black tree is interpreted as belonging to the same node in the equivalent $(2, 4)$-tree. A rotation in a red-black tree corresponds to switchingthe interpretation of the corresponding degree 3 node. A recoloring in a red-black tree corresponds to splitting a degree 4 node. Therefore, red-black

[1]Notice that under this context, B-trees are translates as "B"ayer-trees. (a, b)-trees with $a, b = O(B)$ are also called B-trees, or else "B"lock-trees.

trees have also constant amortized rebalancing time. In particular, when they are implemented as BB-trees, the time for rotations is $O(1)$ in the worst case, and the time for the recolorings is $O(1)$ amortized.

$BB[\alpha]$-trees satisfy the *weight property* [BM80]. It states that after rebalancing a node with weight w, at least $\Omega(w)$ update operations need to be performed to its subtree, before the node becomes unbalanced again.

Augmented Search Trees Balanced search trees can be used to solve dynamic problems that are harder than Dynamic Predecessor Searching, by accommodating appropriate data structures in their internal nodes. Such *augmented search trees* are used that consist of the *base search tree*, whose internal nodes contain *secondary structures* that store auxiliary information. The data structures presented in Sections 2.4, 3.2, and 3.3 are augmented search trees that solve the Dynamic Planar 3-sided Range Reporting problem and the Dynamic Planar Range Maxima Reporting problem, respectively.

1.2.3 Updating Static Data Structures

Static or semi-dynamic data structures can be transformed to fully dynamic data structures. In principle, parts of the static structure are reconstructed from scratch accommodating the modifications that are being accummulated over a sequence of operations.

Amortized Update Time The *logarithmic method* [Ove87] is a generic technique to transform static data structures into fully dynamic data structures, incurring a logarithmic overhead in all complexities. More precisely, let the static data structure that stores n elements support a particular query in $Q(n)$ worst case time, be preprocessed in $P(n)$ worst case time, and use $S(n)$ space. The resulting dynamic data structure supports the queries in $Q(n) \log n$ worst case time and the updates in $P(n) \log n$ amortized time, using $S(n) \log n$ space.

An update operation is *weak* if it does not degrade the running time of the succeeding query operations by more that a constant factor. More precisely, if a sequence of operations that contains αn weak updates is executed on a dynamic data structure that supports the query in $Q(n)$ worst case time, then the query operations in the sequence need at most $\beta Q(n)$ time in the worst case, where $0<\alpha<1$ and $\beta\leq 1$ are constants. For example, a sequence of $n/2$ deletions are weak updates for a search tree tree since the height of the tree becomes at least $\lfloor \log \frac{n}{2} \rfloor$, and thus queries and updates will still take $O(\log n)$ time. On the other hand, insertions are not weak updates for a search tree, since the n insertions of an elements that is larger than largest element currently stored in the tree causes the height of the tree, and thus the worst case query time, to be $O(n)$.

The technique of *global rebuilding* [Ove87] allows static or semi-dynamic data structures to moreover support weak updates in amortized time. The new version of the data structure is reconstructed from scratch, every time a sequence of update operations (*epoch*) is over. The updates of the previous epoch are incorporated in the construction of the new structure. By selecting an adequately long epoch with respect to the preprocessing time of the structure, the amortized time for the weak updates can be decreased down to a constant.

Augmented Search Trees with Efficient Amortized Update Time The static or semi-dynamic secondary structures of an augmented search tree can be updated by global rebuilding. The data structures presented in Section 2.4 are search trees augmented with static secondary structures. We update them using *partial rebuilding* [Ove87, MT93] that rebalances the nodes in the subtree of the highest unbalanced node, and updates their secondary structures.

Worst Case Update Time The update time becomes worst case when the update operations that exhibit expensive worst case time, are performed *incrementally* over the sequence of succeeding updates. The total time to execute the expensive update is divided into *incremental steps*, such that one step is executed for every succeeding operation. In Section 4.3 we present an implementation of (a, b)-trees where rebalancing operations are performed incrementally and take $O(1)$ worst case time.

The data structures presented in Section 2.3 are updated using *incremental global rebuilding* [Ove87]. This technique constructs incrementally a new data structure, over the sequence of succeeding updates. In particular, two copies of the structure are maintained during an epoch. The working copy, which is initiated every time the reconstruction begins, and the background copy, which is under construction. The queries are performed to the working copy. For every update operation, a *reconstruction step* is executed on the background copy, and thus the reconstruction advances using at most $O(1)$ time. The update is stored in a buffer and performed in the working copy. As soon as the reconstruction is over, the buffered updates are performed incrementally to the background copy. When this is over, the background copy becomes the working copy and a new background copy is constructed, initiating a new epoch.

1.2.4 Dynamic Interpolation Searching

Chapter 2 provides implementations for dynamic data structures that solve efficiently the Dynamic Planar 3-Sided Range Reporing Problem in the *average case*. Worst case sequences of operations do not arise usually in real-world applications. Data structures can be adapted accordingly in order to support efficiently an average case sequence of operations.

Random Input To model the data that arises from real-world applications, we assume that the input is random. In particular, we assume that the input elements are drawn *continuously* from a μ-random *probability distribution*, namely a probability distribution with density function μ. By "drawn continuously" we mean that the distribution does not change during the lifespan of the structure.

The running time of search trees can be significantly improved when the input is "smoothly" distributed, namely when not too many insertions and deletions occur within a constrained interval of the universe U of the elements. Informally, a probability distribution defined over an interval I is *smooth* if the probability density over any subinterval of I does not exceed a specific bound, however small this subinterval is. In other words the distribution does not contain "sharp peaks". In particular, given two functions f_1 and f_2, a density function $\mu = \mu[a,b](x)$ is (f_1, f_2)-*smooth* [MT93] if there exists a constant β, such that for all c_1, c_2, c_3, $a \leq c_1 < c_2 < c_3 \leq b$ and all integers n, it holds that:

$$\int_{c_2 - \frac{c_3 - c_1}{f_1(n)}}^{c_2} \mu[c_1, c_3](x)dx \leq \frac{\beta \cdot f_2(n)}{n}$$

where $\mu[c_1, c_3](x) = 0$ for $x < c_1$ or $x > c_3$, and $\mu[c_1, c_3](x) = \mu(x)/p$ for $c_1 \leq x \leq c_3$ where $p = \int_{c_1}^{c_3} \mu(x)dx$. Intuitively, function f_1 partitions an arbitrary subinterval $[c_1, c_3] \subseteq [a, b]$ into f_1 equal parts, each of length $\frac{c_3 - c_1}{f_1} = O(\frac{1}{f_1})$; that is, f_1 measures how fine is the partitioning of an arbitrary subinterval. Function f_2 guarantees that no part, of the f_1 possible, gets more probability mass than $\frac{\beta \cdot f_2}{n}$; that is, f_2 measures the sparseness of any subinterval $[c_2 - \frac{c_3 - c_1}{f_1}, c_2] \subseteq [c_1, c_3]$. The class of (f_1, f_2)-smooth distributions (for appropriate choices of f_1 and f_2) is a superset of both regular and uniform classes of distributions, as well as of several non-uniform classes [AM93, KMS$^+$03]. Actually, *any* probability distribution is $(f_1, \Theta(n))$-smooth, for a suitable choice of β.

We define the *restricted class* of distributions as the class that contains distributions that exhibit *unbounded density.* The motivation for assuming these distributions derives from the fact that they occur often in practice. Among others the class contains the following districutions. The *Zipfian* distribution is a distribution where the probabilities of occurrance follow Zipf's law. Let N be the number of elements, k be their rank and s be the value of the exponent characterizing the distribution. Then Zipf's law is defined as the function $f(k; s, N) = \frac{1/k^s}{\Sigma_{k=1}^{N} 1/n^s}$. Intuitively, few elements occur very often, while many elements occur rarely. The *Power Law* distributions is a distribution over probabilities that satisfy $Pr[X \geq x] = cx^{-b}$ for constants $c, b > 0$.

Dynamic Interpolation Search Trees Searching among elements that are drawn from a smooth distribution can be benefited by *interpolation* [Pet57], namely by utilizing an estimation of the inverse distribution.

The implementations of the data structure presented in Sections 2.3 and 2.4 contain a variant of *interpolation search trees* [MT93]. An interpolation search tree occupies $O(n)$ space and supports searching for an element in $O(\log\log n)$ expected time, when it stores n elements drawn from a $(n^\alpha, \sqrt{n})$-smooth distribution, for a constant $1/2 \leq \alpha < 1$. The worst case query and update time is $O(\log^2 n)$. Updates need $O(\log n)$ amortized time, and $O(\log\log n)$ expected amortized time when the sequence of update operations is average case.

Augmented sampled forests [AM93] occupy $O(n)$ space and support searching for an element in $\Theta(\log\log n)$ expected time, when n elements are stored that are drawn from a broader class of $(n/(\log\log n)^{1+\epsilon}, n^\delta)$-smooth distributions, for constants $0 < \epsilon, 0 < \delta < 1$. The worst case query time is $O(\log n)$. Updates are supported in $O(\log n)$ worst case time, and $O(\log\log n)$ expected amortized time. The expected search time is $o(\log\log n)$ for $(\frac{n}{o(poly\log^* n)}, \log^{O(1)})$-smooth distributions, and $O(\log n)$ for any distribution.

For the internal memory data structures presented in Sections 2.3 and 2.4, we use the dynamic interpolation search trees of [KMS$^+$06]. This implementation occupies $O(n)$ space and supports searching in $O(\log\log n)$ expected time with high probability, when n elements are stored that are drawn from an even broader class of (n^{α}, n^{δ})-smooth distribution, for constants $0 < \alpha, \delta < 1$, using $O(n)$ space. The worst case query time is $O(\sqrt{\frac{\log n}{\log\log n}})$. The structure moreover allows for multiple occurances of the same elements, and for discrete μ-random distributions. Given the position of the inserted or deleted element, the structure can be updated in $O(1)$ worst case time. The structure uses $O(n)$ space. The expected search time is $O(1)$ with high probability for $(O(n), O(\ln^{O(1)} n))$-smooth distributions, and $O(\log\log n)$ for the resticted class of distributions.

For the external memory data structures presented in Sections 2.3 and 2.4, we use the variant of interpolation search trees for external memory, namely *interpolation search B-trees* [KMM$^+$05]. This implementation occupies $O(n/B)$ blocks of space and supports searching in $O(\log_B \log n)$ expected I/Os with high probablity, when the n elements are drawn from $(n/(\log\log n)^{1+\epsilon}, n^{1/B})$-smooth distributions, for a constant $0 < \epsilon$, using $O(n)$ space. The worst case query I/Os are $O(\log_B n)$. Given the position of the inserted or deleted element, the structure can be updated in $O(1)$ worst case I/Os.

1.2.5 Persistent Data Structures

A *persistent* data structure is a dynamic data structure that remembers all the its versions as updates are performed to it. Depending on the operations we are allowed to do on previous versions, we get several notions of persistence. If we can only update the version produced last and other versions are read-only, the data structure is *partially* persistent. In this case the versions form a list (*version list*). A more general case, the *full* persistence, allows any version to be updated, yielding a *version tree* instead. In turn, this is a special case of *confluent* persistence, where the additional operation of merging different versions together is allowed. Here, the versions form a directed acyclic graph (*version DAG*). A survey on persistence can be found in [Kap04]. Below we survey the generic methods that make dynamic data structures persistent.

Partial Persistence We augment the search trees presented in Sections 3.2 and 3.3, using partially persistent secondary structures. Many data structures, such as search trees and lists, have been made partially persistent. Overmars [Ove87] present generic approaches to make any ephemeral data structure partially persistent in internal memory. The first approach stores explicitly a copy of the structure after every update operation. It has the drawback that the space and update time is at least linear to the size of the updated structure. The second approach stores the sequence of updates, and reconstructs the data structure from scratch in order to query a particular version. A hybrid approach stores only a sequence of k updates, for and appropriate choice of k.

Driscoll et al. [DSST89] present two generic methods that make pointer-based ephemeral data structures partially persistent. The ephemeral structure consists of records that contain a constant number of fields, where either a value or a pointer to another record is stored. Both methods implement a persistent mechanism that records only the modifications that an update operation makes to the ephemeral data structure.

The update operation of the ephemeral structure is decomposed into *update steps* that make a constant number of modifications to one record of the ephemeral structure. The query operation is decomposed into *access steps* that access one record of the ephemeral structure each. The time and space efficiency of the persistent mechanism is measured in terms of the time *overhead* of every access step, and the time and space overhead of every update step performed to the ephemeral structure. In other words, by the aggravation incurred to the running time and space of the ephemeral structure.

The *fat-node method* can be applied to any pointer-based data structure. Over a sequence of m update operations, it achieves $O(\log m)$ worst case time overhead per update and access step and $O(1)$ worst case space overhead per update step. This method stores all the modifications to a record of the ephemeral structure in a corresponding persistent record or *persistent node*, called the *fat node*. The version list induces a total order on the versions. Therefore the contents of a fat node for a particular version can be retrieved by binary search over the versions stored in the fat node.

The *node-copying method* can be applied to pointer-based data structures where at most a constant number of pointers point to the same record, i.e. a structure with constant bounded in-degree. Over a sequence of m update operations, it achieves $O(1)$ amortized time and space overhead per update step and $O(1)$ worst case time overhead per access step. This method follows the fat-node method and creates a new copy of the fat-node as soon as its size exceeds a constant. Since there is only one updatable copy, every ephemeral node corresponds to a linked list of constant size copies of persistent nodes.

Brodal [Bro96] shows how to achieve $O(1)$ worst case update overhead. In particular, the method makes a pointer-based ephemeral data structure with constant bounded in-degree partially persistent, achieving $O(1)$ worst case time and space overhead per update step and $O(1)$ worst case time overhead per access step.

Full Persistence In Section 4.2 we present a method for the I/O model that makes pointer-based ephemeral structures fully persistent. Driscoll et al. [DSST89] modify their methods in order to make ephemeral pointer-based data structures fully persistent in internal memory. The modification of the fat-node method achieves $O(\log m)$ worst case time overhead per update and access step and $O(1)$ worst case space overhead per update step. The version tree induces only a partial order on the versions. The preorder of the version tree induces a total order on the versions, and thus renders any two versions comparable. The preorder of the tree is stored in a dynamic data structure for the *order maintenance* problem [Die82, Tsa84, DS87, BCD+02]. This problem asks to maintain a list under insertions and deletions of elements, such that order maintenance queries can be answered efficiently, namely given two elements in the list to decide which lies to the left of the other. Dietz and Sleator [DS87] present a data structure that supports all operations in $O(1)$ worst case time, and [BCD+02] present a simplified implementation that achieves the same bounds.Finally, the algorithm for updating the persistent mechanism is modified such that the query algorithm can retrieve the contents of a fat node for a particular version, by comparing versions with respect to their preorder.

The *node-splitting method* [DSST89], a modification of the node-copying method, can be applied to pointer-based data structures with constant-bounded indegree. It achieves $O(1)$ amortized time and space overhead per update step and $O(1)$ worst case time overhead per access step. Every ephemeral record corresponds to a linked list of constant size copies of persistent nodes. Since every copy is updatable, the persistent nodes have to be split in order to preserve the structure of the linked list.

Chapter 2

Dynamic Planar 3-Sided Range Queries

Abstract

We consider the problem of maintaining dynamically in internal and external memory a set of planar points, and supporting planar orthogonal range reporting queries of the type $[x_\ell, x_r]\times]-\infty, y_t]$ efficiently, when the x- and y-coordinates of the points are continuously drawn from a probabilistic distribution. By n we denote number of currently stored points, by t the size of the query's output, and by B the block size.

For the word-RAM model, we present a dynamic data structure that supports 3-sided range queries in $O(\log n + t)$ worst case time and updates in $O(\log\log n)$ expected time with high probability, using $O(n)$ space, when the x- and y-coordinates of the points are both drawn from an unknown continuous μ-random distribution. We improve upon the query time by a dynamic data structure that supports 3-sided range queries in $O(\log\log n + t)$ expected time with high probability and updates in $O(\log\log n)$ expected time with high probability, using $O(n)$ space, when the x-coordinates of the points are drawn from a smooth distribution, and the y-coordinates of the points are drawn from a restricted class of distributions that exhibit high density, such as the Zipfian or the Power Law distribution. Finally we waive the assumption on the y-coordinates, and present a dynamic data structure that supports 3-sided range queries in $O(\log\log n + t)$ expected time with high probability and updates in $O(\log\log n)$ expected amortized time, using $O(n)$ space, when the x-coordinates of the points are drawn from a smooth distribution, and the y-coordinates are arbitrarily distributed. For all structures we assume that every stored point is deleted with equal probability.

We present an $O(n/B)$ space I/O-efficient variant of each structure that is efficient under similar assumptions. We present a dynamic data structure that supports 3-sided range queries in $O(\log_B n{+}t/B)$ worst case I/Os and updates in $O(\log_B \log n)$ expected amortized I/Os with high probability, a dynamic data structure that supports 3-sided range queries in $O(\log_B \log n{+}t/B)$ expected I/Os with high probability and updates in $O(\log_B \log n)$ expected amortized I/Os with high probability, and a dynamic data structure that supports 3-sided range queries in $O(\log \log_B n{+}t/B)$ expected I/Os with high probability and updates in $O(\log_B \log n)$ expected amortized I/Os, respectively.

2.1 Introduction

In this Chapter we present dynamic data structures for the internal and external memory that support planar orthogonal 3-sided range reporting queries and updates efficiently in the average case, namely when the x- and y-coordinates of the points are continuously drawn from a probabilistic distribution.

In Section 2.2 we strive for $O(\log\log n)$ update time. We present a dynamic data structure for the word-RAM model and its variant for the I/O model. The structure for the word-RAM model is a modification of the priority search tree [McC85]. A priority search tree is a hybrid of a binary heap and a balanced binary search tree. In particular, it consists of a binary balanced search tree that supports predecessor queries for the the x-coordinates of the points. The root of the tree contains the search key for the x-dimension and the point $p_{y_{\min}}$ with minimum y-coordinate among the points in its subtree. The left and right subtrees of the root are defined recursively as priority search trees over the set of points that they contain, excluding point $p_{y_{\min}}$. The tree satisfies the *min-heap property*, namely that the points $p_{y_{\min}}$ in the nodes of any root-to-leaf path occur in non-increasing order.

An update operation to a priority search tree involves recomputing the point $p_{y_{\min}}$ along a root-leaf-path. However, we can only update nodes up to height $O(\log\log n)$ in $O(\log\log n)$ time . We define a *multi-level* priority search tree, namely we divide it into a *lower level* that contains all the nodes of height at most $\log\log n$, and a *higher level* that contains all the nodes of height larger than $\log\log n$. In Theorem 2.1 we prove that if the x- and y- coordinates of the points are continuously drawn from an unknown continuous μ-random distribution, then during a sequence of $O(\log n)$ update operations, at most a constant number of update operations will involve nodes in the higher level with high probability. The higher level is updated incrementally over an epoch of $O(\log n)$ update operations, and thus updating the lower level comprises the bottleneck of every update operation. The I/O-efficient variant of this structure is an analogous modification to the external priority search tree [ASV99].

In Section 2.3 we strive for both $O(\log\log n)$ query and update time. We present a dynamic data structure for the word-RAM model and its variant for the I/O model. The structure for the word-RAM model is a modification of the structure in Section 2.2. To identify the points that have x-coordinates within the query range in $O(\log\log n)$ time, we store the points at the leaves of the lower level in an interpolation search tree. In fact, we use the implementation of [KMS$^+$06], since it attains $O(\log\log n)$ expected time with high probability allowing the weakest assumption on the distribution of the x-coordinates.

To report the points that have y-coordinates within the query range in $O(\log\log n+t)$ time, we implement the higher level as a modified priority search tree [SMK$^+$04]. Since it is a static data structure, it can be dynamized by incremental global rebuilding in $O(n/\log n)$ worst case time. In Theorem 2.4 we prove that if the x- coordinates are drawn from a smooth distribution and the y- coordinates are drawn from a distribution of the restricted class, then during a sequence of $O(n)$ update operations, $O(\log n)$ update operations will involve nodes in the higher level with high probability. The higher level is updated by incremental global rebuilding over an epoch of $O(n)$ update operations, and thus the bottleneck of the update operation consists of updating the interpolation search tree, updating the lower levels, and managing the $O(\log n)$ update operations that involve the higher level during the epoch. The variant of this data structure for the I/O model is an analogous modification to the external priority search tree [ASV99], where we use modified external priority search trees [KPS$^+$10] and interpolation search B-trees [KMM$^+$05] instead.

In Section 2.4 we strive for $O(\log\log n)$ query and update time, while waiving the assumption on the distribution of the y-coordinates of the points. We make no restricting assumption about the distribution of the y-coordinates of the points. In particular, we assume that y-coordinates are drawn from any arbitrary distribution. We use interpolation search trees to store the x-coordinates, and thus we assume that they are drawn from a smooth distribution.

To manage the y-coordinates we first present *weight-balanced exponential trees*, a combination of weight-balanced B-trees [AV03] with exponential search trees [AT07]. *Weight-balanced B-trees* [AV03] is an adaptation of $BB[\alpha]$-trees that combines weight-balancing with degree-balancing and is also suitable for the I/O model. More precisely, a weight-balanced B-tree with degree parameter k and leaf parameter ℓ, for costants $k > 4$ and $\ell > 0$, if the leaves are in the same level and have weight within $[\ell, 2\ell - 1]$, the internal nodes at level i have weight w_i within $[\frac{1}{2}a^i, 2a^i]$, and the root has minimum degree 2 and maximum weight n. The tree is weight balanced by definition, and also degree balanced, since the degree of a node at level i is $\frac{w_i}{w_{i-1}} \in [\frac{k}{4}, 4k]$. Weight-balanced B-trees also satisfy the weight property [AV03].

Exponential search trees [AT07] is a method for the word-RAM model that transforms static data structures that support a particular query operation into a fully dynamic data structure. In particular, let S be a static data structure that supports the query in $Q(n)$ worst case time, and needs $O(n^{k-1})$ space and preprocessing time, when n elements are stored in it, for constant $k \geq 2$. An exponential search tree supports the same query and updates in $T(n)$ worst case time, where $T(n) \leq T(n^{1-1/k}) + O(Q(n))$, using $O(n)$ space. The weight w_i of a node of height i is $\Theta(c^{(\frac{1}{1-k})^i})$, for a constant c. Every node at height i is augmented with a static secondary structure that contains an element in the subtree of every child of the node, and thus contains $\Theta(\frac{w_i}{w_{i-1}}) = \Theta(w_i^{\frac{1}{k}})$ elements. Thus, the time to rebuild the secondary structure of a node at level i is linear to the time for rebuilding the whole subtree (*dominance property*).

In order to report the points with y-coordinate that that lie within the query range, we define a weight-balanced exponential tree that moreover satisfies the min-heap property. We follow the paradigm of [AT07] and augment every node of the weight-balanced exponential search tree with a static secondary structure that supports 3-sided range queries [ABR00] on the points $p_{y-\min}$ of its children nodes. In fact, the authors in [ABR00] observe that 3-sided range queries can be supported by a static data structure for *range minimum queries*. In particular, given an array with n elements from a total order, a range minimum query asks to report the minimum element among the elements that lie between to given indices of the array. The static data structure of [HT84] supports range minimum queries in $O(1)$ worst case time and uses $O(n)$ space and preprocessing time. In Lemma 2.2 we prove that the weight-balanced exponential search tree also satisfies the weight-property. In Lemma 2.4 we prove that the augmented weight-balanced exponential search tree also satisfies the dominance property, and thus it supports an update operation in $O(1)$ amortized time (Lemma 2.5). In Lemma 2.6 we prove that the tree can be updated in $O(1)$ expected amortized time, regardless of the distribution of the y-coordinates of the points. Thus the total update time becomes expected amortized and is asymptotically dominated by the $O(\log \log n)$ time for the interpolation search.

For the variant of this data structure for the I/O model, we use interpolation search B-trees [KMM$^+$05] to store the points by x-coordinate. We adapt the weight-balanced exponential search trees to the I/O model and augment them with precomputed tables that support 3-sided range queries. To achieve $O(n/B)$ blocks of space, we implement every consecutive $Theta(B^2)$ leaves of the interpolation search B-tree with the dynamic data structure of Arge et al. [ASV99, Lem. 1] that stores at most $k \leq B^2$ points in the plane, and supports 3-sided range queries in $O(t/B + 1)$ worst case I/Os, and updates in $O(1)$ worst case I/Os, using $O(k/B)$ blocks of space.

2.2 3-Sided Queries for Unknown Random Distributions

The internal memory structure consists of two levels, as well as an auxiliary data structure. All of them are implemented as priority search trees. The lower level partitions the points into *buckets* of almost equal logarithmic size according to their x-coordinates. That is, the points are sorted in increasing order according to x-coordinate and then divided into sets of $O(\log n)$ elements each of which constitutes a bucket. A bucket C is implemented as a priority search tree and is represented by a point C^{min} which has the smallest y-coordinate among all points in it. This means that for each bucket the cost for insertion, deletion and search is equal to $O(\log \log n)$, since this is the height of the priority search tree that represents C. The higher level is a priority search tree on the representatives of the lower level. Thus, the number of leaves in the higher level is $O\left(\frac{n}{\log n}\right)$. As a result, the higher level supports the operations of insert, delete and search in $O(\log n)$ time. In addition, we keep an auxiliary priority search tree for insertions of violating points. Under this context, we call a point p *violating*, when its y-coordinate is less than C^{min} of the bucket C in which it should be inserted. In case of a violating point we must change the representative of C and as a result we should make an update operation on the PST of the higher level, which costs too much, namely $O(\log n)$.

Lemma 2.1 *Let μ be the unknown continuous density function of a probability distribution $\mathcal{F}$ over a totally ordered universe of elements. If μ does not change as elements are drawn from $\mathcal{F}$ and stored in the set $\mathcal{S}$, the probability that the next drawn element q is less than the minimum element s in $\mathcal{S}$ is equal to $\frac{1}{|\mathcal{S}|+1}$.*

Proof. Suppose that we have n random observations $X_1,\ldots,X_n$ from an unknown continuous probability density function $f(X)$, with cumulative distribution $\mu = F(X)$, $X \in [a,b]$. We want to compute the probability that the $(n+1)-th$ observation is less than $min\{X_1,\ldots,X_n\}$. Let $X_{(1)} = min\{X_1,\ldots,X_n\}$. Therefore, $P\left\{X_{n+1} < X_{(1)}\right\} = \sum_x P\left\{X_{n+1}{<}X_{(1)}/X_{(1)}{=}x\right\} \cdot P\left\{X_{(1)} = x\right\}$ (α).

We can see that $P\left\{X_{n+1} < X_{(1)}/X_{(1)}{=}x\right\} = F(X){=}P\left\{X_{n+1}{<}x\right\}$ (β). Also $P\left\{X_{(k)} = x\right\} = n \cdot f(x) \cdot \binom{n-1}{k-1} \cdot F(X)^{k-1} \cdot (1-F(X))^{n-k}$ (γ), where $X_{(k)}$ is the $k-th$ smallest value in $\{X_1,\ldots,X_n\}$.

In our case $k = 1$, which intuitively means that we have n choices for one in $\{X_1,\ldots,X_n\}$ being the smallest value. This is true if all the rest $n-1$ are more than x, which occurs with probability: $(1-F(X))^{n-1} = (1-P\{X < x\})^{n-1}$. By (β) and (γ), expression (α) becomes:

$$P\left\{X_{n+1} < X_{(1)}\right\} = \int_b^a n \cdot f(X)\binom{n-1}{k-1} \cdot F(X) \cdot (1-F(X))^{n-1}\,dX =$$
$$\int_b^a n \cdot f(X) \cdot (1-F(X))^{n-1} \cdot F(X)dX = \int_b^a \left[-(1{-}F(X))^n\right]' F(X)\,dX =$$
$$\int_b^a \left[-(1{-}F(X))^n \cdot F(X)\right]' dX + \int_b^a (1{-}F(X))^n \cdot F'(X)dX =$$
$$\left\{-(1{-}F(X))^n \cdot F(x)|_a^b\right\} + \int_b^a -\left[\frac{(1{-}F(X))^{n+1}}{n+1}\right]' dX =$$
$$-(1{-}F(b))^n \cdot F(b) + (1{-}F(a))^n \cdot F(a) - \left\{\frac{(1{-}F(X))^{n+1}}{n+1}|_a^b\right\} =$$
$$-\left\{\frac{(1{-}F(b))^{n+1}}{n+1} - \frac{(1{-}F(a))^{n+1}}{n+1}\right\} = \frac{1}{n+1}.$$
□

Proposition 2.1 *Let the x-coordinates be continuously drawn from an unknown μ-random distribution on the interval $[a,b] \subseteq \Re$. Let there be n elements stored in the data structure. Partition $[a,b]$ into $\lceil \frac{n}{\log n} \rceil$ buckets, such that each bucket contains $\log n$ points. Consider an epoch that contains $\log n$ update operations. Let $N(i)$ denote the number of stored elements into after the i-th update operation of the epoch. Then $N(i) \in [n, r \cdot n]$, for a constant $r > 1$, and the $N(i)$ elements remain μ-randomly distributed in each bucket.*

Proof. The proof is analogous to [KMS+03, Lem. 2] and is omitted. □

Theorem 2.1 *For a sequence of $O(\log n)$ update operationss, the expected number of violating elements is $O(1)$ with high probability, assuming that the elements are being continuously drawn from a μ-random distribution.*

Proof. According to Proposition 2.1, there are $N(i) \in [n, r \cdot n]$ (with constant $r > 1$) elements with their x-coordinates μ-randomly distributed in the buckets $j = 1,\ldots,\frac{n}{\log n}$, that partition $[a,b] \subseteq \Re$. By [KMS+03, Th. 4], with high probability, each bucket j receives an x-coordinate with probability $p_j = \Theta(\frac{\log n}{n})$. It follows that during the i-th update operation, the elements in bucket j is a Binomial random variable with mean $p_j \cdot N(i) = \Theta(\log n)$.

The elements with x-coordinates in an arbitrary bucket j are $\alpha N(i)$ with probability $\binom{N(i)}{\alpha N(i)} p_j^{\alpha N(i)} (1-p_j)^{(1-\alpha)N(i)} \sim \left[\left(\frac{p_j}{\alpha}\right)^\alpha \left(\frac{1-p_j}{1-\alpha}\right)^{1-\alpha}\right]^{N(i)}$. These are $\leq \alpha N(i) = \frac{p_j}{2} N(i)$ (less than half of the bucket's mean) with probability

$$\leq \frac{p_j N(i)}{2} \cdot \left[\left(\frac{p_j}{\alpha}\right)^\alpha \left(\frac{1-p_j}{1-\alpha}\right)^{1-\alpha}\right]^{N(i)} \to 0 \text{ as } n \to \infty \text{ and } \alpha = \frac{p_j}{2}. \qquad (2.1)$$

Suppose that an element is inserted in the i-th update. It induces a violation if its y-coordinate is strictly the minimum element of the bucket j it falls into.

- If the bucket contains $\geq \frac{p_j}{2} \log N(i) \geq \frac{p_j}{2} \log n$ coordinates, then by Lemma 2.1 element y incurs a violation with probability $O(\frac{1}{\log n})$.

- If the bucket contains $< \frac{p_j}{2} \log N(i)$ coordinates, which is as likely as in Eq. (2.1), then element y may induce ≤ 1 violation.

Putting these cases together, element y expectedly induces at most $O(\frac{1}{\log n})$+Eq. (2.1)= $O(\frac{1}{\log n})$ violations. We conclude that during the whole epoch of $\log n$ insertions the expected number of violations are at most $\log n \cdot O(\frac{1}{\log n})$ plus $\log n \cdot$ Eq. (2.1) which is $O(1)$. $\square$

Update Operation We assume that the x and y-coordinates are drawn from an unknown μ-random distribution and that the μ function never changes. Under this assumption, according to the combinatorial game of bins and balls, presented in Section 5 of [KMS$^+$03], the size of every bucket is $O(\log^c n)$, where $c > 0$ is a constant, and no bucket becomes empty with high probability. We consider epochs of size $O(\log n)$, with respect to update operations. During an epoch, according to Theorem 2.1, the number of violating points is expected to be $O(1)$ with high probability. The auxiliary priority search tree stores exactly those $O(1)$ violating points. When a new epoch starts, we take all points from the auxiliary priority search tree and insert them in the respective buckets in time $O(\log \log n)$ expected with high probability. Then we need to incrementally update the auxiliary priority search tree of the higher level. This is done during the new epoch that just started. In this way, we keep the auxiliary priority search tree of the higher level updated and the size of the extra auxiliary priority search tree constant. As a result, the update operations are carried out in $O(\log \log n)$ time expected with high probability, since the update of the higher level costs $O(1)$ time in the worst case.

Query Operation The query algorithm first searches in the higher priority search tree for the buckets L and R that contain the points p_ℓ and p_r with x-coordinates that respectively succeed and precede the query parameters x_ℓ and x_r. It traverses the nodes of the respective search paths P_ℓ and P_r that span both levels of the structure and reports their points $p_{y_{\min}}$ that belong to the query range.

The subtrees rooted at the right children of the nodes of $P_\ell - P_r$ and the left children of $P_r - P_\ell$ contain points whose x-coordinates belong to the query range. The algorithm checks if the y-coordinate of the point $p_{y_{\min}}$ stored in the roots of these subtrees is smaller or equal to y_t. If it is not, the algorithm terminates for the particular subtree, since all the points in the subtree have even larger y-coordinate than $p_{y_{\min}}$. Otherwise, the point is reported and the algorithm recurses in both its children nodes. When $p_{y_{\min}}$ is the representative point of a bucket, the x-coordinates of the points in the bucket belong to the query range. If $p_{y_{\min}}$ also belongs to the query range, the query algorithm continues reporting the points in the priority search tree of the bucket in the same way. Finally, we check the auxiliary priority search tree for points to be reported points. The total worst case query time is $O(\log n + t)$.

Note that deletions of points do not affect the correctness of the query algorithm. If a non violating point is deleted, it resides on the lower level and thus it would be deleted online. Otherwise, the auxiliary priority search tree contains it and thus the deletion is performed online. No deleted violating point is incorporated into the higher level, since by the end of the epoch the auxiliary priority search tree contains only inserted violating points.

Theorem 2.2 *There exists a dynamic data structure for the word-RAM model that supports 3-sided range reporting queries in $O(\log n + t)$ worst case time, supports updates in $O(\log\log n)$ expected time with high probability, using $O(n)$ space, under the assumption that the x- and y-coordinates are continuously drawn from a continuous unknown μ-random distribution, and that the stored points are deleted with equal probability.*

External Memory The I/O-efficient variant of the above structure is obtained by using external priority search trees [ASV99], instead of internal memory priority search trees. However the update time becomes amortized.

Theorem 2.3 *There exists a dynamic data structure for the word-RAM model that supports 3-sided range reporting queries in $O(\log_B n + t/B)$ worst case I/Os, supports updates in $O(\log_B \log n)$ amortized expected I/Os with high probability, using $O(n)$ space, under the assumption that the x- and y-coordinates are continuously drawn from a continuous unknown μ-random distribution, and that the stored points are deleted with equal probability.*

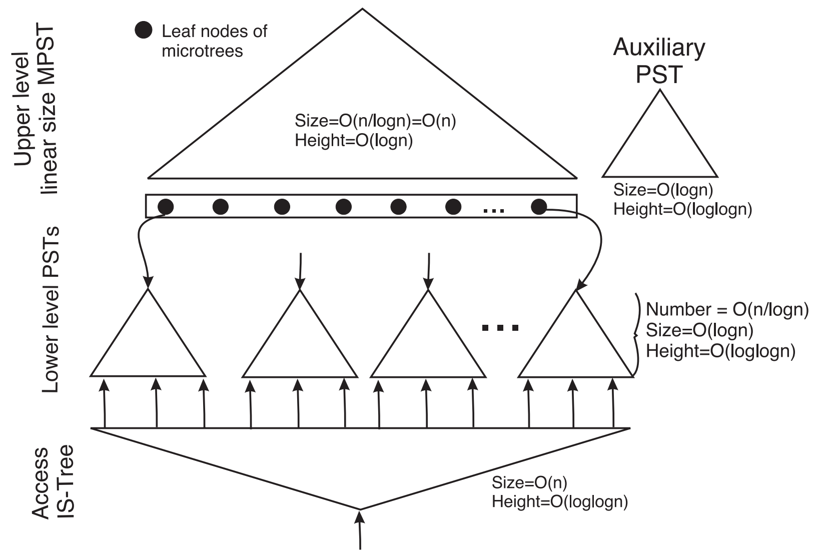

Figure 2.1: Internal memory structure for smooth and restricted distributions.

2.3 3-Sided Queries for Smooth and Restricted Distributions

We modify the internal memory structure of Section 2.2 as following. We store the x-coordinates of the points in a dynamic interpolation search tree [KMS$^+$06]. Every point belongs to a bucket of the multi-level structure. For every point, we store a pointer from the x-coordinate in the interpolation search tree to the corresponding leaf of the priority search tree that implements the bucket. The upper level is implemented as a static modified priority search tree [SMK$^+$04]. Figure 2.1 shows the modified structure.

Theorem 2.4 *For a sequence of $O(n)$ updates, the expected number of violating elements is $O(\log n)$ with high probability, assuming that x- coordinates are drawn from a continuous smooth distribution and the y- coordinates are drawn from the restricted class of distributions (*Power Law *or* Zipfian*).*

Proof. Suppose an element is inserted, with its y-coordinate following a discrete distribution (while its x-coordinate is arbitrarily distributed) in the universe $\{y_1, y_2, \ldots\}$ with $y_i < y_{i+1}, \forall i \geq 1$. Also, let $q = \Pr[y > y_1]$ and y_j^* the min y-coordinate of the elements in bucket j as soon as the current epoch starts. Clearly, the element just inserted incurs a violation when landing into bucket j with probability $\Pr[y < y_j^*]$.

- If the bucket contains $\geq \frac{p_j}{2} \log N(i) \geq \frac{p_j}{2} \log n$ coordinates, then coordinate y incurs a violation with probability $\leq q^{\frac{p_j}{2} \log n}$. (In other words, a violation may happens when at most all the $\Omega(\log n)$ coordinates of the elements in bucket j are $> y_1$, that is, when $y_j^* > y_1$.)
- If the bucket contains $< \frac{p_j}{2} \log N(i)$ coordinates, which is as likely as in Eq. (2.1) then coordinate y may induces ≤ 1 violation.

All in all, y coordinate expectedly induces $\leq q^{\Omega(\log n)}+$ Eq. (2.1) violations. Thus, during the whole epoch of n insertions the expected number of violations are at most $n \cdot \left(q^{\Omega(\log n)}\right) + n\cdot$ Eq. (2.1) $= nq^{\Omega(\log n)} + o(1)$ violations. This is at most $c \cdot \log n = O(\log n)$ if $q \leq \left(\frac{c \log n}{n}\right)^{(\log n)^{-1}} \to e^{-1}$ as $n \to \infty$. Note that *Power Law* and *Zipfian* distributions also satify this property. □

Update Operation We consider an epoch of $O(n)$ update operations. For every insertion of a point, we first insert the x-coordinate of the point in the interpolation search tree, traverse the pointer and update the corresponding bucket with the y-coordinate. If there is a violation, we store the point in the auxiliary priority search tree. When a new epoch starts, we update the modified priority search tree with the points stored in the auxiliary priority search tree using incremental global rebuilding [Ove87].

The time for updating the interpolation search tree is $O(\log\log n)$ expected with high probability. A bucket is updated in $O(\log\log n)$ worst case time. By Theorem 2.4 the expected number of violating points in the epoch is $O(\log n)$ with high probability. Thus the auxiliary priority search tree consumes at most $O(\log n)$ space during an epoch and is updated in $O(\log\log n)$ expected time with high probability. The modified priority search tree is updated in $O(1)$ worst case time per update operation.

Query Operation To answer a 3-sided range query, we query first the interpolation search tree in order to find the points p_ℓ and p_r with x-coordinates that respectively succeed and precede the query parameters x_ℓ and x_r.

Then the query proceeds bottom up the multi-level structure. First it traverses the priority search trees of the buckets that contain p_ℓ and p_r, starting from the leaves that contain the points up to the root node. Then it queries the modified priority search tree to report points in the upper level that need to be reported, and to identify the buckets where the y-coordinate of the representative point and x-coordinates of all the stored points of the bucket lie within the query range. The query traverses these buckets top-down. Finally, the auxiliary priority search tree is queried.

The search in the interpolation search tree takes $O(\log\log n)$ expected time with high probability. The paths in the buckets that contain p_ℓ and p_r have length $O(\log\log n)$. The time to process the modified priority search tree and to report the points that are stored in the buckets, is charged to the output. By Theorem 2.4, the size of the auxiliary priority search tree is $O(\log n)$ with high probability, thus it is queried in $O(\log\log n)$ expected time with high probability. The total query time is $O(\log\log n{+}t)$ expected with high probability.

Theorem 2.5 *There exists a dynamic data structure for the word-RAM model that supports 3-sided range reporting queries in $O(\log\log n + t)$ expected time with high probability, supports updates in $O(\log\log n)$ expected time with high probability, using $O(n)$ space, under the assumption that the x-coordinates of the points are continuously drawn from a (n^α, n^δ)-smooth distribution, for constants $0 < \alpha, \delta < 1$, the y-coordinates are continuously drawn from a distribution of the restricted class such as the Zipfian or the Power Law distribution, and that the stored points are deleted with equal probability.*

External Memory The I/O-efficient variant of the above structure consists of an interpolation search B-tree [KMM+05] for the x-coordinates, a three-level structure for the y-coordinates, and auxiliary external priority search tree [ASV99]. The n points are divided with respect to their x-coordinate into $n' = \frac{n}{\log n}$ buckets of size $\log n$. Each bucket is implemented as an external priority search tree [ASV99]. The external priority search trees constitute the lower level of the structure. The n' representatives of the external priority search trees are again divided into buckets of size $O(B)$. These buckets single blocks and constitute the middle level. The $n'' = \frac{n'}{B}$ representatives are stored in the leaves of an external modified priority search tree [KPS+10], which constitutes the upper level of the structure. In total, the space of the aforementioned structures is $O(n' + n'' + n'' \log^{(k)} n'') = O(\frac{n}{\log n} + \frac{n}{B \log n} + \frac{n}{B \log n} B) = O(\frac{n}{\log n}) = O(\frac{n}{B})$, where k is such that $\log^{(k)} n'' = O(B)$ holds.

The update and query algorithms are similar to those of the internal memory structure. The interpolation search B-tree is updated in $O(\log_B \log n)$ expected I/Os with high probability. The external priority search tree of the bucket that contains the updated point is updated in $O(\log_B \log n)$ amortized expected I/Os with high probability. A middle level block is updated in one I/O. The upper level is updated in $O(1)$ worst case I/Os per update operation. By Theorem 2.4 the auxiliary external priority search tree is updated in $O(\log_B \log n)$ amortized expected I/Os with high probability. Thus in total I/Os for an update are $O(\log_B \log n)$ amortized expected I/Os with high probability.

The query operation takes $O(\log_B \log n)$ expected I/Os with high probability for the interpolation search B-tree, for the traversal of the two buckets that contain p_ℓ, p_r, and for the traversal of the auxiliary external priority search tree. In total the query takes $O(\log_B \log n + t/B)$ I/Os expected with high probability, since the rest of the query I/Os can be charged to the output.

Theorem 2.6 *There exists a dynamic data structure for the I/O model that supports 3-sided range reporting queries in $O(\log_B \log n + t)$ expected I/Os with high probability, supports updates in $O(\log_B \log n)$ amortized expected I/Os with high probability, using $O(n/B)$ blocks of space, under the assumption that the x-coordinates of the points are continuously drawn from a $(n/(\log \log n)^{1+\epsilon}, n^{1/B})$-smooth distribution, for a constant $0 < \epsilon$, the y-coordinates are continuously drawn from a distribution of the restricted class such as the Zipfian or the Power Law distribution, and that the stored points are deleted with equal probability.*

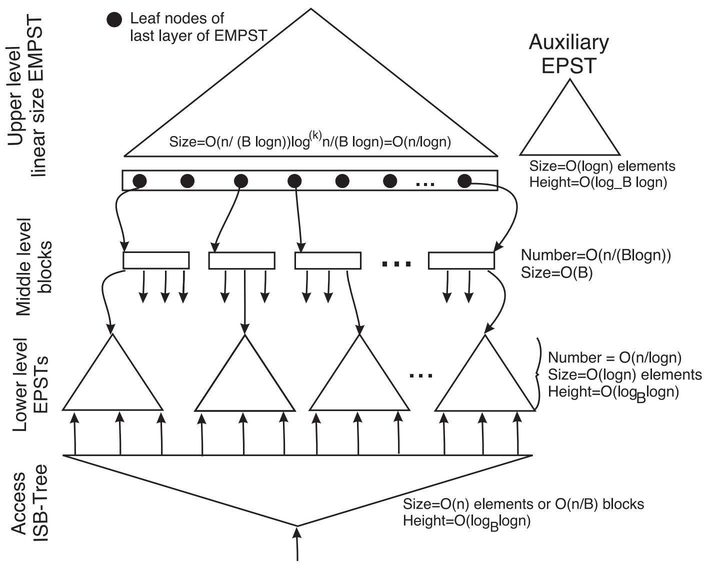

Figure 2.2: External memory structure for smooth and restricted distributions.

2.4 3-Sided Queries for Smooth and Arbitrary Distributions

Here we describe a variant of exponential search trees [AT07] that we dynamize using a rebalancing scheme relative to that of the weight-balanced search trees [AV03]. In particular, a *weight-balanced exponential tree* T on n points is a leaf-oriented rooted search tree where the degrees of the nodes increase double exponentially on a leaf-to-root path. All leaves have the same depth and reside on the lowest level of the tree (level zero). The *weight* of a subtree T_u rooted at node u is defined to be the number of its leaves. If u lies at level $i \geq 1$, the weight of T_u ranges within $\left[\frac{1}{2} \cdot w_i + 1, 2 \cdot w_i - 1\right]$, for a *weight parameter* $w_i = c_1^{c_2^i}$ and constants $c_2 > 1$ and $c_1 \geq 2^{3/(c_2-1)}$ (see Lemma 2.2). Note that $w_{i+1} = w_i^{c_2}$. The root does not need to satisfy the lower bound of this range. Intuitively, a node at level i has degree $d_i = \sqrt{w_{i-1}}$ and has as children nodes that root trees of size w_{i-1}. Moreover, it holds that $d_i = d_{i-1}^{3/2}$. The tree has height $\Theta(\log_{c_2} \log_{c_1} n)$.

The insertion of a new leaf to the tree increases the weight of the nodes on the leaf-to-root path by one. This might cause some weights to exceed their range constraints ("overflow"). We *rebalance* the tree in order to revalidate the constraints by a leaf-to-root traversal, where we "split" each node that overflowed. An overflown node u at level i has weight $2w_i$. A split is performed by creating a new node v that is a sibling of u and redistributing the children of u among u and v such that each node acquires a weight within the allowed range. In particular, we scan the children of u, accumulating their weights until we exceed the value w_i, say at child x. Node u gets the scanned children and v gets the rest. Node x is assigned as a child to the node with the smallest weight. Processing the overflown nodes u bottom up guarantees that, during the split of u, its children satisfy their weight constraints.

The deletion of a leaf might cause the nodes on the leaf-to-root path to "underflow", i.e. a node u at level i reaches weight $\frac{1}{2}w_i$. By an upwards traversal of the path, we discover the underflown nodes. In order to revalidate their node constraints, each underflown node chooses a sibling node v to "merge" with. That is, we assign the children of u to v and delete u. Possibly, v needs to "split" again if its weight after the merge is more than $\frac{3}{2}w_i$ ("share"). In either case, the traversal continues upwards, which guarantees that the children of the underflown nodes satisfy their weight constraints. The following lemma, which is similar to [AV03, Lem. 9], holds.

Lemma 2.2 *After rebalancing a node u at level i, $\Omega(w_i)$ insertions or deletions need to be performed on T_u, for u to overflow or underflow again.*

Proof. A split, a merge or a share on a node u on level i yield nodes with weight in $\left[\frac{3}{4}w_i - w_{i-1}, \frac{3}{2}w_i + w_{i-1}\right]$. If we set $w_{i-1} \leq \frac{1}{8}w_i$, which always holds for $c_1 \geq 2^{3/(c_2-1)}$, this interval is always contained in $[\frac{5}{8}w_i, \frac{14}{8}w_i]$. □

Internal Memory

Our internal memory construction for storing n points in the plane consists of an interpolation search tree [KMS$^+$06] that stores the points in sorted order with respect to the x-coordinates. On the sorted points, we maintain a weight-balanced exponential search tree T with $c_2 = 3/2$ and $c_1 = 2^6$. Thus its height is $\Theta(\log\log n)$. In order to use T as a priority search tree, we augment it as follows. The root stores the point with overall minimum y-coordinate. Points are assigned to nodes in a top-down manner, such that a node u stores the point with minimum y-coordinate p_{y-min} among the points in T_u that is not already stored at an ancestor of u. Note that the point from a leaf of T can only be stored at an ancestor of the leaf and that the y-coordinates of the points p_{y-min} stored at a leaf-to-root path are monotonically decreasing (*Min-Heap Property*). Finally, every node contains a static structure for range minimum queries [HT84], henceforth reffered to as an *RMQ-structure*, that in turn stores the y-coordinates of the points p_{y-min} in the children nodes and an array with pointers to the children nodes.

Corollary 2.1 *The total space of the internal memory data structure is $O(n)$.*

Proof. In total, excluding the leaves, each element occurs twice in T, namely at an internal node u and at the RMQ-structure of u's parent. Thus T consumes $O(n)$ space. Moreover, the space for the interpolation search tree is linear [MT93, KMS$^+$06], which yields total $O(n)$ space. □

Query Operation Before we describe the query algorithm of the data structure, we will describe the query algorithm that finds all points with y-coordinate less than y_t in a subtree T_u, rooted at internal node u. The query begins at node u. At first we check if the y-coordinate of the point p_{y-min} stored at u is smaller or equal to y_t (we call it a *member* of the query). If not we stop. Else, we identify the t_u children of u storing points p_{y-min} with y-coordinate less than or equal to y_t, using the RMQ-structure of u. That is, we first query the whole array and then recurse on the two parts of the array partitioned by the index of the returned point. The recursion ends when the accessed point p_{y-min} has y-coordinate larger than y_t (*non-member* point).

Lemma 2.3 *For an internal node u and value y_t, all points stored in T_u with y-coordinate $\leq y_t$ can be found in $O(t+1)$ time, when t points are reported.*

Proof. Querying the RMQ-structure at a node v that contains t_v member points will return at most t_v+1 non-member points. We only query the RMQ-structure of a node v if we have already reported its point as a member point. Summing over all visited nodes we get a total cost of $O\left(\sum_v (2t_v+1)\right) = O(t+1)$. □

In order to query the whole structure, we first process a 3-sided query $[x_\ell, x_r] \times]-\infty, y_t]$ by searching for x_ℓ and x_r in the interpolation search ree. The two accessed leaves ℓ, r of the interpolation search tree comprise leaves of T as well. We traverse T from ℓ and r to the root. Let P_ℓ (resp. P_r) be the root-to-leaf path for ℓ (resp. r) in T and let $P_m = P_\ell \cap P_r$. During the traversal we also record the index of the traversed child. When we traverse a node u on the path $P_\ell - P_m$ (resp. $P_r - P_m$), the recorded index comprises the leftmost (resp. rightmost) margin of a query to the RMQ-structure of u. Thus all accessed children by the RMQ-query will be completely contained in the query's x-range $[x_\ell, x_r]$. Moreover, by Lemma 2.3 the RMQ-structure returns all member points in T_u.

For the lowest node in P_m, i.e. the lowest common ancestor (LCA) of ℓ and r, we query the RMQ-structure for all subtrees contained completely within x_ℓ and x_r. We don't execute RMQ-queries on the rest of the nodes of P_m, since they root subtrees that overlap the query's x-range. Instead, we merely check if the x- and y-coordinates of their stored point lies within the query. Since the paths P_m, $P_\ell - P_m$ and $P_r - P_m$ have length $O(\log\log n)$, the query time of T becomes $O(\log\log n + t)$. When the x-coordinates are smoothly distributed, the query to the interpolation search tree takes $O(\log\log n)$ expected time with high probability [MT93]. Hence the total query time is $O(\log\log n + t)$ expected with high probability.

Update Operation Before we describe the update algorithm of the data structure, we will first prove some properties of updating the points in T. Suppose that we decrease the y-value of a point p_u at node u to the value $y' < y$. Let v be the ancestor node of u highest in the tree whose point p_{y-min} has y-coordinate bigger than y'. We remove p_u from u. This creates an "empty slot" that has to be filled by the point of u's child with smallest y-coordinate. The same procedure has to be applied to the affected child, thus causing a *"bubble down"* of the empty slot until a node is reached with no points at its children. Next we replace v's point p_v with p_u (*swap*). We find the child of v that contains the leaf corresponding to p_v and swap its point with p_v. The procedure recurses on this child until an empty slot is found to place the last swapped out point (*"swap down"*). In case of increasing the y-value of a node the update to T is the same, except that p_u is now inserted at a node along the path from u to the leaf corresponding to p_u.

For every swap we will have to rebuild the RMQ-structures of the parents of the involved nodes, since the RMQ-structures are static data structures. This has a linear cost to the size of the RMQ-structure.

Lemma 2.4 *Let i be the highest level where the point has been affected by an update. Rebuilding the RMQ-structures due to the update takes $O(w_i^{c_2-1})$ time.*

Proof. The executed "bubble down" and "swap down", along with the search for v, traverse at most two paths in T. We have to rebuild all the RMQ-structures that lie on the two v-to-leaf paths, as well as that of the parent of the top-most node of the two paths. The RMQ-structure of a node at level j is proportional to its degree, namely $O\left(w_j/w_{j-1}\right)$. Thus, the total time becomes $O\Big(\sum_{j=1}^{i+1} w_j/w_{j-1}\Big) = O\Big(\sum_{j=0}^{i} w_j^{c_2-1}\Big) = O\Big(w_i^{c_2-1}\Big)$. □

To insert a point p, we first insert it in the interpolation search tree. This creates a new leaf in T, which might cause several of its ancestors to overflow. We split them as described in Section 2.4. For every split a new node is created that contains no point. This empty slot is filled by "bubbling down" as described above. Next, we search on the path to the root for the node that p should reside according to the Min-Heap Property and execute a "swap down", as described above. Finally, all affected RMQ-structures are rebuilt.

To delete point p, we first locate it in the interpolation search tree, which points out the corresponding leaf in T. By traversing the leaf-to-root path in T, we find the node in T that stores p. We delete the point from the node and "bubble down" the empty slot, as described above. Finally, we delete the leaf from T and rebalance T if required. Merging two nodes requires one point to be "swapped down" through the tree. In case of a share, we additionally "bubble down" the new empty slot. Finally we rebuild all affected RMQ-structures and update the interpolation search tree.

Analysis. We assume that the point to be deleted is selected uniformly at random among the points stored in the data structure. Moreover, we assume that the inserted points have their x-coordinates drawn independently at random from an $(n^{\alpha}, n^{1/2})$-smooth distribution for a constant $1/2<\alpha<1$, and that the y-coordinates are drawn from an arbitrary distribution. Searching and updating the interpolations search tree needs $O(\log\log n)$ expected with high probability [MT93,KMS+06], under the same assumption for the x-coordinates.

Lemma 2.5 *Starting with an empty weight balanced exponential tree, the amortized time of rebalancing it due to insertions or deletions is $O(1)$.*

Proof. A sequence of n updates requires at most $O(n/w_i)$ rebalancings at level i (Lemma 2.2). Rebuilding the RMQ-structures after each rebalancing costs $O\left(w_i^{c_2-1}\right)$ time (Lemma 2.4). We sum over all levels to get a total time of $O(\sum_{i=1}^{height(T)} \frac{n}{w_i} \cdot w_i^{c_2-1}) = O(n\sum_{i=1}^{height(T)} w_i^{c_2-2}) = O(n)$, when $c_2<2$. □

Lemma 2.6 *The expected amortized time for inserting or deleting a point in a weight balanced exponential tree is $O(1)$.*

Proof. The insertion of a point creates a new leaf and thus T may rebalance, which by Lemma 2.5 costs $O(1)$ amortized time. Note that the shape of T only depends on the sequence of updates and the x-coordinates of the points that have been inserted. The shape of T is independent of the y-coordinates, but the assignment of points to the nodes of T follows uniquely from the y-coordinates, assuming all y-coordinates are distinct. Let u be the ancestor at level i of the leaf for the new point p. For any integer $k \geq 1$, the probability of p being inserted at u or an ancestor of u can be bounded by the probability that a point from a leaf of T_u is stored at the root down to the k-th ancestor of u plus the probability that the y-coordinate of p is among the k smallest y-coordinates of the leaves of T. The first probability is bounded by $\sum_{j=i+k}^{height(T)} \frac{2w_{j-1}}{\frac{1}{2}w_j}$, whereas the second probability is bounded by $k/\frac{1}{2}w_i$. It follows that p ends up at the

i-th ancestor or higher with probability at most $O\Big(\sum_{j=i+k}^{height(T)} \frac{2w_{j-1}}{\frac{1}{2}w_j} + \frac{k}{\frac{1}{2}w_i}\Big) = O\Big(\sum_{j=i+k}^{height(T)} w_{j-1}^{1-c_2} + \frac{k}{w_i}\Big) = O\Big(w_{i+k-1}^{1-c_2} + \frac{k}{w_i}\Big) = O\Big(w_i^{(1-c_2)c_2^{k-1}} + \frac{k}{w_i}\Big) = O\Big(\frac{1}{w_i}\Big)$ for $c_2 = 3/2$ and $k = 3$. Thus the expected cost of "swapping down" p is $O\Big(\sum_{i=1}^{height(T)} \frac{1}{w_i} \cdot \frac{w_{i+1}}{w_i}\Big) = O\Big(\sum_{i=1}^{height(T)} w_i^{c_2-2}\Big) = O\Big(\sum_{i=1}^{height(T)} c_1^{(c_2-2)c_2^i}\Big) = O(1)$ for $c_2 < 2$. A deletion results in "bubbling down" an empty slot, whose cost depends on the level of the node that contains it. Since the point to be deleted is selected uniformly at random and there are $O(n/w_i)$ points at level i, the probability that the deleted point is at level i is $O(1/w_i)$. Since the cost of an update at level i is $O(w_{i+1}/w_i)$, we get that the expected "bubble down" cost is $O\left(\sum_{i=1}^{height(T)} \frac{1}{w_i} \cdot \frac{w_{i+1}}{w_i}\right) = O(1)$ for $c_2 < 2$. $\square$

Theorem 2.7 *There exists a dynamic data structure for the word-RAM model that supports 3-sided range reporting queries in* $O(\log\log n + t)$ *expected time with high probability, supports updates in* $O(\log\log n)$ *expected amortized time, using* $O(n)$ *space, under the assumption that the* x*-coordinates of the points are continuously drawn from a* (n^α, n^δ)*-smooth distribution, for constants* $0 < \alpha, \delta < 1$*, the* y*-coordinates are continuously drawn from any arbitrary distribution, and that the stored points are deleted with equal probability.*

External Memory

We now convert our internal memory into a solution for the I/O model. First we substitute the interpolation search tree with its I/O-efficient variant, the interpolation search B-tree [KMM+05]. We implement every consecutive $\Theta(B^2)$ leaves of the interpolation search B-tree with the data structure of Arge et al. [ASV99]. Each such structure constitutes a leaf of a weight balanced exponential tree T that we build on top of the $O(n/B^2)$ leaves.

In T every node now stores B points sorted by y-coordinate, such that the maximum y-coordinate of the points in a node is smaller than all the y-coordinates of the points of its children (Min-Heap Property). The B points with overall smallest y-coordinates are stored at the root. At a node u we store the B points from the leaves of T_u with smallest y-coordinates that are not stored at an ancestor of u. At the leaves we consider the B points with smallest y-coordinate among the remaining points in the leaf to comprise this list. Moreover, we define the weight parameter of a node at level i to be $w_i = B^{2\cdot(7/6)^i}$. Thus we get $w_{i+1} = w_i^{7/6}$, which yields a height of $\Theta(\log\log_B n)$. Let $d_i = \frac{w_i}{w_{i-1}} = w_i^{1/7}$ denote the *degree parameter* for level i. All nodes at level i have degree $O(d_i)$. We store an array in every node stores that indexes the children according to their x-order. To identify the children with respect to their y-coordinates, we replace the structure for range minimum queries of the internal memory solution with a table. For every possible interval $[k, l]$ over the children of the node, we store in an entry of the table the points of the children that belong to this interval, sorted by y-coordinate. Since every node at level i has degree $O(d_i)$, there are $O(d_i^2)$ different intervals and for each interval we store $O(B \cdot d_i)$ points. Thus, the total size of this table is $O(B \cdot d_i^3)$ points or $O(d_i^3)$ disk blocks.

Corollary 2.2 *The I/O-efficient data structure occupies $O(n/B)$ blocks.*

Proof. The interpolation search B-tree consumes $O(n/B)$ blocks [KMM$^+$05]. Moreover, there are $O(n/B^2)$ leaves at T, each of which contains $O(B^2)$ points. Thus the leaves contain $O(n)$ points that are stored in $O(n/B)$ blocks. An internal node at level $i \geq 1$ of T contains B points plus the points in its table, which are $O(B \cdot d_i^3)$. There are n/w_i nodes at level i, thus the total space of level i is $O\left(\frac{n}{w_i} \cdot B \cdot d_i^3\right) = O\left(n \cdot B \cdot \frac{1}{w_i^{4/7}}\right)$. The total space of T is bounded by $O\left(\sum_{i=1}^{height(T)} \frac{n \cdot B}{(B^{2 \cdot \frac{7}{6}^i})^{\frac{4}{7}}}\right) = O(n)$ points, i.e. $O(n/B)$ disk blocks. □

Query Operation The query is similar to the internal memory construction. First we access the interpolations search B-tree, spending $O(\log_B \log n)$ expected I/Os with high probability, given that the x-coordinates are smoothly distributed [KMM$^+$05]. This points out the leaves of T that contain the leftmost point to the right of x_ℓ and the rightmost point to the left of x_r. We perform a 3-sided range query at the two leaf structures. Next, we traverse upwards the leaf-to-root path P_ℓ (resp. P_r) on T, while recording the index k (resp. l) of the traversed child in the table. That costs $\Theta(\log \log_B n)$ I/Os. At each node we report the points of the node that belong to the query range. For all nodes on $P_\ell - P_r$ and $P_r - P_\ell$ we query as follows: We access the table at the appropriate children range, recorded by the index k and l. These ranges are always $[k+1,$last child$]$ and $[0, l-1]$ for the node that lie on $P_\ell - P_r$ and $P_r - P_\ell$, respectively. The only node where we access a range $[k+1, l-1]$ is the lowest common ancestor of the leaves that contain a and b. The recorded indices facilitate access to these entries in $O(1)$ I/Os. We scan the list of points sorted by y-coordinate, until we reach a point with y-coordinate bigger than y_t. All scanned points are reported. If the scan has reported all B elements of a child node, the query proceeds recursively to that child, since more member points may lie in its subtree. Note that for these recursive calls, we do not need to access the B points of a node v, since we accessed them in v's parent table. The accessed table entries contain the complete range of children. If the recursion accesses a leaf, we execute a 3-sided query on it, with respect to x_ℓ and x_r [ASV99].

The list of B points in every node can be accessed in $O(1)$ I/Os. The construction of [ASV99] allows us to load the B points with minimum y-coordinate in a leaf also in $O(1)$ I/Os. Thus, traversing P_ℓ and P_r costs $\Theta(\log \log_B n)$ I/Os worst case. There are $O(\log \log_B n)$ nodes u on $P_\ell - P_m$ and $P_r - P_m$. The algorithm recurses on nodes that lie within the x-range. Since the table entries that we scan are sorted by y-coordinate, we access only points that belong to the answer. Thus, we can charge the scanning I/Os to the output. The algorithm recurses on all children nodes whose B points have been reported. The I/Os to access these children can be charged to their points reported by their parents, thus to the output. That allows us to access the child even if it contains only $o(B)$ member points to be reported. The same property holds also for the access to the leaves. Thus we can perform a query on a leaf in $O(t/B)$ I/Os. Summing up, the worst case query complexity of querying T is $O(\log\log_B n + \frac{t}{B})$ I/Os. Hence in total the query costs $O(\log\log_B n + \frac{t}{B})$ expected I/Os with high probability.

Update Operation Insertions and deletions of points are analogous to the internal solution. For the case of insertions, first we update the interpolation search B-tree. This creates a new leaf in the interpolation search B-tree that we also insert at the appropriate leaf of T in $O(1)$ I/Os [ASV99]. This might cause some ancestors of the leaves to overflow. We split these nodes, as in the internal memory solution. For every split B empty slots "bubble down". Next, we update T with the new point. For the inserted point p we locate the highest ancestor node that contains a point with y-coordinate larger than p's. We insert p in the list of the node. This causes an excess point, namely the one with maximum y-coordinate among the B points stored in the node, to "swap down" towards the leaves. Next, we scan all affected tables to replace a single point with a new one.

In case of deletions, we search the interpolation search B-tree for the deleted point, which points out the appropriate leaf of T. By traversing the leaf-to-root path and loading the list of B point, we find the point to be deleted. We remove the point from the list, which creates an empty slot that "bubbles down" T towards the leaves. Next we rebalance T as in the internal solution. For every merge we need to "swap down" the B largest excess points. For a share, we need to "bubble down" B empty slots. Next, we rebuild all affected tables and update the interpolation search B-tree.

Analysis. Searching and updating the interpolation search B-tree requires $O(\log_B \log n)$ expected I/Os with high probability, when the x-coordinates are drawn from an $(\frac{n}{(\log\log n)^{1+\varepsilon}}, n^{1/B})$-smooth distribution, for $\varepsilon>0$ [KMM+05].

Lemma 2.7 *For every path corresponding to a "swap down" or a "bubble down" starting at level i, the cost of rebuilding the tables of the paths is $O\big(d_{i+1}^3\big)$ I/Os.*

Proof. Analogously to Lemma 2.4, a "swap down" or a "bubble down" traverse at most two paths in T. A table at level j costs $O(d_j^3)$ I/Os to be rebuilt, thus all tables on the paths need $O\Big(\sum_{j=1}^{i+1} d_j^3\Big) = O\big(d_{i+1}^3\big)$ I/Os. □

Lemma 2.8 *Starting with an empty external weight balanced exponential tree, the amortized I/Os for rebalancing it due to insertions or deletions is $O(1)$.*

Proof. We follow the proof of Lemma 2.5. Rebalancing a node at level i requires $O\big(d_{i+1}^3 + B \cdot d_i^3\big)$ I/Os (Lemma 2.7), since we get B "swap downs" and "bubble downs" emanating from the node. For a sequence of n updates the I/Os are $O\big(\sum_{i=1}^{height(T)} \frac{n}{w_i}\cdot(d_{i+1}^3 + B\cdot d_i^3)\big) = O\big(n \cdot \sum_{i=1}^{height(T)} w_i^{-1/2} + B\cdot w_i^{-4/7}\big) = O(n)$. □

Lemma 2.9 *The expected amortized I/Os for inserting or deleting a point in an external weight balanced exponential tree is* $O(1)$.

Proof. By similar arguments as in Lemma 2.6 and considering that a node contains B points, we bound the probability that point p ends up at the i-th ancestor or higher by $O(B/w_i)$. An update at level i costs $O(d_{i+1}^3) = O\left(w_i^{1/2}\right)$ I/Os. Thus "swapping down" p costs $O\big(\sum_{i=1}^{height(T)} w_i^{1/2} \cdot \frac{B}{w_i}\big) = O(1)$ expected I/Os. $O\left(\sum_{j=i+k}^{height(T)} B \cdot \frac{2w_{j-1}}{\frac{1}{2}w_j} + \frac{k \cdot B}{\frac{1}{2}w_i}\right) =$ $O\left(\sum_{j=i+k}^{height(T)} B \cdot \frac{1}{w_{j-1}^{1/6}} + \frac{k \cdot B}{w_i}\right) = O\left(B \cdot \frac{1}{w_{i+k-1}^{1/6}} + \frac{k \cdot B}{w_i}\right) = O\left(\frac{B}{w_i}\right)$ for $k = 11$. In case of deletions, following a similar argument as Lemma 2.6 of the internal solution, we get that the cost of an update at level i is $O\left(w_i^{1/2}\right)$ I/Os. Thus, the expected I/Os for the "bubble down" are $O\left(\sum_{i=1}^{height(T)} \frac{B}{w_i} \cdot w_i^{1/2}\right)$ $=O(1)$.□

Theorem 2.8 *There exists a dynamic data structure for the I/O model that supports 3-sided range reporting queries in* $O(\log\log_B n + t)$ *expected I/Os with high probability, supports updates in* $O(\log_B \log n)$ *amortized expected I/Os, using* $O(n/B)$ *blocks of space, under the assumption that the* x*-coordinates of the points are continuously drawn from a* $(n/(\log\log n)^{1+\epsilon}, n^{1/B})$*-smooth distribution, for a constant* $0 < \epsilon$*, the* y*-coordinates are continuously drawn from any arbitrary distribution, and the stored points are deleted with equal probability.*

2.5 Open Problems

- Can the solutions for external memory in Sections 2.2 and 2.3 attain exactly the complexities of their internal memory counterparts? That requires external memory priority search trees [ASV99] with $O(\log_B n)$ **worst case** update I/Os instead of amortized. Is there an I/O-efficient static data structures that supports 3-sided range reporting queries in $O(1 + t/B)$ **worst case** I/Os [KPS$^+$10]?

- Can we modify the solution of Section 2.4 for the arbitrary distribution of the y-coordinated, such that it attains exactly the complexities of the other solutions that impose more restricting assumptions to the input? More precisely, when the y-coordinates are arbitrarily distributed, can the update time become expected amortized time with high probability? Can the query complexity for the external memory variant become $O(\log_B \log n)$ I/Os instead of $O(\log\log_B n)$ I/Os?

- The dynamic interpolation search trees of [KMS$^+$06] have the most preferrable properties, as they can manage efficiently elements that are drawn from the broadest smooth distribution, from discrete distributions and from distibutions with unbounded density. An interesting open problem is to adapt this implementation to the external memory, and thus improve upon the existing interpolation search B-trees [KMM$^+$05].

Chapter 3

Dynamic Planar Range Maxima Queries

Abstract

We consider the problem of maintaining dynamically in internal memory a set of points in the plane, and supporting planar orthogonal range maxima reporting queries efficiently in the worst case. Let P be a set of n points in the plane. A point is *maximal* if it is not dominated by any other point in P.

We present a dynamic data structure for the pointer machine model that supports reporting the maximal points among the points that lie within a given planar orthogonal 3-sided range $[x_\ell, x_r] \times [y_b, +\infty[$ in $O(\log n + t)$ worst case time, and supports insertions and deletions of points in optimal $O(\log n)$ worst case time, using $O(n)$ space, where t is the size of the query's output. This is the first dynamic data structure that supports these queries and the updates in logarithmic worst case time.

We adapt the data structure to the RAM model with word size w, where the coordinates of the points are integers in the range $U{=}\{0, \ldots, 2^w{-}1\}$. We present a linear space data structure that supports 3-sided range maxima reporting queries in optimal $O(\frac{\log n}{\log \log n} + t)$ worst case time, and supports updates in $O(\frac{\log n}{\log \log n})$ worst case time, using $O(n)$ space. These are the first sublogarithmic worst case bounds for all operations in the RAM model.

Finally, we present a dynamic data structure for the pointer machine model that supports reporting the maximal points that lie within a given 4-sided planar orthogonal range $[x_\ell, x_r] \times [y_b, y_t]$ in $O(\log^2 n + t)$ worst case time, and support updates in $O(\log^2 n)$ worst case update time, using $O(n \log n)$ space. This improves by a logarithmic factor the space or the worst case deletion time of the previous strutures for the dynamic rectangular visibility query problem.

3.1 Introduction

In this Chapter we present dynamic data structures that support planar orthogonal range maxima reporting queries and support insertions and deletions of points efficiently in the worst case. Maintaining the maximal points is not difficult, when only insertions are allowed. It suffices to store the maximal points in a binary search tree, in order to support the insertions in $O(\log m)$ time. However deletions cannot be processed efficiently, when only the maximal points are stored. A deletion of a maximal point p may cause the maximal points among the points dominated by p to become maximal for the whole pointset. The fact that deletions modify the query's output significantly is the reason that previous results attain $O(\log^2 n)$ deletion time. Kapoor [Kap00] attains $O(\log n)$ worst case time by delaying the recomputation of maximal points for the time they are returned by the query operation. However, this only allows for $O(\log n)$ *amortized* query time.

The structure of [OvL81] consists of a balanced binary search tree where every node contains a linked list that stores the maximal points among the points stored in the subtree of the node. Given that the linked lists in two sibling nodes are already computed, the maximal points of their common parent node are computed by moving to the parent node the points in the linked list of the left child with y-coordinate larger than the largest y-coordinate of the points stored in the right child, and the linked list in the right child. The rest of the linked list of the left child remains stored in the left child. In this way all the maximal points are stored in the root, and an update operation involves a bottom-up computation that takes $O(\log^2 n)$ time.

In Section 3.2 we present a pointer based data structure that maintains the maximal points and supports the insertion and deletion of a point in $O(\log n)$ worst case time, using $O(n)$ space. Comparisons are the only allowed computation on the coordinates of the points. Our structure follows the basic approach of previous structures. We store the points sorted by x-coordinate at the leaves of a tree, and maintain a *tournament* in the internal nodes of the tree with respect to their y-coordinates. An internal node u is assigned the point with maximum y-coordinate in its subtree. We observe that the nodes in the subtree of u that contain this point form a path. We also observe that the maximal points among the points assigned to the nodes hanging to the right of this path are also maximal among the points in the subtree of u. We store these representative points in node u, so that the query algorithm accesses recursively only points to be reported.

The implementation of our structures is based on the fact that we can obtain the set of points we store in a node from the set stored in its child node that belongs to the same path. In particular if that is the left child, we need to delete the points that are dominated by the point assigned to the right child and insert this point into the set. Sundar [Sun89] shows how to implement a priority queue that supports exactly this operation (*attrition*) in $O(1)$ worst case time. In particular, given a set of elements from a totally ordered universe, a *priority queue with attrition (PQA)* supports the operations DeleteMax, which deletes and returns the maximum element from the

PQA, and InsertAndAttrite(x), which inserts element x into the PQA and removes all elements smaller than x from the PQA (attrition). PQAs uses space linear to the number of inserted elements, and support both operations in $O(1)$ worst case time. Operation InsertAndAttrite(x) has to find the largest element that is smaller than x. However, this can not be achieved in $O(1)$ worst case time by searching among the inserted elements. A PQA is implemented as a doubly linked list with a constant number of additional pointers that facilitate removing the elements that have been attrited incrementally. In this way the elements are only attrited semantically. Namely, the attrited elements may remain in the structure, but they are never returned by DeleteMax. Given a value x, the elements in the PQA that are larger than x can be reported in sorted order in $O(t + 1)$ time, where t is the size of the output.

Instead of a linked list, we augment every node with a PQA that contains the representative points. This allows us to complete a bottom-up recomputation of the secondary structures in $O(\log n)$ worst case time, using the operation InsertAndAttrite.The query operation utilizes the DeleteMax operation to report identify the next point to be reported. Moreover, in Lemma 3.3 we prove that our implementation moreover supports the more general planar orthogonal 3-sided range maxima reporting queries in $O(\log n + t)$ worst case time, where t is the number of reported points.

To achieve linear space we implement every path that contains the same point as a *partially persistent* priority queue with attrition. The priority queue with attrition abides by the assumptions of the methods for partial persistence. However, we do not use the node-copying method [DSST89] to make the priority queues with attrition partially persistent, since the time overhead is amortized. Instead we use the method of Brodal [Bro96], since it has a worst case ovehead. This allows the *rollback* operation that discards the latest version and renders its preceding version updatable. Since the overhead is constant, discarding the secondary structures in a top-down fashion along a root-to-leaf path takes $O(\log n)$ worst case time. Thus a point is inserted and deleted by a updating the secondary structures along a root-to-leaf path in $O(\log n)$ time.

This is the first dynamic search tree that explicitly uses partially persistent secondary structures. The structure of [OvL81] can be seen as an implicit implementation of partially persistent linked lists. In fact, our technique can be used to augment any search tree, as long as the secondary structure of a node in the tree is obtained by updating the secondary structure in *only one* of its children nodes. The major benefit of this approach is that for our problem a deletion takes essentially $O(\text{insertion})$ time, even if the deletion modifies the output of the query significantly. Moreover, the technique obviates tedious case analysis that usually arises when designing algorithms for deletion. Instead of deleting a point, the structure in essence “forgets” how the point was inserted, due to the rollback operation.

In Section 3.3 we adapt the above structure to the word-RAM model by increasing the degree of the tree to $\Theta(\log^{\varepsilon} n)$ for some $1/4<\varepsilon<1$. To search and update a node in $O(1)$ time, we use precomputed tables that solve dynamic 3-sided range maxima reporting for a set of $O(\log^{\frac{1}{4}} n)$ points, and the *Q-heap* of Fredman and Willard [FW94]. They are dynamic data structures for the word-RAM model that supports predecessor searching, insertions and deletions of integers in $O(1)$ worst case time, when the stored set contains at most $w^{1/4}$ integers. They use $O(w^{1/4})$ space, and utilizes an $O(2^w)$ space global precomputed lookup-table that needs $O(2^w)$ preprocessing time. Our adaptation is similar to the implementation of *fusion trees* [FW93]. They support predecessor searching, insertions and deletions of integers in $O(\frac{\log n}{\log\log\ n})$ worst case time, using $O(n)$ space. In particular, fusion trees are (a, b)-trees with $b = O(\log^{1/4} n)$, that are augmented with a constant number of Q-heaps per internal node.

In Section 3.4 we present a dynamic data structure that supports 4-sided range maxima queries. The result is obtained by following the approach of [OW88]. Namely we augment $BB[\alpha]$-trees [WL85] with our dynamic structure for 3-sided range maxima reporting queries of Section 3.2.

Optimality Brodal et al. [BCR96] show that if the insertion and deletion of an element requires t comparisons, then operation FindMax requires at least $\frac{n}{e2^{2t}-1}$ comparisons. Mapping element x to the point (x, x), inserts and deletes the points along the diagonal $x = y$. The only maximal point corresponds to the element returned by FindMax and can be computed in $O(1)$ worst case time. Thus $O(1) \geq \frac{n}{e2^{2t}-1} \Rightarrow t = \Omega(\log n)$. Alstrup et al. [AHR98] show that if the insertion and deletion of a two-dimensional point requires $O(\text{polylog}n)$ cell probes, deciding whether a given query point is dominated by a point or not requires $\Omega\left(\frac{\log n}{\log\log n}\right)$ cell probes. The answer can be inferred by whether the query point lies above or below the staircase. The lower bound also holds for the problem of reporting the maximal points.

3.2 3-Sided Maxima Queries in the Pointer Machine

We store the points sorted by increasing x-coordinate at the leaves of a red-black tree [GS78] T. We maintain a *tournament* on the internal nodes of T with respect to the y-coordinates of the points. In particular, every internal node u contains the points $p_{x-\max}(u)$ and $p_{y-\max}(u)$ that are respectively the points with maximum x- and y-coordinate among the points in the subtree of u. Points $p_{x-\max}(u)$ allow us to search in T with respect to the x-coordinate. Points $p_{y-\max}(u)$ define n disjoint *winning paths* in T whose nodes contain the same point. Let u be an internal node. Let u belong to a winning path π. We denote by π_u the suffix of π that starts at node u and ends at the leaf that stores $p_{y-\max}(u)$. We denote by R_u the set of right children of the nodes in π_u that do not belong to π_u themselves. We denote by $\text{MAX}(u)$ the points that are maximal among $\{p_{y-\max}(v) \mid v \in R_u\}$. We can obtain $\text{MAX}(u)$ from the set of points $\text{MAX}(c)$, where c is the child of u that belongs to π_u. In particular, if that is the right child u_R then $R_u = R_{u_R}$ and thus $\text{MAX}(u) = \text{MAX}(u_R)$. Otherwise if that is the left child u_L, then $R_u = R_{u_L} \cup \{u_R\}$. Point $p_{y-\max}(u_R)$ belongs to $\text{MAX}(u)$ since it has the largest x-coordinate among all points in R_u. Moreover, all the points in $\text{MAX}(u_L)$ with y-coordinate at most $p_{y-\max}(u_R)_y$ should be excluded from $\text{MAX}(u)$ since they are dominated by $p_{y-\max}(u_R)$. Figure 3.1 illustrates R_u and $\text{MAX}(u)$ for a generic winning path.

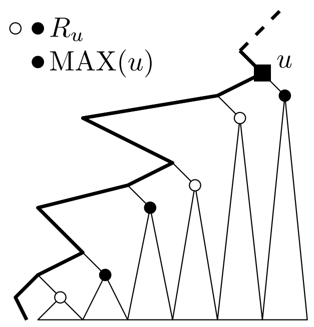

Figure 3.1: The winning path π is bold. The circular nodes are in R_u. The black circular nodes are in MAX(u).

We implement the sets MAX(u) of the nodes u along a winning path π as the versions of a *partially persistent priority queue with attrition* (PPPQA). That is, every internal node u stores the y-coordinates of the points in MAX(u) in a priority queue with attrition (PQA). If u is the i-th node of π, then it contains the i-th version of the PPPQA. The leaf of π contains version 0. The child of u that belongs to π contains the $i-1$-th version. If that is u_R, the i-th version and the $i-1$-th versions have the same content. Else if that is u_L, the i-th version is obtained by executing InsertAndAttrite($p_{y-\max}(u_R)_y$) to the $i-1$-th version. Moreover, for every point in MAX(u) we store a pointer to the node of R_u that stores the point. This node is the highest node of its winning path. Note that R_u is not explicitly maintained anywhere in the data structure.

Let p be the point with maximum y-coordinate among the points $p_{y-\max}(u_L)$ and $p_{y-\max}(u_R)$. To *extend* the winning path that contains p from the child node to node u, we assign to u the points $p_{x-\max}(u_R)$ and p, and compute MAX(u).

Lemma 3.1 *Extending a winning path from an internal node to its parent node needs $O(1)$ time and $O(1)$ extra space.*

Proof. Points $p_{x-\max}(u)$ and $p_{y-\max}(u)$ can be computed from the children of u in $O(1)$ time. To extend the winning path that contains $p_{y-\max}(u_R)$, we create the reference to the new version of the PPPQA for the winning path in $O(1)$ time. To extend the winning path that contains $p_{y-\max}(u_L)$, we record persistently the operation InsertAndAttrite($p_{y-\max}(u_R)_y$) and create a reference to the new version in $O(1)$ worst case time using $O(1)$ extra space [Bro96, Sun89]. □

Lemma 3.1 implies that T can be constructed bottom-up in $O(n)$ worst case time, assuming that the n points are given sorted by increasing x-coordinate. The total space usage is $O(n)$.

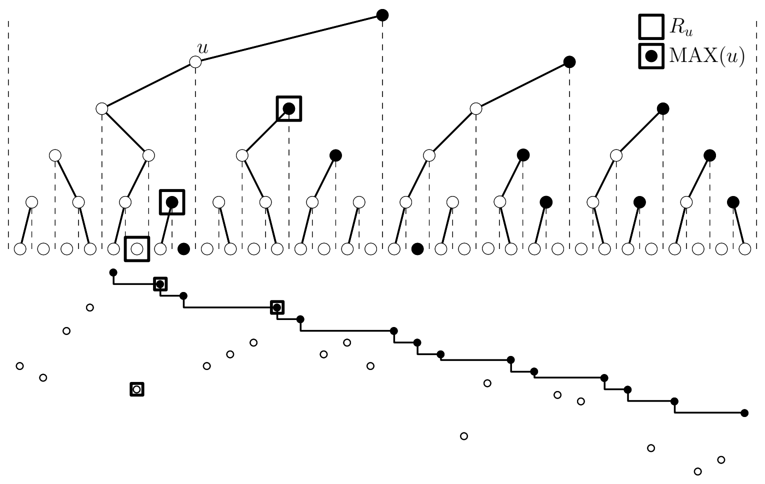

Figure 3.2: A concrete example of the structure for planar orthogonal 3-sided range maxima reporting queries for a given pointset.

Query Operation

In the following we describe how to answer 3-sided range maxima queries. This immediately also gives us maxima dominance and maxima contour queries, since these are special cases where the query ranges are $[q_x, +\infty[\times[q_y, +\infty[$ and $]-\infty, x_r]\times]-\infty, +\infty[$ respectively. Figure 3.2 shows a concrete example of our structure for a particular pointset.

Reporting the maximal points that lie above q_y We first show how to report all maximal points with y-coordinate larger than a given value q_y among the points that belong to a subtree T_u of T rooted at an internal node u. If $p_{y-\max}(u)_y \le q_y$ we terminate since no maximal point of T_u has to be reported. Otherwise we report $p_{y-\max}(u)$ and compute the prefix $p[1], \ldots, p[k]$ of $\mathrm{MAX}(u)$ of points with y-coordinate larger than q_y. To do so we report the elements of the PQA of u that are larger than q_y. Let u_i be the node of R_u such that $p_{y-\max}(u_i) = p[i]$. We report recursively the maximal points in T_{u_i} with y-coordinate larger than $p[i+1]_y$ for $i \in \{1, \ldots, k-1\}$, and larger than q_y for $i = k$. The algorithm reports the maximal points in T_u in decreasing y-coordinate and terminates when the maximal point with the smallest y-coordinate larger than q_y is reported.

For the correctness of the above observe that the point $p_{y-\max}(u)$ is the leftmost maximal point among the points in T_u. The x-coordinates of the rest of the maximal points in T_u are larger than $p_{y-\max}(u)_x$. The subtrees rooted at nodes of R_u divide the x-range $]p_{y-\max}(u)_x, +\infty[$ into disjoint x-ranges. The point p' with maximum y-coordinate of each x-range is stored at a node u' of R_u. The points in $\mathrm{MAX}(u)$ are maximal among the points in T_u. Let $p[i+1]$ be the leftmost maximal point in T_u to the right of p'. If p' does not belong to $\mathrm{MAX}(u)$ then none of the points in $T_{u'}$ are maximal, since $p'_y \le p[i+1]_y$ and $p[i+1]_x$ is larger than the x-coordinate of all points in $T_{u'}$. Otherwise

p' belongs to $\mathrm{MAX}(u)$ and more precisely $p' = p[i]$. Point $p[i]$ is the leftmost maximal point among the points in $T_{u'}$. The maximal points among the points in T_u with x-coordinate at least $p[i]_x$ and y-coordinate larger than $p[i+1]_y$ belong to $T_{u'}$. In particular, they are the maximal points among the points in $T_{u'}$ with y-coordinate larger than $p[i+1]_y$.

Lemma 3.2 *Reporting the maximal points with y-coordinate larger than q_y among the points of a subtree of T takes $O(t+1)$ worst case time, where t is the number of reported points.*

Proof. If $p_{y-\max}(u)$ is not reported we spend in total $O(1)$ time to assess that no point will be reported. Otherwise $p_{y-\max}(u)$ is reported. We need $O(k+1)$ time to compute the k points in $\mathrm{MAX}(u)$ with y-coordinate larger than q_y. If $k = 0$ we charge the $O(1)$ time we spent in node u to the reporting of $p_{y-\max}(u)$. Otherwise we charge the time to compute $p[i]$ and to access node u_i to the reporting of $p[i]$. In total every reported point is charged $O(1)$ time. □

In order to report all maximal points of P we apply the above algorithm to the root of T with $q_y{=}-\infty$. The total time is $O(m)$.

3-sided Range Maxima Queries We show how to report all maximal points of the point set $P \cap ([x_\ell, x_r] \times [y_b, +\infty[)$. We search for the leaves ℓ and r of T that contain the points with the largest x-coordinate smaller than x_r and smallest x-coordinate larger than x_ℓ, respectively. Let π_ℓ and π_r be the root-to-leaf paths to ℓ and r, respectively. Let R denote the set of right children of nodes of $\pi_\ell \setminus \pi_r$ that do not belong to π_ℓ themselves, and the set of left children of nodes of $\pi_r \setminus \pi_\ell$ that do not belong to π_r themselves. The subtrees rooted at the nodes u of R divide the x-range $]x_\ell, x_r[$ into disjoint x-ranges. We compute the maximal points $p[1], \ldots, p[k]$ among the points $p_{y-\max}(u)$ in R with y-coordinate larger than y_b using [KLP75]. Let u_i be the node of R such that $p_{y-\max}(u_i) = p[i]$. We report recursively the maximal points in T_{u_i} with y-coordinate larger than $p[i{+}1]_y$ for $i{\in}\{1, \ldots, k{-}1\}$, and larger than y_b for $i{=}k$.

Lemma 3.3 *Reporting the maximal points for the 3-sided range maxima query takes $O(\log n + t)$ worst case time, where t is the number of reported points*

Proof. There are $O(\log n)$ nodes u in R. The points $p_{y-\max}(u)$ are accessed in decreasing x-coordinate. Therefore we can compute the maximal points $p[i], i \in \{1, \ldots, k\}$ in $O(\log n)$ time [KLP75]. Let $p[i] = p_{y-\max}(u_i)$ for a particular node u_i of R, and let there be t_i maximal points to be reported in the subtree T_{u_i}. Since $p[i]$ will be reported we get that $t_i \geq 1$. By Lemma 3.2 we need $O(t_i)$ time to report the t_i points. Therefore the total worst case time to report the $t = \sum_{i=1}^{k} t_i$ maximal points is $O(\log n + t)$. □

In order to answer whether a given query point $q = (q_x, q_y)$ lies above or below the staircase (maxima emptiness query) in $O(\log n)$ time we terminate a 3-sided range maxima query for the range $[q_x, +\infty[\times[q_y, +\infty[$ as soon as the first maximal point is reported. If so, then q lies below the staircase. Else if no point is reported then q lies above the staircase.

Update Operation

To insert (resp. delete) a point $p = (p_x, p_y)$ in the structure, we search for the leaf ℓ of T that contains the point with the largest x-coordinate smaller than p_x (resp. contains p). We traverse the nodes of the parent(ℓ)-to-root search path π top-down. For each node u of π we discard the points $p_{x-\max}(u)$ and $p_{y-\max}(u)$, and rollback the PQA of u in order to discard MAX(u). We insert a new leaf ℓ' for p immediately to the right of the leaf ℓ (resp. delete ℓ) and we rebalance T. Before performing a rotation, we discard the information stored in the nodes that participate in it. Finally we recompute the information in each node u missing $p_{x-\max}(u)$, $p_{y-\max}(u)$, and MAX(u) bottom-up following π. For every node u we extend the winning path of its child u' such that $p_{y-\max}(u) = p_{y-\max}(u')$ as in Section 3.2. Correctness follows from the fact that every node that participates in a rotation either belongs to π or is adjacent to a node of π. The winning path that ends at the rotated node will be considered when we extend the winning paths bottom-up.

Lemma 3.4 *Inserting or deleting a point in T takes $O(\log n)$ worst case time.*

Proof. The height of T is $O(\log n)$. Rebalancing a red-black takes $O(\log n)$ time. We need $O(1)$ time to discard the information in every node of π. By Lemma 3.1 we need $O(1)$ time to extend the winning path at every recomputed node. The total time to update the internal nodes of T is $O(\log n)$. □

3.3 3-Sided Maxima Queries in the word-RAM

We store the points of P sorted by increasing x-coordinate at the leaves of an (a, b)-tree T, such that $a = \frac{1}{2}\log^{\frac{1}{4}} n$ and $b = \log^{\frac{1}{4}} n$. The height of T is $O(\frac{\log n}{\log\log n})$. Every node u is assigned the points $p_{y-\max}(u)$ and $p_{x-\max}(u)$. Define $C_y(u) = \{p_{y-\max}(u_i)_y \mid u_i \text{ is a child of } u\}$ and similarly let $C_x(u)$ be the x-coordinates of the points with maximum x-coordinate assigned to the children of u. In every internal node u we store $C_x(u)$ and $C_y(u)$ in two Q-heaps $X(u)$ and $Y(u)$, respectively. We store two global lookup-tables for the Q-heaps. We store a global look-up table S that supports 3-sided range maxima queries for a set of at most $\log^{\frac{1}{4}} n$ points in rank-space. Every node u stores a reference to the entry of S that corresponds to the permutation in $C_y(u)$. We adopt the definitions for the winning paths and MAX(u) from Section 3.2, with the difference that now R_u denotes the children of nodes v of π_u that contain the second maximal point in $c_y(v)$. As in Section 3.2, we implement the sets MAX(u) of the nodes u along a winning path as the versions of a PPPQA. For every point p in MAX(u) we store a pointer to the node v of π_u whose child is assigned p. The tree and the look-up tables use $O(n)$ space.

Query Operation To report the maximal points in a subtree T_u with y-coordinate larger than q_y, we first compute the prefix $p[1], \ldots, p[k]$ of the points in MAX(u) with y-coordinate larger than q_y, as in Section 3.2. For each computed point $p[i]$ we visit the node v of π_u whose child is assigned $p[i]$. We use the

$Y(v)$ and the reference to S to compute the maximal points $p'[j], j \in \{1, \ldots, k'\}$ among the points in $C_y(v)$ that lie in the range $]p[i]_x, +\infty[\times]p[i+1]_y, +\infty[$. Let v_j be the child of v such that $p'[j] = p_{y-\max}(v_j)$. We recursively report the maximal points in the subtree T_{v_j} with y-coordinate larger than $p'[j+1]_y$ for $j \in \{1, \ldots, k'-1\}$, and larger than $p[i+1]_y$ for $j = k'$. If $i = k$ and $j = k'$ we recursively report the points in T_{v_j} with y-coordinate larger than q_y. Correctness derives from the fact that $p[i]_x < p'[1]_x < \cdots < p'[k']_x < p[i+1]_x$ and $p[i]_y > p'[1]_y > \cdots > p'[k']_y > p[i+1]_y$ hold, since $p[i]$ and $p'[1]$ are respectively the first and the second maximal points among the points in $C_y(v)$. The worst case query time is $O(t)$. To answer a 3-sided range maxima query, we first use the Q-heaps for the x-coordinates in order identify the nodes on the paths π_ℓ and π_r to the leaves that contain the points with smallest x-coordinate larger than x_ℓ and with largest x-coordinate smaller than x_ℓ, respectively. This takes $O(\frac{\log n}{\log\log n})$ worst case time. For each node on π_ℓ and π_r we identify the interval of children that are contained in the x-range of the query. For each interval we identify the child c with maximum $p_{y-\max}(c)$ using S in $O(1)$ time. These elements define the set R from which we compute the maximal points using [KLP75] in $O(\frac{\log n}{\log\log n})$ worst case time. We apply the above modified algorithm to this set, ignoring the maximal points $p'[j]$ outside the x-range of the query. The worst case time for 3-sided range maxima query is $O(\frac{\log n}{\log\log n} + t)$.

Update Operation To insert and delete a point from T we proceed as in Section 3.2. To extend a winning path to a node v we first find the child u of v with the maximum $p_{y-\max}(u)_y$ using $Y(v)$, i.e. the winning path of u will be extended to v. Then we set $p_{x-\max}(v) = p_{x-\max}(u_k)$ where u_k is the rightmost child of v, we set $p_{y-\max}(v) = p_{y-\max}(u)$, insert $p_{x-\max}(u)_x$ and $p_{y-\max}(u)_y$ to $X(v)$ and $Y(v)$ respectively, we recompute the reference to the table S using $Y(v)$, and we recompute $\mathrm{MAX}(v)$ from $\mathrm{MAX}(u)$ as in Section 3.2. In order to discard the information from a node u we remove $p_{x-\max}(u)$ and $p_{y-\max}(u)$ from node u and from the Q-heaps $X(v)$ and $Y(v)$ respectively, and rollback $\mathrm{MAX}(u)$. These operations take $O(1)$ worst case time. Rebalancing involves splitting a node and merging adjacent sibling nodes. To facilitate these operation in $O(1)$ worst case time we execute them incrementally in advance, by representing a node as a pair of nodes. Therefore we consider each Q-heap as two separate parts Q_ℓ and Q_r, and maintain the size of Q_ℓ to be exactly a. In particular, whenever we insert an element to Q_ℓ, we remove the rightmost element from Q_ℓ and insert it to Q_r. Whenever we remove an element to Q_ℓ, we remove the leftmost element from Q_r and insert it to Q_ℓ. In case Q_r is empty we remove the rightmost or leftmost element from the immediately left or right sibling node respectively and insert it to Q_ℓ. Otherwise if both sibling nodes have a elements in their Q-heaps, we merge them. The total worst case time for an update is $O(\frac{\log n}{\log\log n})$ since we spend $O(1)$ time per node.

Theorem 3.1 *Given n planar points with integer coordinates in the range $U = \{0, 1, \ldots, 2^w - 1\}$, there exists a dynamic data structure for the RAM model with word size w that supports 3-sided range maxima queries in $O(\frac{\log n}{\log\log n} + t)$ worst case time when t points are reported, and supports updates in $O(\frac{\log n}{\log\log n})$ worst case time, using $O(n)$ space.*

3.4 4-sided Maxima Queries and Rectangular Visibility

For the rectangular visibility query problem, where we want to report the points that are rectangularly visible from a given query point $q = (q_x, q_y)$, we note that it suffices to report the maximal points p that lie to the lower left quadrant defined by point q (namely the points where $p_x \leq q_x$ and $p_y \leq q_y$ holds). The rectangularly visible points of the three other quadrants can be found in a symmetric way. This corresponds exactly to a 4-sided range maxima query for the range $]-\infty, q_x] \times]-\infty, q_y]$. The rectangular visibility query problem can be solved in the same bounds, using four 4-sided range maxima structures.

To support 4-sided range maxima queries, as in [OW88], we store all points sorted by increasing y-coordinate at the leaves of a weight-balanced $BB[\alpha]$-tree S [WL85]. In every internal node u of S we associate a secondary structure that can answer 3-sided range maxima queries on the points that belong to the subtree S_u of S. To perform a 4-sided range maxima query we first search for the leaves of S that contain the point with the smallest y-coordinate larger than y_b and the point with largest y-coordinate smaller than y_t, respectively. Let π_b and π_t be the root-to-leaf search paths respectively. Let L denote the set of nodes of S that are left children of the nodes in $\pi_t \setminus \pi_b$ and do not belong to π_t themselves, and the set of nodes of S that are right children of the nodes in $\pi_b \setminus \pi_t$ and do not belong to π_b themselves. The subtrees rooted at the nodes of L divide the y-range $]y_b, y_t[$ into $O(\log n)$ disjoint y-ranges. We consider the nodes $u_1, \ldots, u_k$ of L in decreasing y-coordinate. In the secondary structure of u_1 we perform a 3-sided range maxima query with the range $]x_\ell, x_r] \times]-\infty, +\infty[$. In the secondary structure of every other node u_i of L we perform a 3-sided range maxima query with the range $]b_{i-1}, x_r] \times]-\infty, +\infty[$, where b_{i-1} is the x-coordinate of the last point reported from the secondary structures of $u_1, \ldots, u_{i-1}$, i.e. the rightmost point that lies to the left of x_r and lies in y-range spanned by the subtrees $S_{u_1}, \ldots, S_{u_{i-1}}$. See Figure 3.3 for the decomposition of a 4-sided query.

To insert (resp. delete) a point in S, we first search S and create a new leaf (resp. delete the leaf) for the point. Then we insert (resp. delete) the point to the $O(\log n)$ secondary structures that lie on the search path. Finally we rebalance the weight-balanced $BB[\alpha]$-tree S and reconstruct the secondary structures at rebalanced nodes.

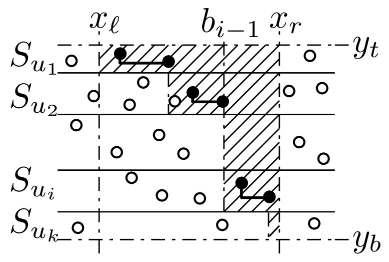

Figure 3.3: The decomposition of a 4-sided range maxima query into $O(\log n)$ 3-sided range maxima queries.

Theorem 3.2 *Given a set of n points in the plane, the dynamic 4-sided range maxima query problem can be solved using $O(n \log n)$ space, $O(\log^2 n+t)$ worst case query time, and $O(\log^2 n)$ worst case update time, when reporting t points.*

Proof. There are $k=O(\log n)$ nodes in L where we execute a 3-sided range maxima query. By Lemma 3.3 each such query takes $O(\log n + t_i)$ worst case time, where t_i is the number of reported points. In total the worst case time to report the $t = \sum_{i=1}^{k} t_i$ points is $O(\log^2 n+t)$. Each point p occurs once in every secondary structure of a node of S that is an ancestor of the leaf that contains p. There are $O(\log n)$ such ancestor nodes, thus the total space is $O(n \log n)$. By Lemma 3.4 we need $O(\log n)$ time to insert and delete a point p from each secondary structure at an ancestor node. The worst case time to insert and delete p to the secondary structures that contain it is $O(\log^2 n)$. When the incremental rebalancing algorithm of [AV03] is applied, then for each insertion and deletion to S, $O(1)$ updates are applied to the secondary structures at each of $O(\log n)$ levels of S. By Lemma 3.4 each such update needs $O(\log n)$ worst case time, thus the rebalancing costs $O(\log^2 n)$ worst case time. □

3.5 Open Problems

- An immediate adaptation of the structure of Section 3.3 for maintaining the maximal points to the I/O model yields an I/O-efficient dynamic data structure that supports updates $O(\log_B n)$ worst case I/Os, dominance maxima queries in $O(\log_B n + t)$ worst case I/Os and all maxima queries in $O(m)$ worst case I/Os, using $O(n/B)$ blocks. The query operations make random I/Os, since they access the output via representative points. Can all maxima queries be supported in $O(m/B)$ I/Os, while supporting updates in $O(\log_B n)$ worst case I/Os?

- Is the space of our dynamic data structure for 4-sided range maxima reporting queries optimal? Our structure in Section 3.4 shows that it is difficult to report the maximal points among the points dominated by upper right corner of the query rectangle using linear space. These queries are also known as *dominated maxima queries* [ABGP07]. Can we prove an $\Omega(n \log n)$ space lower bound for a static data structure that supports such queries in $O(\text{polylog}\, n)$ worst case time?

- All dynamic data structures for the pointer machine that maintain maximal points [WL85,Kap00] or support rectangular visibility queries [OW88] on points of dimension $d \geq 3$, are obtained by augmenting $BB[\alpha]$-trees accordingly. This imposes a logarithmic overhead per dimension over the query, update and space complexity. Even for $d = 3$ and even for the static case, does there exist data structures that support reporting orthogonal range maxima queries in $O(\text{polylog}\, n + t)$ time, where t is the size of the output?

- Can our technique for augmenting search trees using partially persistent secondary structures be used to improve the deletion time for other dynamic problems, besides dynamic range maxima reporting problem?

Chapter 4

Fully Persistent B-Trees

Abstract

We present I/O-efficient fully persistent B-Trees that support range searches at any version in $O(\log_B n + t/B)$ I/Os and updates at any version in $O(\log_B n + \log_2 B)$ amortized I/Os, using space $O(m/B)$ disk blocks. By n we denote the number of elements in the accessed version, by m the total number of updates, by t the size of the query's output, and by B the block size. The result improves the previous fully persistent B-Trees of Lanka and Mays by a factor of $O(\log_B m)$ for the range query complexity and $O(\log_B n)$ for the update complexity. To achieve the result, we first present a new B-Tree implementation that supports searches and updates in $O(\log_B n)$ I/Os, using $O(n/B)$ blocks of space. Moreover, every update makes in the worst case a constant number of modifications to the data structure. We make these B-Trees fully persistent using an I/O-efficient method for full persistence that is inspired by the *node-splitting* method of Driscoll et al. The method we present is interesting in its own right and can be applied to any external memory pointer based data structure with maximum in-degree d_{in} and out-degree bounded by $O(B)$, where every node occupies a constant number of blocks on disk. The I/O-overhead per modification to the ephemeral structure is $O(d_{in} \log_2 B)$ amortized I/Os, and the space overhead is $O(d_{in}/B)$ amortized blocks. Access to a block of the ephemeral data structure is supported in $O(1)$ worst case I/Os.

4.1 Introduction

In this Chapter we consider the problem of making B-Trees fully persistent in the I/O model. In the pointer machine model, a direct application of the node-splitting method [DSST89] to B-Trees with constant degree is efficient since the in-degree of every node is one. However, applying this method directly to the I/O model will not yield an I/O-efficient fully persistent data structure. The persistent node of Driscoll et al. has constant size and thus corresponds to at most a constant number of updated elements of the ephemeral structure. However, a persistent node of size $\omega(1)$, say $\Theta(B)$, may correspond to $\Theta(B)$ versions of an ephemeral node. In order to find the appropriate version during navigation in the persistent node, as many versions must be compared in the version list. Since the versions are too many to fit in internal memory, the order maintenance structure is stored in external memory. Thus, one I/O is needed per version comparison, which necessitates $O(B)$ I/Os to handle one node in the worst case. By simple modifications an $O(\log_2 B)$ I/O-overhead per update and access step can be achieved.

In Section 4.2 we present a method that makes pointer-based ephemeral structures fully persistent in the I/O model. In particular, we present an interface of basic operations to be called by the implementation of the ephemeral structure. We assume that a record of the ephemeral structure occupies a constant number of blocks. The method achieves $O(1)$ worst case I/O-overhead per access step, and $O(d_{in} \log B)$ amortized I/O-overhead and $O(\frac{d_{in}}{B})$ amortized space-overhead per update step, where d_{in} is the maximum indegree of any ephemeral record.

We follow a similar approach to the node-splitting method. We further modify this method such that whenever a persistent node is accessed by a pointer traversal, the contents of the node for a particular version can be retrieved by at most a predefined number of version comparisons. In this way we manage to optimize the I/O-cost of an access step. However, maintaining this property after a sequence of update operations requires comparing a particular version against all the versions that are stored in a node. Moreover, a packed memory is used in order to manage the persistent nodes that have small size.

Applying our method to B-trees even when they need $O(1)$ amortized I/Os to rebalance, yields fully persistent B-trees that support range queries at any version in $O(\log_B n + \frac{t}{B})$ worst case I/Os and updates at any version in $O(\log_B n \cdot \log B)$ amortized I/Os, using $O(\frac{m}{B} \log_B n)$ space. By definition of full persistence, an update operation that causes $O(\log_B n)$ nodes to be rebalanced in the worst case can be repeated arbitrarily many times. The update and the space overhead comes from the fact that the necessary rebalancing operations can be divided into $O(\log_B n)$ update steps, where each makes a constant number of modifications to the tree.

In Section 4.3 we present the Incremental B-trees, an implementation of (a, b)-trees that has the same complexities as regular B-trees, and where moreover each update operation performs in the worst case $O(1)$ modifications to the tree. In regular search trees some update operations may unbalanced more that a constant number of nodes, over a worst case sequence of update operations. Balance is restored only when all the unbalanced nodes are rebalanced. In our implementation, the necessary rebalancing operations are executed incrementally over a sequence of operations, and they spend at most $O(1)$ worst case time per update operation.

Inclined AVL trees [Tsa86] is an implementation of AVL-trees, where a constant number of rebalancings per update operation that involves the nodes of a fixed root-to-leaf path is are executed. Extra information is stored on the nodes of this path, in order to decide which rotations of the nodes are necessary to be executed. This information is also updated in $O(1)$ time. Driscoll et al. [DSST89] present an implementation of red-black trees where at most a constant number of nodes are rebalanced per update operation. In particular, it is an implementation of BB-trees that also does *lazy recoloring*, namely it performs $O(1)$ recolorings per update operation. They observe that the nodes to be recolored after an update operation, form a consecutive path. They define *incremental paths* that are maintained in a similar way as [Tsa86]. Moreover, the incremental paths are maintained node disjoint, spending $O(1)$ worst case time per every update operation that affects them. Our Incremental B-trees can be seen as a generalization of lazy recoloring to (a, b)-trees.

In Section 4.4 we present fully persistent B-trees that support range queries at any version in $O(\log_B n + t/B)$ I/Os and updates at any version in $O(\log_B n + \log_2 B)$ amortized I/Os, using $O(m/B)$ disk blocks. The result is obtained by applying our I/O-efficient method for full persistence (Section 4.2) to Incremental B-trees (Section 4.3).

4.2 I/O-Efficient Method for Full Persistence

In this section we present a generic method that makes a pointer based ephemeral data structure fully persistent in the I/O model, provided that every node of the underlying graph occupies at most a constant number of disk blocks. The overhead for accessing an ephemeral node is $O(1)$ I/Os in the worst case. The overhead for updating a field in an ephemeral node is $O(d_{in} \log_2 B)$ amortized I/Os and the space overhead is $O(d_{in}/B)$ amortized blocks, where d_{in} denotes the maximum in-degree of any ephemeral node.

In particular, the ephemeral data structure D is represented by a graph where every *ephemeral node* u contains at most $c_f B$ *fields* for some constant c_f. Each field stores either an *element*, or a *pointer* to another ephemeral node. One ephemeral *entry node* provides access to the graph. An ephemeral node is *empty* if none of its fields contains elements or pointers.

The following interface provides the necessary operations to navigate and to update any version of the fully persistent data structure $\bar{D}$. The interface assumes that the user has only knowledge of the ephemeral structure. The i-th version of $\bar{D}$ is an ephemeral data structure D_i where all nodes, elements and pointers are associated with version i. A `field` is an identifier of a field of a node of D_i. The `value` of the `field` is either the element in the field at version i, or a `pointer` p_i to another node of D_i. Since the `pointer` resides in and points to nodes of the same version, we associate this version with the `pointer`. The `version` i is a unique identifier of D_i.

`pointer` p_i `= Access(version` i`)` returns a `pointer` p_i to the entry node at version i.

`value` x `= Read(pointer` p_i`, field` f`)` returns the `value` x of the `field` f in the node at version i pointed by `pointer` p_i. If x is a `pointer`, it points to a node at the same version i.

`Write(pointer` p_i`, field` f`, value` x`)` writes the `value` x in the `field` f of the node at version i pointed by p_i.

`pointer` p_i `= NewNode(version` i`)` creates a new empty node at version i and returns a `pointer` p_i to it.

`version` j `= NewVersion(version` i`)` creates an new version D_j that is a copy of the current version D_i and returns a new version identifier for D_j.

By definition of full persistence, the outcome of updating D_i is a new ephemeral structure D_j, where the new version $j \neq i$ becomes a leaf of the version tree. The above interface allows the outcome of updating D_i to be D_i itself. In other words, it provides the extra capability of updating an internal node of the version tree[1]. In order to abide by the definition, the user has to explicitly create a new version D_j before every update operation.

The Structure

Our method is inspired by the node-splitting method [DSST89] to which we make non-trivial modifications, such that whenever a node of the structure is accessed by a pointer traversal, the contents of the node for a particular version can be retrieved by at most a predefined number of version comparisons. Moreover, a packed memory is used in order to manage the persistent nodes that have small size.

As defined by full persistence, all the versions of the ephemeral data structure can be represented by a directed rooted *version tree* T. If version i is obtained by modifying version j, version j is the parent of i in T. Similarly to [DSST89] we store the preorder layout of T in a dynamic list that supports *order maintenance* queries [Die82, Tsa84, DS87, BCD+02], called the **global version list (GVL)**. Given two versions i and j, an order maintenance query returns true if i lies before j in the list, and it returns false otherwise. To preserve the preorder layout of T whenever a new version is created, it is inserted in the GVL immediately to the right of its parent version. In this way, the descendants of every version occur consecutively in the GVL. By implementing the GVL as in [DS87], order maintenance queries are supported in $O(1)$ worst case I/Os. The insertion of a version is supported in $O(1)$ worst case I/Os, given a pointer to its parent version.

We record all the changes that occur to an ephemeral node u in a linked list of *persistent nodes* $\bar{u}$, called the **family** $\phi(u)$. To implement the linked list, the persistent node $\bar{u}$ contains a pointer $c(\bar{u})$ to the next persistent node in $\phi(u)$. For every field f of the corresponding ephemeral node u, the persistent

[1]This does not immediately yield a retroactive data structure [DIL04], since all the versions in the subtree of the updated node have to be modified appropriately as well.

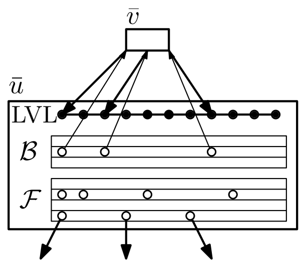

Figure 4.1: The persistent node $\bar{u}$. The versions stored in the local version list of $\bar{u}$ are represented by black dots. The values stored in $\mathcal{F}(\bar{u})$ and $\mathcal{B}(\bar{u})$ are represented with white dots. The values lie on the column that corresponds to the version they are associated with. When the field in $\mathcal{F}(\bar{u})$ contains pointers, the values contain forward pointers that are represented with thick arrows. The values in $\mathcal{B}(\bar{u})$ contain backward pointers that are represented with thin arrows. They point to the persistent node $\bar{v}$ that contains the corresponding forward pointers. Forward pointers point to their associated version in the local version list of $\bar{u}$.

node $\bar{u}$ stores a set $\mathcal{F}_f(\bar{u})$. If field f stores elements, then $\mathcal{F}_f(\bar{u})$ contains pairs (version i, value x) where x is the element stored in f at version i. Else, if field f stores pointers, then $\mathcal{F}_f(\bar{u})$ contains pairs (version i, pointer $\overrightarrow{p}$) where the *forward pointer* $\overrightarrow{p}$ corresponds to the ephemeral pointer p that is stored in f at version i. If p points to the node v at version i, then $\overrightarrow{p}$ points to the persistent node $\bar{v}$ in $\phi(v)$ that corresponds to node v at version i. For every persistent node $\bar{v}$ that contains forward pointers pointing to $\bar{u}$, the persistent node $\bar{u}$ stores a set $\mathcal{B}_{\overleftarrow{p}}(\bar{u})$ of pairs (version i, pointer $\overleftarrow{p}$) where the *backward pointer* $\overleftarrow{p}$ points to $\bar{v}$. Backward pointers do not correspond to ephemeral pointers and they are only used by the persistent mechanism to accommodate updates. The pairs in the sets $\mathcal{F}_f(\bar{u})$ and $\mathcal{B}_{\overleftarrow{p}}(\bar{u})$ are sorted with respect to the order of their first component (version i) in the GVL. We denote $\mathcal{F}(\bar{u}) = \cup_{f\in\bar{u}}\mathcal{F}_f(\bar{u})$ and $\mathcal{B}(\bar{u}) = \cup_{\overleftarrow{p}\in\bar{u}}\mathcal{B}_{\overleftarrow{p}}(\bar{u})$. Finally, the persistent node $\bar{u}$ contains a *local version list* LVL($\bar{u}$) that stores all the versions i in the pairs of $\mathcal{F}(\bar{u})$ and $\mathcal{B}(\bar{u})$, sorted with respect to their order in the GVL. The first version in the LVL($\bar{u}$) comprises the *version $i_{\bar{u}}$ of the persistent node $\bar{u}$.*

To provide access to the structure we maintain an **access array** whose i-th position stores a forward pointer $\overrightarrow{p}$ to the persistent node that corresponds to the entry node at version i, and a pointer to version i in the GVL. For ease of description we consider that the access array contains the pairs (version i, pointer $\overrightarrow{p}$). We define the *size* of a persistent node $\bar{u}$ to be the number of pairs in $\mathcal{F}(\bar{u})$ and $\mathcal{B}(\bar{u})$. This dominates the number of versions in the LVL($\bar{u}$), since there exists at least one pair per version. We call a persistent node *small* if its size is at most $\frac{c_f}{2}B$. To utilize space efficiently, we pack all families of small size in an I/O-efficient **auxiliary linked list**.

Invariant 4.1 *In every set $\mathcal{F}_f(\bar{u})$ and $\mathcal{B}(\bar{u})$ there exist a pair with version $i_{\bar{u}}$, the version of the persistent node $\bar{u}$.*

Invariant 4.1 ensures that an ephemeral node u can be retrieved by accessing exactly one persistent node $\bar{u}$ in the family $\phi(u)$.

Invariant 4.2 *The size of a persistent node $\bar{u}$ that is not stored in the auxiliary linked list is $\frac{c_f}{2}B \leq |\bar{u}| \leq c_{max}B$ for $c_{max} = \Omega(c_f(d_{in} + \frac{d_{in}^2}{B}))$.*

Invariant 4.2 ensures that a persistent node $\bar{u}$ occupies at most a constant number of blocks.

Invariant 4.3 *For every forward pointer $\overrightarrow{p}$ that points to the persistent node $\bar{v}$ and resides in a pair $(i, \overrightarrow{p})$ of $\mathcal{F}_f(\bar{u})$ or in a pair of the access array, there exists a pair $(i, \overleftarrow{p})$ in $\mathcal{B}_{\overleftarrow{p}}(\bar{v})$ where the backward pointer $\overleftarrow{p}$ points to the persistent node $\bar{u}$ or to the i-th position of the access array, respectively.*

Invariant 4.3 associates a forward pointer $\overrightarrow{p}$ with a corresponding backward pointer $\overleftarrow{p}$. It moreover ensures that the version i of the forward pointer $\overrightarrow{p}$ belongs to the LVL of the pointed persistent node $\bar{v}$. We set all forward pointers to particularly point to the version i in the LVL$(\bar{v})$. It suffices for backward pointer $\overleftarrow{p}$ to only point to the persistent node $\bar{u}$.

We define the *valid interval* of a pair (i, x) in $\mathcal{F}_f(\bar{u})$ to be the set of versions in the GVL for which field f has the particular value x. In particular, it is the interval of versions in the GVL from version i up to but not including version j. Version j is the version in the next pair of $\mathcal{F}_f(\bar{u})$, if this pair exists. Otherwise, j is the version in the first pair of $\mathcal{F}_f(c(\bar{u}))$, if $c(\bar{u})$ exists. Otherwise, the valid interval is up to the last version in the GVL. The valid interval of a pair $(i, \overrightarrow{p})$ in $\mathcal{F}_f(\bar{u})$, where $\overrightarrow{p}$ is a forward pointer to the persistent node $\bar{v}$, is identical to the valid interval of the pair $(i, \overleftarrow{p})$ in $\mathcal{B}_{\overleftarrow{p}}(\bar{v})$, where the backward pointer $\overleftarrow{p}$ corresponds to $\overrightarrow{p}$. A pair $(i, \overrightarrow{p})$ where the forward pointer $\overrightarrow{p}$ points to the persistent node $\bar{v}$, implements the `pointers` p_j that point to v at every version j that belongs to the valid interval of the pair. All versions occur in the access array, since `NewVersion` is called before every update. Thus, the valid interval of a pair $(i, \overrightarrow{p})$ in the access array is only version i. We define the *valid interval of a persistent node* $\bar{u}$ to be the union of the valid intervals of the pairs in $\mathcal{F}(\bar{u})$ and $\mathcal{B}(\bar{u})$. In particular, it is the interval of versions in the GVL from version $i_{\bar{u}}$ up to but not including version $i_{c(\bar{u})}$, if $c(\bar{u})$ exists. Otherwise, it is up to the last version in the GVL.

We define the *span of a forward pointer* $\overrightarrow{p}$ that points to the persistent node $\bar{v}$ to be the versions in the intersection of the valid interval of the pair that contains $\overrightarrow{p}$ with the LVL$(\bar{v})$. The backward pointer $\overleftarrow{p}$ that corresponds to $\overrightarrow{p}$ has the same span with $\overrightarrow{p}$.

Invariant 4.4 *The size of the span of every forward pointer is $d \in \mathbb{N}$ where $1 \leq d \leq 2\pi$ for $\pi \geq d_{in} + 9$.*

Invariant 4.4 ensures that whenever a persistent node is accessed by traversing a forward pointer, the contents of the persistent node for a particular version can be retrieved by comparing against a set of at most d versions.

Algorithms

Here we present the implementation of the user-interface. Operations `Write`, `NewNode`, and `NewVersion` immediately restore Invariants 4.1 and 4.3. This may cause at most d_{in} forward pointers to violate Invariant 4.4 and some persistent nodes to violate Invariant 4.2. The auxiliary subroutine `Repair()` restores those invariants utilizing an auxiliary *violation queue.*

We say that a version *precedes* version i in the local version list, if it is the rightmost version of the list that is not to the right of version i in the global version list. Note that version i precedes itself when it belongs to the set. We denote by i^+ the version immediately to the right of version i in the GVL.

pointer p_i **= Access(version** i**)** We return the forward pointer in i-th position of the access array, since it points to the entry node at version i.

value x **= Read(pointer** p_i**, field** f**)** Let `pointer` p_i point to the ephemeral node u at version i. Let $\bar{u}$ be the persistent node in $\phi(u)$ whose valid interval contains version i. To return the value x that field f has in the ephemeral node u at version i, we determine the pair in $\mathcal{F}_f(\bar{u})$ whose valid interval contains version i.

The pairs in $\mathcal{F}(\bar{u})$ whose valid interval contain version i, also contain the version j that precedes version i in the LVL($\bar{u}$). We determine j by searching in the LVL($\bar{u}$) as following. Let the pair $(i', \overrightarrow{p})$ contain the forward pointer that implements `pointer` p_i. By Invariant 4.3 version i' belongs to the LVL($\bar{u}$). Since version i belongs to the valid interval of this pair, version i' lies to the left of version i in the GVL. If $i' \neq j$, then version j lies to the right of version i' in the LVL($\bar{u}$). Version j belongs to the span of $\overrightarrow{p}$.

We perform a binary search on the version of the span of $\overrightarrow{p}$ in the LVL($\bar{u}$). Every comparison is implemented by an order maintenance query between the accessed version in the span and version i. In this way, we locate the rightmost version j in the span for which the order maintenance query returns true. At least one order maintenance query returns true, since version i' lies to the left of version i in the GVL. We find the pair of $\mathcal{F}_f(\bar{u})$ with the version that precedes version j in the LVL($\bar{u}$), and return the value it contains.

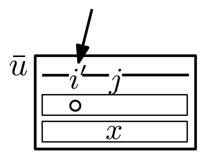

Figure 4.2: Operation `Read(`p_i`,`f`)`. The thick arrow represents the forward pointer $\overrightarrow{p}$ that implements `pointer` p_i. $\mathcal{B}(\bar{u})$ and $\mathcal{F}_f(\bar{u})$ are represented by thin rectangles. The white dot represents the backward pointer that corresponds to $\overrightarrow{p}$. We assume that version i does not belong to the LVL($\bar{u}$).

Write(pointer p_i, field f, value x) Let `pointer` p_i point to the ephemeral node u. Let $\bar{u}$ be the persistent node in $\phi(u)$ whose valid interval contains version i. As in `Read`, we find the version j that precedes version i in the LVL($\bar{u}$), and the pair (j', y) in $\mathcal{F}_f(\bar{u})$ whose valid interval contains version i.

If $j' = i$, we merely replace y with x. In this case, version i is the currently updated version and it belongs to the LVL($\bar{u}$). Otherwise, we add the pairs (i, x) and (i^+, y) to $\mathcal{F}_f(\bar{u})$. By this way version i belongs only to the valid interval of the pair (i, x). Moreover, the versions that belonged to the valid interval of the pair (j', y) and succeed version i in the GVL, continue having the previous value y. If there is already a pair in $\mathcal{F}_f(\bar{u})$ with version i^+, it suffices to only add the pair (i, x). If $\mathcal{F}_f(\bar{u})$ is empty, we add the pairs $(i_{\bar{u}}, null)$, (i, x) and $(i^+, null)$ instead, where $i_{\bar{u}}$ is the version of the persistent node $\bar{u}$.

Version i is inserted to the LVL($\bar{u}$) immediately to the right of version j. Unless version i^+ already exists in the LVL($\bar{u}$), it is inserted immediately to the right of version i. These insertions may cause at most d_{in} forward pointers $\overrightarrow{P_{\bar{u}}}$ that point to $\bar{u}$ to violate Invariant 4.4. The persistent nodes that contain them have to be inserted to the violation queue. To find the forward pointers $\overrightarrow{P_{\bar{u}}}$, we determine the corresponding backward pointers in $\bar{u}$. In particular, we find all the pairs $(k, \overleftarrow{p})$ in $\mathcal{B}(\bar{u})$ whose valid intervals contain the inserted versions, and check if there are more than 2π versions in the LVL($\bar{u}$) between version k and the version of the next pair in $\mathcal{B}_{\overleftarrow{p}}(\bar{u})$. If so, we access the persistent node $\bar{z}$ pointed by $\overleftarrow{p}$ and mark the pair in $\mathcal{F}_f(\bar{z})$ with the corresponding forward pointer. We insert $\bar{z}$ to the violation queue, unless it has already been inserted.

If x is a `pointer` to an ephemeral node v at version i, the argument `pointer` x_i is implemented by a forward pointer $\overrightarrow{x}$ to the persistent node $\bar{v}$ in $\phi(v)$ whose valid interval contains version i. Version i belongs to the span of $\overrightarrow{x}$. We add to $\mathcal{F}_f(\bar{u})$ the pairs $(i, \overrightarrow{x})$ and $(i^+, \overrightarrow{y}')$ instead, where $\overrightarrow{y}'$ is a forward pointer to the persistent node $\bar{w}$ pointed by the forward pointer $\overrightarrow{y}$ of the pair $(j', \overrightarrow{y})$ in $\mathcal{F}_f(\bar{u})$. We restore Invariant 4.3 for the added pair $(i, \overrightarrow{x})$ by inserting the corresponding backward pointer $\overleftarrow{x}$ to $\mathcal{B}(\bar{v})$. In particular, we add the pair $(i, \overleftarrow{x})$ to $\mathcal{B}_{\overleftarrow{x}}(\bar{v})$ where the backward pointer $\overleftarrow{x}$ points to the persistent node $\bar{u}$. We perform a binary search in the span of $\overrightarrow{x}$ in order to find the version in the LVL($\bar{v}$) that precedes version i. Unless it is i itself, we insert version i immediately to the right of it. The at most d_{in} forward pointers $\overrightarrow{P_{\bar{v}}}$ that violate Invariant 4.4 are processed as described above. We set $\overrightarrow{x}$ to point to version i in the LVL($\bar{v}$). The added pair implements `pointer` x_i. If $(i^+, \overrightarrow{y}')$ was also added, we restore Invariant 4.3 for $\overrightarrow{y}'$ as described above. Version i^+ belongs to the span of $\overrightarrow{y}$. The at most d_{in} forward pointers $\overrightarrow{P_{\bar{w}}}$ may violate Invariant 4.4. Notice that $\bar{u}$ may contain a pointer from $\overrightarrow{P_{\bar{v}}}$ or $\overrightarrow{P_{\bar{w}}}$.

The insertion of pairs in the persistent nodes $\bar{u}$, $\bar{v}$ and $\bar{w}$ increases their size. If the nodes are not small anymore due to the insertion, we remove them from the auxiliary linked list. If they violate Invariant 4.2, we insert them to the violation queue, unless they have already been inserted. Finally we call `Repair`.

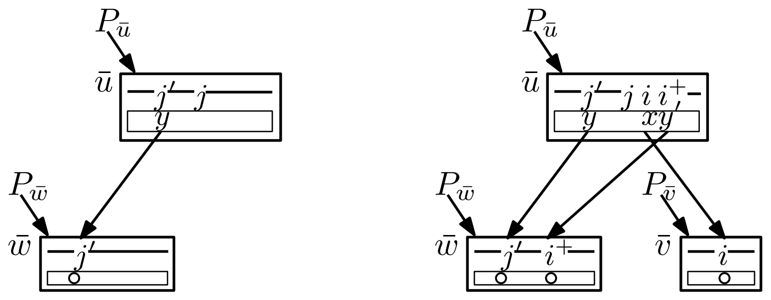

Figure 4.3: Operation `Write` inserts to the ephemeral node u at version i a pointer x that points to the ephemeral node v at version i. The figures show how the corresponding persistent nodes $\bar{u}$, $\bar{v}$ and $\bar{w}$ look before and after the operation. The persistent node $\bar{w}$ is the node that is pointed by the forward pointer of the pair in $\mathcal{F}_f(\bar{u})$ whose valid interval contains version i, where f is the field in $\bar{u}$ that contains the inserted pointer. The local version lists are represented by horizontal lines.

pointer p_i **= NewNode(version** i**)** We create a new family $\phi(u)$ which consists of one empty persistent node $\bar{u}$. We insert version i to the LVL($\bar{u}$), so that $\bar{u}$ satisfies Invariant 4.1. Node $\bar{u}$ is added to the auxiliary linked list since it is small. We return a forward pointer to version i in the LVL($\bar{u}$).

version j **= NewVersion(version** i**)** We traverse the pointer stored at the i-th position of the access array to find the position of version i in the GVL. We insert version j immediately to the right of version i in the GVL. We insert to the j-th position of the access array a pointer to version j in the GVL. Let $\bar{u}$ be the persistent node pointed by the forward pointer in the i-th position of the access array. We insert in the j-th position of the access array a forward pointer to $\bar{u}$. At most d_{in} forward pointers $\overrightarrow{P_{\bar{u}}}$ may violate Invariant 4.4. If $\bar{u}$ violates Invariant 4.2 we insert it to the violation queue, unless it has been already inserted. Finally `Repair` is called.

Repair() iteratively pops a persistent node $\bar{u}$ from the violation queue, and restores Invariant 4.4 for the forward pointers in the marked pairs of $\mathcal{F}_f(\bar{u})$ and Invariant 4.2 for $\bar{u}$. These invariants may in turn be violated in other persistent nodes, which we insert in the violation queue as well. This iteration terminates when the queue becomes empty.

To restore Invariant 4.4 for the forward pointer in the marked pair $(i, \overrightarrow{p})$ in $\mathcal{F}_f(\bar{u})$, we reset the size of its span to π as following. Let $\overrightarrow{p}$ point to the persistent node $\bar{v}$. We find the version j in the span of $\overrightarrow{p}$ that resides π positions to the right of version i in the LVL($\bar{v}$). We set the forward pointer $\overrightarrow{p}'$ to version j in the LVL($\bar{v}$), and add the pair $(j, \overrightarrow{p}')$ to $\mathcal{F}_f(\bar{u})$. If the span of $\overrightarrow{p}'$ violates Invariant 4.4, we mark its pair. We restore Invariant 4.3 for the added pair as described in `Write`. Node $\bar{v}$ may violate Invariant 4.2. We find the version that precedes version j in the LVL($\bar{u}$), by a binary search over the whole LVL($\bar{u}$). We insert j immediately to the right of its preceding version, unless it already exists. This may cause at most d_{in} forward pointers $\overrightarrow{P_{\bar{u}}}$ to violate Invariant 4.4. Node $\bar{u}$ may violate Invariant 4.2.

To restore Invariant 4.2 for the persistent node $\bar{u}$, we split it into two persistent nodes, such that the right one has size approximately $\frac{c_{max}}{2}B$. We first determine the version j at which we will split $\bar{u}$, by scanning LVL($\bar{u}$) from right to left. Version j is the leftmost version in the LVL($\bar{u}$), such that the number of pairs whose version succeeds j is less than $\frac{c_{max}}{2}B$. Unless $j' = j$, for every pair (j', x) in $\bar{u}$ whose valid interval contains version j, we add a pair (j, x) in $\bar{u}$. If x is a forward pointer to a persistent node $\bar{v}$, we restore Invariant 4.3 as described in `Write`. If x is a backward pointer to $\bar{v}$, restoring Invariant 4.3 involves a binary search for the version that precedes j' in the LVL($\bar{v}$). Node $\bar{v}$ may violate Invariant 4.2. Moreover, at most d_{in} forward pointers $\overrightarrow{P_{\bar{v}}}$ may violate Invariant 4.4. We create a new persistent node $\bar{u}'$ that succeeds $\bar{u}$ in the family $\phi(u)$, by setting $c(\bar{u}') = c(\bar{u})$ and $c(\bar{u}) = \bar{u}'$. We split the LVL($\bar{u}$) at version j. The right part becomes LVL($\bar{u}'$). Version j becomes the version of $\bar{u}'$. All the pairs in $\bar{u}$ with a version in LVL($\bar{u}'$) are moved to $\bar{u}'$. We traverse all forward and backward pointers in $\mathcal{F}(\bar{u}')$ and $\mathcal{B}(\bar{u}')$ in order to set the corresponding backward and forward pointers to point to $\bar{u}'$, respectively. Version j becomes the version of $\bar{u}'$. The node $\bar{u}'$ satisfies Invariant 4.1 due to the addition of the pairs (j, x).

Analysis

The following remarks are necessary for the analysis. A version in the LVL($\bar{u}$) belongs to the span of at most d_{in} forward pointers that point to $\bar{u}$, and thus it belongs to the valid interval of at most d_{in} pairs in $\mathcal{B}(\bar{u})$.

Lemma 4.1 *After splitting a persistent node $\bar{u}$ the size of $\bar{u}'$ is within the range* $[(\frac{c_{max}}{2} - c_f)B, \left(\frac{c_{max}}{2} + c_f\right)B + d_{in} - 1]$.

Proof. The number of pairs with version j and with versions that succeed version j in LVL($\bar{u}$) before the split is at least $(\frac{c_{max}}{2} - c_f)B$. This sets the lower bound. The number of pairs with a version that succeeds j in the LVL($\bar{u}$) is at most $\frac{c_{max}}{2}B - 1$. There are at most $c_f B$ pairs with a single version in $\mathcal{F}(\bar{u})$, and at most d_{in} pairs in $\mathcal{B}(\bar{u})$ whose valid interval contains version j. We add one pair for each of them to $\bar{u}'$. This sets the upper bound. $\square$

First we analyze the worst case cost of every operation. `Access` performs one I/O to the access array. `Read` performs at most c_{max} I/Os to load the persistent node $\bar{u}$ into memory, and at most $\log_2 2\pi$ I/Os for the order maintenance queries in order to determine the appropriate version j. `Write` performs at most c_{max} I/Os to load $\bar{u}$ as well as at most $\log_2 2\pi$ I/Os to locate the proper version as in `Read`. One I/O is needed to access version i^+ in the GVL, and at most $d_{in}c_{max}$ I/Os are needed to access the forward pointers of $\overrightarrow{P_{\bar{u}}}$. If the written value is a pointer, then we also need at most $2\log_2 2\pi$ I/Os to process $\bar{v}$ and $\bar{w}$, and $2d_{in}c_{max}$ I/Os to access the forward pointers of $\overrightarrow{P_{\bar{v}}}$ and $\overrightarrow{P_{\bar{w}}}$. In total, without taking into account the call to `Repair`, the worst case cost is $3d_{in}c_{max} + 3\log_2 2\pi + c_{max} + 1$ I/Os. `NewNode` makes at most c_{max} I/Os to access the entry node. `NewVersion` spends one I/O to update the GVL, and at most c_{max} I/O to process $\bar{u}$ and to insert $\bar{u}$ to the violation queue. To restore Invariant 4.4 for one forward pointer `Repair` makes at most $2c_{max}$ I/Os to load $\bar{u}$ and $\bar{v}$ into memory, at most $\log_2 c_{max}B$ I/Os to insert version j to the LVL($\bar{v}$), and at most $d_{in}c_{max}$ I/Os to insert the nodes with $\overrightarrow{P_{\bar{v}}}$ to the violation queue. In total the worst case cost is $c_{max}(d_{in}+2)+\log_2 c_{max}B$ I/Os. To restore Invariant 4.2 for a persistent node, `Repair` makes at most c_{max} I/Os to load the node into memory, at most $c_f B(c_{max}+\log_2 2\pi)$ I/Os to restore Invariant 4.3 for the added pairs in $\bar{u}$ with forward pointers, at most $d_{in}(c_{max}+\log_2 c_{max}B)$ I/Os to restore Invariant 4.3 for the added pairs in $\bar{u}$ with backward pointers, at most $(c_f B + d_{in})d_{in}c_{max}$ I/Os to insert the nodes with $\overrightarrow{P_{\bar{v}}}$ to the violation queue, and at most $((\frac{c_{max}}{2})B-1)c_{max}$ I/Os to set the forward and backward pointers to point to $\bar{u}'$. In total the worst case cost is $B(\frac{c_{max}^2}{2}+c_f(c_{max}(d_{in}+1)+\log_2 2\pi))+d_{in}(c_{max}(d_{in}+1)+\log_2 c_{max}B)$ I/Os.

Let D_i be the persistent structure after the i-th operation. We define the potential of D_i to be $\Phi(D_i)=\sum_{\overrightarrow{p}\in\mathcal{P}}\Xi(\overrightarrow{p})+\sum_{\bar{u}\in\overline{\mathcal{U}}}\Psi(\bar{u})$, where $\mathcal{P}$ is the set of all forward pointers and $\overline{\mathcal{U}}$ is the set of all persistent nodes in D_i. The function $\Xi(\overrightarrow{p})=\max\{0,|\overrightarrow{p}|-\pi\}$ provides the potential to the forward pointer $\overrightarrow{p}$ for the splitting of its span. By $|\overrightarrow{p}|$ we denote the size of the span of $\overrightarrow{p}$. Function $\Psi(\bar{u})=\max\left\{0,3\left(|\bar{u}|-\left(\left(\frac{c_{max}}{2}+c_f\right)B+d_{in}\right)\right)\right\}$ provides the potential to the persistent node $\bar{u}$ for its split. By $|\bar{u}|$ we denote the size of the persistent node $\bar{u}$.

Operation `Write`, without the call to `Repair`, increases $\Psi(\bar{u})$, $\Psi(\bar{v})$ and $\Psi(\bar{w})$ by at most 12 in total. The potential of the at most d_{in} forward pointers of $\overrightarrow{P_{\bar{u}}}$ is increased by at most 2. The potential of the at most $2d_{in}$ forward pointers $\overrightarrow{P_{\bar{v}}}$ and $\overrightarrow{P_{\bar{u}}}$ is increased by at most 1. In total the potential increases by $12+4d_{in}$. Operation `NewVersion` increases $\Psi(\bar{u})$ by 3 and the potential of at most d_{in} forward pointers $\overrightarrow{P_{\bar{u}}}$ by 1. In total the potential increases by $3+d_{in}$. When operation `Repair` restores Invariant 4.4 for one marked pair it increases $\Psi(\bar{u})$ and $\Psi(\bar{v})$ by 6 in total, and the potential of the at most d_{in} forward pointers $P_{\bar{u}}$ by at most 1. The potential of the forward pointer in the pair is decreased by π. In total the potential changes by $d_{in}+6-\pi$. When operation `Repair` restores Invariant 4.2 for the persistent node $\bar{u}$, it increases $\Psi(\bar{u})$ by at most $3c_f B$ due to the added pairs with elements and forward pointers, and by at most $3d_{in}$ due to the added pairs with backward pointers. The addition of the corresponding backward and forward pointers increases the potential of at most $c_f B+d_{in}$ persistent nodes by 3. The potential of at most $d_{in}(c_f B+d_{in})$ forward pointers to these persistent nodes is increased by 1. After the split, the potential decreases by $3(\frac{c_{max}}{2}-c_f)B$ by the lower bound in Lemma 4.1. In total the potential changes by $B(c_f(9+d_{in})-\frac{3}{2}c_{max})+d_{in}(d_{in}+6)$.

Let D_i be the result of executing `Write` or `NewVersion` on D_{i-1}, followed by a `Repair` operation. We assume that a version and a pair fit in a field of a block. A modification of a field involves adding a pair or a version in a persistent node, or changing the value in a pair. The number of modifications caused by `Repair` bounds asymptotically the number of persistent nodes it accesses. The amortized number of fields modified by `Repair` is $\tilde{c}_i=c_i+\Phi(D_i)-\Phi(D_{i-1})$, where c_i is the real number of modifications in D_{i-1}. If `Repair` restores Invariant 4.4 for α forward pointers, the amortized number of modified fields is

$$\alpha\,(3 + (d_{in} + 6 - \pi))$$

This is be case we add one version and two pairs. It is non-positive for $\pi \geq d_{in} + 9$. If `Repair` restores Invariant 4.2 for β persistent nodes, the amortized number of modified fields is

$$\beta\left(3c_fB{+}3d_{in}{+}2(\frac{c_{max}}{2}{+}c_f)B{+}2d_{in}{-}2{+}B(c_f(9{+}d_{in}){-}\frac{3}{2}c_{max}){+}d_{in}(d_{in}{+}6)\right)$$

This is because we add at most c_fB pairs with forward pointers in the node and one corresponding backward pointer and version at the node pointed by each of these forward pointers. We add at most d_{in} pairs with backward pointers in the node and one corresponding forward pointer and version at the node pointed by each backward pointer. We transfer at most $(\frac{c_{max}}{2} + c_f)B + d_{in} - 1$ pairs to $\bar{u}'$ and update as many pointers. It is non-positive for $c_{max} \geq c_f(28 + 2d_{in} + 2\frac{d_{in}^2+11d_{in}-2}{B})$. The amortized number of fields modified by `Write` is $8{+}12{+}4d_{in}$. This is because we add at most 4 versions and 4 pairs. The amortized number of fields modified by `NewVersion` is $2 + 3 + d_{in}$. This is because we add at most one version and one pair. Thus, the amortized number of fields modified by `Write` and `NewVersion` is $O(d_{in})$, which implies an $O(d_{in}(c_{max} + \log_2 \pi))$ amortized I/O-cost per modification, since $O(c_{max})$ I/Os are needed to load a node in memory and $O(\log_2 \pi)$ I/Os are needed to insert a version to the span. Moreover, when we insert a version to the local version list of a persistent node that is accessed by a backward pointer, we need $\log_2 c_{max}B$ I/Os. Thus, we charge every modified field with $\log_2 c_{max}B$ I/Os. The total cost for an update step is $O(d_{in}(c_{max} + \log_2 c_{max}B) + \log_2 \pi)$ amortized I/Os. Since an update operation makes $O(d_{in})$ amortized number of fields modifications and since all small blocks are packed in the auxiliary linked list, it follows that the space usage after m operations is $\Theta(d_{in}\frac{m}{B})$. The following theorem is proved.

Theorem 4.1 *Let D be a pointer-based ephemeral data structure that supports queries in $O(q)$ worst case I/Os and where updates make $O(u)$ modifications to the structure in the worst case. Given that every node of D occupies at most $\lceil c_f \rceil$ blocks, for a constant c_f, D can be made fully persistent such that a query to a particular version is supported in $O(q(c_{max} + \log_2 \pi))$ worst case I/Os, and an update to any version is supported in $O(d_{in}(c_{max} + \log_2 c_{max}B) + \log_2 \pi)$ amortized I/Os, where d_{in} is the maximum in-degree of any node in D, $c_{max} = \Omega(c_f(d_{in} + \frac{d_{in}^2)}{B})$ is the number of blocks occupied by a node of $\bar{D}$, and $\pi \geq d_{in} + 9$ is a parameter. After performing a sequence of m updates, the fully persistent structure occupies $O(m\frac{ud_{in}}{B})$ blocks of space.*

4.3 Incremental B-Trees

In this section we design B-Trees [BM72, Com79, HM82] that use $O(n/B)$ disk blocks of space, support insertions and deletions of elements in $O(\log_B n)$ I/Os, and range queries in $O(\log_B n + t/B)$ I/Os. They are designed such that an update makes in the worst case $O(1)$ modifications to the tree. This is achieved by marking the unbalanced nodes and by incrementally performing the expensive rebalancing operations of ordinary B-Trees over the sequence of succeeding updates.

The Structure

An *Incremental B-Tree* is a rooted tree with all leaves on the same level. Each element is stored exactly once in the tree, either in a leaf or in an internal node. In the latter case it acts as a *search key*. An internal node u with k children stores a list $[p_1, e_1, p_2, \ldots, e_{k-1}, p_k]$ of $k-1$ elements $e_1, \ldots, e_{k-1}$ stored in non-decreasing order and k children pointers $p_1, \ldots, p_k$. The discussion that follows shows that $\frac{B}{2} - 1 \leq k \leq 2B+1$. If x_i is an element stored in the i-th subtree of u, then $x_1 < e_1 < x_2 < e_2 < \cdots < e_{k-1} < x_k$ holds. To handle the incremental rebalancing of the tree, each node can either be *unmarked* or it contains one of the following marks:

Overflowing mark: The node should be replaced by two nodes.

Splitting mark: The node w is being *incrementally split* by moving elements and children pointers to its unmarked right sibling w'. We say that nodes w and w' define an *incremental splitting pair.*

Fusion mark: The node w is being *incrementally fused* by moving elements and children pointers to its unmarked right sibling w'. In case w is the rightmost child of its parent, then w' is its unmarked left sibling and elements and children pointers are moved from w' to w. We say that nodes w and w' define an *incremental fusion pair.*

Unmarked nodes that do not belong to incremental pairs are called *good* nodes. We define the *size* s_u of an internal node u to be the number of good children plus twice the number of its children with an overflowing mark plus the number of its children with a fusion mark. The size of a leaf is the number of elements in it. Conceptually, the size of an internal node is the degree that the node would have, when the incremental rebalancing of its children has been completed. The advance of the incremental rebalancing is captured by the following invariants.

Invariant 4.5 *A node with an overflowing mark has size* $2B+1$.

Invariant 4.6 *An incremental splitting pair* (w, w') *with sizes* s_w *and* $s_{w'}$ *respectively satisfies* $2 \cdot |s_w + s_{w'} - 2B - 1| \leq s_{w'} < s_w$.

The left inequality of Invariant 4.6 ensures that the incremental split terminates before the resulting nodes may participate in a split or a fusion again. In particular, it ensures that the number of the transferred elements and children pointers from w to w' is at least twice the number of insertions and deletions that involve the nodes of the splitting pair since the beginning of the incremental split. This allows for the transfer of one element and one child pointer for every such insertion and deletion. The right inequality of Invariant 4.6 ensures that the incremental split terminates, since the size of w' increases and the size of w decreases for every such insertion and deletion.

Invariant 4.7 *An incremental fusion pair (w, w') with sizes s_w and $s_{w'}$ respectively, where elements and children pointers are moved from w to w', satisfies $0 < s_w \leq \frac{B}{2} + 3 - 2 \cdot |s_w + s_{w'} - B + 1|$.*

Conversely, the right inequality of Invariant 4.7 ensures that the incremental fusion terminates before the resulting node may participate in a split or a fusion again. The left inequality of Invariant 4.7 ensures that the incremental fusion terminates, since the size of w decreases for every insertion and deletion that involve the nodes of the incremental pair.

Invariant 4.8 *Except for the root, all good nodes have size within $[B/2, 2B]$. If the root is unmarked, it has at least two children and size at most $2B$.*

It follows from the invariants that the root of the tree cannot have a splitting or a fusion mark, since no sibling is defined. It can only have an overflowing mark or be unmarked. All kinds of marks can be stored in the nodes explicitly. However, we cannot afford to explicitly mark all unbalanced nodes since an update operation may unbalance more than a constant number of them. Thus, overflowing and fusion marks can also be stored implicitly, based on the observation that the unbalanced nodes occur consecutively in a path of the tree. In particular, for a $u \rightarrow v$ path in the tree, where u is an ancestor of v and all nodes in the path have overflowing marks, we can represent the marks implicitly, by marking u with an overflowing mark and additionally storing in u an element of v. The rest of the nodes in the path have no explicit mark. This defines an *overflowing path.* Similarly, we can represent paths of nodes with fusion marks, which defines a *fusion path.*

Invariant 4.9 *All overflowing and fusion paths are node-disjoint.*

Lemma 4.2 *The height of the Incremental B-Tree with n elements is $O(\log_B n)$.*

Proof. We transform the Incremental B-Tree into a tree where all incremental operations are completed and thus all nodes are unmarked. We process the marked nodes bottom-up in the tree and replace them by unmarked nodes, such that when processing a node all its children are already unmarked. A node with an overflowing mark that has size $2B + 1$ is replaced by two unmarked nodes of size B and $B + 1$ respectively. The two nodes in an incremental splitting pair (w, w') are replaced by two nodes, each containing half the union of their children. More precisely, they have sizes $\lfloor \frac{s_w+s_{w'}}{2} \rfloor$ and $\lceil \frac{s_w+s_{w'}}{2} \rceil$ respectively. By Invariant 4.6 we derive that $\frac{8}{5}B \leq s_w + s_{w'}$, i.e. each of the nodes has degree at least $B/2$. The two nodes in an incremental fusion pair (w, w') are replaced by a single node that contains the union of their children and has size $s_w + s_{w'}$. By Invariant 4.7 we derive that $\frac{3}{4}B - 1 \leq s_w + s_{w'}$. In all cases the nodes of the transformed tree have degree at least $B/2$, thus its height is $O(\log_B n)$. The height of the transformed tree is at most the height of the initial tree minus one. It may be lower than that of the initial tree, if the original root had degree two and its two children formed a fusion pair. □

Algorithms

The insertion and deletion algorithms use the *explicit mark* and the *incremental step* algorithms as subroutines. The former maintains Invariant 4.9 by transforming implicit marks into explicit marks. The latter maintains Invariants 4.6 and 4.7 by moving at most four elements and child pointers between the nodes of an incremental pair, whenever an insertion or a deletion involve these nodes.

In particular, let $u \rightarrow v$ be an implicitly defined overflowing (resp. fusion) path where u is an ancestor of v in the tree. That is, all marks are implicitly represented by marking u explicitly with an overflowing (resp. fusion) mark and storing in u an element e of v. Let w be a node on $u \rightarrow v$, and w_p, w_c be its parent and child node in the path respectively. Also, let e_p be an element in w_p. The subroutine *explicit mark* makes the mark on w explicit, by breaking the $u \rightarrow v$ path into three node-disjoint subpaths $u \rightarrow w_p$, w, and $w_c \rightarrow v$. This is done by replacing the element at u with e_p, explicitly setting an overflowing mark on w, and explicitly setting an overflowing mark together with the element e in w_c. If $u = w$ or $w = v$, then respectively the first or the third subpath is empty.

The *incremental step algorithm* is executed on a node w that belongs to a fusion or a splitting pair (w, w'), or on an overflowing node w. In the latter case, we first call the procedure *explicit mark* on w. Then, we mark it with an incremental split mark and create a new unmarked right sibling w', defining a new incremental splitting pair. The algorithm proceeds as in the former case, moving one or two children from w to w', while preserving consistency for the search algorithm. Note that the first moved child causes an element to be inserted to the parent of w, increasing its size.

In the former case, the rightmost element e_k and child p_{k+1} of w are moved from w to w'. If the special case of the fusion mark definition holds, they are moved from w' to w. Let w_p be the common parent of w and w', and let e_i be the element at w_p that separates w and w'. If p_{k+1} is part of an overflowing or a fusion path before the move, we first call *explicit mark* on it. Next, we delete e_k and p_{k+1} from w, replace e_i with e_k, and add p_{k+1} and e_i to w'. If p_{k+1} was part of a splitting or fusion pair, we repeat the above once again so that both nodes of the pair are moved to w'. We also ensure that the left node of the pair is marked with an incremental fusion mark, and that the right node is unmarked. Finally, if the algorithm causes $s_{w'} \geq s_w$ for a splitting pair (w, w'), the incremental split is complete and thus we unmark w. It is also complete if it causes $s_w = 0$ for a node w of a fusion pair. Thus, we unmark the nodes of the pair, possibly first marking them explicitly with a fusion mark and dismissing the empty node w from being a child of its parent.

Insert The insertion algorithm inserts one new element e in the tree. Like in ordinary B-Trees, it begins by searching down the tree to find the leaf v in which the element should be inserted, and inserts e in v as soon as it is found. If v is marked, we perform two incremental steps at v and we are done. If v is unmarked and has size at most $2B$ after the insertion, we are done as well. Finally, if v has size $2B+1$ it becomes overflowing. We define an overflowing path from the highest ancestor u of v, where all the nodes on the $u{\rightarrow}v$ path have size exactly $2B$, are unmarked and do not belong to an incremental pair. We do this by explicitly marking u with an overflowing mark and inserting element e in it as well. This increases the size of u_p, the parent of u. We perform two incremental steps to u_p, if it is a marked node or if it is an unmarked node that belongs to an incremental pair. Otherwise, increasing the size of u_p leaves it unmarked and we are done. Note that in order to perform the above algorithms, the initial search has to record node u_p, the topmost ancestor and the bottommost node of the last accessed implicitly marked path, and the last accessed explicitly marked node.

Delete The deletion algorithm removes an element e from the tree. Like in ordinary B-Trees, it begins by searching down the tree to find node z that contains e, while recording the topmost and the bottommost node of the last accessed implicitly marked path and the last accessed explicitly marked node. If z belongs to an overflowing or a fusion path, it explicitly marks it. If z is not a leaf, we then find the leaf v that stores the successor element e' of e. Next, we swap e and e' in order to guarantee that a deletion always takes place at a leaf of the tree. If v belongs to an overflowing or a fusion path, we mark it explicitly as well. The explicit markings are done in order to ensure that e and e' are not stored as implicit marks in ancestors of z or v.

We then delete e from v. If v is good and has size at least $B/2$ after the deletion, then we are done. If v is overflowing or belongs to an incremental pair, we perform two incremental steps on v and we are done. Otherwise, if leaf v is unmarked and has size $B/2-1$ after the deletion, we check its right sibling v'. If v' is overflowing or belongs to an incremental pair, we perform two incremental steps on v', move the leftmost child of v' to v and we are done. Only the move of the leftmost child suffices when v' is good and has degree more than $B/2+1$. Finally, if v' is good and has size at most $B/2+1$, we begin a search from the root towards v in order to identify all its consecutive unmarked ancestors u of size $B/2$ that have a good right sibling u' of size at most $B/2+1$. We act symmetrically for the special case of the fusion pair.

Let u_p be the node where the search ends and u be its child that was last accessed by this search towards v. We implicitly mark all the nodes on the $u{\rightarrow}v$ path as fusion pairs by setting a fusion mark on u and storing an element of v in u. We next check node u_p. If it is unmarked and has size greater than

$B/2$, defining the fusion path only decreases the size of u_p by one, hence we are done. If node u_p is marked, we additionally apply two incremental steps on it and we are done. If u_p is good and has size $B/2$, and its sibling u'_p is good and has size bigger than $B/2+1$, we move the leftmost child of u'_p to u_p. This restores the size of u'_p back to $B/2$ and we are done. Finally, if node u_p is good and has size $B/2$, but its sibling is marked or belongs to an incremental pair, we explicitly mark u_p and move the leftmost child of u'_p to u_p. Next, we apply two incremental steps on u'_p.

Range Search A range search is implemented as in ordinary B-Trees. It decomposes into two searches for the leaves that contain the marginal elements of the range, and a linear scan of the leaves that lie in the range interleaved with a inorder traversal of the search keys in the range.

Correctness & Analysis

We show that the update algorithms maintain all invariants. Invariant 4.5 follows from the definition of overflowing paths in the insert algorithm. The insert and delete algorithms perform two incremental steps, whenever the size of a node that belongs to an incremental pair increases or decreases by one. This suffices to move at least one element and pointer and thus to preserve Invariants 4.6 and 4.7. Invariant 4.8 is a corollary of Invariant 4.5, 4.6 and 4.7. With respect to Invariant 4.9, the insert and delete algorithms define node-disjoint incremental paths. Moreover, each incremental step ensures that two paired nodes remain children of a common parent node. Finally, the moves performed by the delete algorithm excluding incremental steps, explicitly mark the involved node preventing the overlap of two paths.

Theorem 4.2 *Incremental B-Trees can be implemented in external memory using $O(n/B)$ blocks of space and support searching, insertions and deletions of elements in $O(\log_B n)$ I/Os and range searches in $O(\log_B n + t/B)$ I/Os. Moreover, every update makes a constant number of modifications to the tree.*

Proof. By Lemma 4.2 we get that the height of the tree is $O(\log_B n)$. We now argue that the tree has $O(n/B)$ nodes, each of which has degree $O(B)$, i.e. the space bound follows and each node can be accessed in $O(1)$ I/Os.

From Invariants 4.5 - 4.8, it follows that all overflowing and the good leaves have size at least $B/2$. Also two leaves in an incremental pair have at least $B/2$ elements combined. Thus the leaves consume $O(n/B)$ disk blocks of space, which dominates the total space of the tree. The same invariants show that all nodes have at most $\frac{8B+4}{3}$ elements in them and their degree is upper bounded by $\frac{16B+8}{3}$. Thus every node consumes $O(1)$ disk blocks of space and can be accessed in $O(1)$ I/Os.

We conclude that searching, inserting and deleting an element costs $O(\log_B n)$ I/Os. A range search costs $O(\log_B n)$ I/Os for the search and $O(t/B)$ I/Os for the traversal. Finally, the rebalancing algorithms are defined such that they perform at most a constant number of modifications (incremental steps and definitions of paths) to the structure. □

4.4 Fully Persistent Incremental B-Trees

The interface can be used to make Incremental B-Trees fully persistent. The marks of every node are recorded by an additional field. Since $d_{in} = 1, c_f \leq 3$, by Theorem 4.1 we get constants $\pi \geq 10$ and $c_{max} = 96$. A range search operation on the i-th version of the fully persistent incremental B-Tree is implemented by an `Access(`i`)` operation to determine the root at version i, and a `Read` operation for every node at version i visited by the ephemeral algorithm. Since every node at version i is accessed in $O(1)$ I/Os, the range search makes $O(\log_B n + t/B)$ I/Os. An update operation on the i-th version is implemented first by a `NewVersion(`i`)` operation that creates a new version identifier j for the structure after the update operation. Then, an `Access(`j`)` operation and a sequence of `Read` operations follow in order to determine the nodes at version j to be updated. Finally, a sequence of `Write` operations follows in order to record the modifications made by the insertion and the deletion algorithms described in 4.3. By Theorem 4.1 the corollary follows.

Corollary 4.1 *There exist fully persistent B-Trees that support range searches at any version in $O(\log_B n + t/B)$ I/Os and updates at any version in $O(\log_B n + \log_2 B)$ amortized I/Os, using space $O(m/B)$ disk blocks, where n denotes the number of elements in the accessed version, m the total number of updates, t the size of the query's output, and B the disk block size.*

4.5 Open Problems

- Obtaining fully persistent B-Trees with $O(1)$ I/O- and space-overhead per access and update step remains an open problem, even when the update operations to the ephemeral B-trees makes at most a constant number of modifications to it.

- An even more difficult problem was posed by Vitter [Vit08], who asks of designing fully persistent B-Trees with $O(1)$ I/O- and space-overhead per access and update step, when the ephemeral B-tree only exhibits $O(1)$ amortized update I/Os.

Bibliography

[ABGP07] Mikhail Atallah, Marina Blanton, Michael Goodrich, and Stanislas Polu. Discrepancy-sensitive dynamic fractional cascading, dominated maxima searching, and 2-d nearest neighbors in any minkowski metric. In Frank Dehne, Jö rg-Rü diger Sack, and Norbert Zeh, editors, *Algorithms and Data Structures*, volume 4619 of *Lecture Notes in Computer Science*, pages 114–126. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-73951-711.

[ABR00] Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. New data structures for orthogonal range searching. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 198–207, Washington, DC, USA, 2000. IEEE Computer Society.

[ADT03] Lars Arge, Andrew Danner, and Sha-Mayn Teh. I/O-efficient point location using persistent B-trees. *J. Exp. Algorithmics*, 8:1.2, 2003.

[AHR98] Stephen Alstrup, Thore Husfeldt, and Theis Rauhe. Marked ancestor problems. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, FOCS '98, pages 534–, Washington, DC, USA, 1998. IEEE Computer Society.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1974.

[AM93] Arne Andersson and Christer Mattsson. Dynamic interpolation search in o(log log n) time. In *Proceedings of the 20th International Colloquium on Automata, Languages and Programming*, ICALP '93, pages 15–27, London, UK, 1993. Springer-Verlag.

[ASV99] Lars Arge, Vasilis Samoladas, and Jeffrey Scott Vitter. On two-dimensional indexability and optimal range search indexing. In *Proceedings of the 18th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '99, pages 346–357, New York, NY, USA, 1999. ACM.

[AT07] Arne Andersson and Mikkel Thorup. Dynamic ordered sets with exponential search trees. *J. ACM*, 54, June 2007.

[AV88] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31:1116–1127, September 1988.

[AV03] Lars Arge and Jeffrey Scott Vitter. Optimal external memory interval management. *SIAM J. Comput.*, 32:1488–1508, June 2003.

[AVL62] Georgi M. Adel'son-Vel'skii and Evgenii M. Landis. An algorithm for the organization of information. *Soviet Mathematics*, pages 1259–1262, 1962.

[Bay72] Rudolf Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972. 10.1007/BF00289509.

[BCD$^+$02] Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of the 10th Annual European Symposium on Algorithms*, ESA '02, pages 152–164, London, UK, UK, 2002. Springer-Verlag.

[BCR96] Gerth Stølting Brodal, Shiva Chaudhuri, and Jaikumar Radhakrishnan. The randomized complexity of maintaining the minimum. *Nordic J. of Computing*, 3:337–351, December 1996.

[BG90] Gabriele Blankenagel and Ralf H. Gueting. Xp-trees-external priority search trees. Technical Report Informatik-Report Nr.92, Informatik-Bericht, Fern Universitaet, May 1990.

[BGO$^+$96] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. An asymptotically optimal multiversion B-tree. *The VLDB Journal*, 5(4):264–275, 1996.

[BKS01] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The skyline operator. In *Proceedings of the 17th International Conference on Data Engineering*, pages 421–430, Washington, DC, USA, 2001. IEEE Computer Society.

[BKS$^+$09] Gerth Stølting Brodal, Alexis C. Kaporis, Spyros Sioutas, Konstantinos Tsakalidis, and Kostas Tsichlas. Dynamic 3-sided planar range queries with expected doubly logarithmic time. In *Proceedings of the 20th International Symposium on Algorithms and Computation*, ISAAC '09, pages 193–202, Berlin, Heidelberg, 2009. Springer-Verlag.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Inf.*, 1:173–189, 1972.

[BM80] Norbert Blum and Kurt Mehlhorn. On the average number of rebalancing operations in weight-balanced trees. *Theoretical Computer Science*, 11(3):303 – 320, 1980.

[Bro96] Gerth Stølting Brodal. Partially persistent data structures of bounded degree with constant update time. *Nordic J. of Computing*, 3:238–255, September 1996.

[BSTT12] Gerth Stølting Brodal, Spyros Sioutas, Konstantinos Tsakalidis, and Kostas Tsichlas. Fully persistent B-trees. In *Submitted to 23rd ACM-SIAM Symposium on Discrete Algorithms 2012*. 2012.

[BT11] Gerth Stølting Brodal and Konstantinos Tsakalidis. Dynamic planar range maxima queries. In Luca Aceto, Monika Henzinger, and Jirí Sgall, editors, *Automata, Languages and Programming*, volume 6755 of *Lecture Notes in Computer Science*, pages 256–267. Springer Berlin/Heidelberg, 2011. 10.1007/978-3-642-22006-722.

[Com79] Douglas Comer. The ubiquitous B-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[CXP$^+$04] Reynold Cheng, Yuni Xia, Sunil Prabhakar, Rahul Shah, and Jeffrey Scott Vitter. Efficient indexing methods for probabilistic threshold queries over uncertain data. In *Proceedings of the 30th international conference on Very large data bases - Volume 30*, VLDB '04, pages 876–887. VLDB Endowment, 2004.

[dFGT97] Fabrizio d'Amore, Paolo Giulio Franciosa, Roberto Giaccio, and Maurizio Talamo. Maintaining maxima under boundary updates. In *Proceedings of the 3rd Italian Conference on Algorithms and Complexity*, CIAC '97, pages 100–109, London, UK, 1997. Springer-Verlag.

[Die82] Paul F. Dietz. Maintaining order in a linked list. In *Proceedings of the 14th annual ACM symposium on Theory of computing*, STOC '82, pages 122–127, New York, NY, USA, 1982. ACM.

[DIL04] Erik D. Demaine, John Iacono, and Stefan Langerman. Retroactive data structures. In *Proceedings of the 15th annual ACM-SIAM symposium on Discrete algorithms*, SODA '04, pages 281–290, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[DS87] Paul Dietz and Daniel Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th annual ACM symposium on Theory of computing*, STOC '87, pages 365–372, New York, NY, USA, 1987. ACM.

[DSST89] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38:86–124, February 1989.

[FMNT87] Otfried Fries, Kurt Mehlhorn, Stefan Näher, and Athanasios Tsakalidis. A log log n data structure for three-sided range queries. *Inf. Process. Lett.*, 25:269–273, June 1987.

[FR90] Greg N. Frederickson and Susan Rodger. A new approach to the dynamic maintenance of maximal points in a plane. *Discrete Comput. Geom.*, 5:365–374, May 1990.

[FW93] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424 – 436, 1993.

[FW94] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48(3):533 – 551, 1994.

[GS78] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21, Washington, DC, USA, 1978. IEEE Computer Society.

[GTVV93] Michael T. Goodrich, Jyh-Jong Tsay, Darren E. Vengroff, and Jeffery S. Vitter. External-memory computational geometry. In *Proceedings of the 34th Ann. Conf. on Foundations of Computer Science*, pages 714–723, 1993.

[HM81] Scott Huddleston and Kurt Mehlhorn. Robust balancing in B-trees. In *In Proceedings of the 5th GI-Conf. on Theoretical Computer Science*, pages 234–244, 1981.

[HM82] Scott Huddleston and Kurt Mehlhorn. A new data structure for representing sorted lists. *Acta Inf.*, 17:157–184, 1982.

[HT84] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13:338–355, May 1984.

[IKO88] Christian Icking, Rolf Klein, and Thomas Ottmann. Priority search trees in secondary memory (extended abstract). In *Proceedings of the International Workshop WG '87 on Graph-theoretic concepts in computer science*, pages 84–93, New York, NY, USA, 1988. Springer-Verlag New York, Inc.

[Jan91] Ravi Janardan. On the dynamic maintenance of maximal points in the plane. *Information Processing Letters*, 40(2):59 – 64, 1991.

[Kap00] Sanjiv Kapoor. Dynamic maintenance of maxima of 2-d point sets. *SIAM J. Comput.*, 29:1858–1877, April 2000.

[Kap04] Haim Kaplan. Persistent data structures. In Dinesh Mehta and Sartaj Sahni, editors, *Handbook of Data Structures and Applications*, chapter 31, pages 23–1–23–18. CRC Press, 2004.

[KLP75] H. T. Kung, Fabrizio Luccio, and Franco P. Preparata. On finding the maxima of a set of vectors. *J. ACM*, 22:469–476, October 1975.

[KMM$^+$05] Alexis C. Kaporis, Christos Makris, George Mavritsakis, Spyros Sioutas, Athanasios K. Tsakalidis, Kostas Tsichlas, and Christos D. Zaroliagis. ISB-tree: A new indexing scheme with efficient expected behaviour. In Xiaotie Deng and Ding-Zhu Du, editors, *ISAAC*, volume 3827 of *Lecture Notes in Computer Science*, pages 318–327. Springer, 2005.

[KMS$^+$03] Alexis Kaporis, Christos Makris, Spyros Sioutas, Athanasios Tsakalidis, Kostas Tsichlas, and Christos Zaroliagis. Improved bounds for finger search on a RAM. In Giuseppe Di Battista and Uri Zwick, editors, *Algorithms - ESA 2003*, volume 2832 of *Lecture Notes in Computer Science*, pages 325–336. Springer Berlin/Heidelberg, 2003.

[KMS$^+$06] Alexis Kaporis, Christos Makris, Spyros Sioutas, Athanasios Tsakalidis, Kostas Tsichlas, and Christos Zaroliagis. Dynamic interpolation search revisited. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *Automata, Languages and Programming*, volume 4051 of *Lecture Notes in Computer Science*, pages 382–394. Springer Berlin/Heidelberg, 2006.

[Knu77] Don E. Knuth. Deletions that preserve randomness. *IEEE Trans. Softw. Eng.*, 3:351–359, September 1977.

[KPS$^+$10] Alexis Kaporis, Apostolos N. Papadopoulos, Spyros Sioutas, Konstantinos Tsakalidis, and Kostas Tsichlas. Efficient processing of 3-sided range queries with probabilistic guarantees. In *Proceedings of the 13th International Conference on Database Theory*, ICDT '10, pages 34–43, New York, NY, USA, 2010. ACM.

[KRVV96] Paris Kanellakis, Sridhar Ramaswamy, Darren E. Vengroff, and Jeffrey Scott Vitter. Indexing for data models with constraints and classes. *J. Comput. Syst. Sci.*, 52:589–612, June 1996.

[LM91] Sitaram Lanka and Eric Mays. Fully persistent B+-trees. In *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data*, pages 426–435, 1991.

[LS93] David B. Lomet and Betty Salzberg. Exploiting a history database for backup. In *Proceedings of the 19th Int. Conf. on Very Large DataBases*, pages 380–390, 1993.

[Mat91] Jiří Matoušek. Computing dominances inen (short communication). *Inf. Process. Lett.*, 38:277–278, June 1991.

[McC85] Edward M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985.

[Mor06] Christian Worm Mortensen. Fully dynamic orthogonal range reporting on RAM. *SIAM J. Comput.*, 35:1494–1525, June 2006.

[MS81] David Maier and Sharon C. Salveter. Hysterical B-trees. *Inf. Process. Lett.*, 12(4):199–202, 1981.

[MT93] Kurt Mehlhorn and Athanasios Tsakalidis. Dynamic interpolation search. *J. ACM*, 40:621–634, July 1993.

[NR72] Jurg Nievergelt and Edward M. Reingold. Binary search trees of bounded balance. In *Proceedings of the 4th annual ACM symposium on Theory of computing*, STOC '72, pages 137–142, New York, NY, USA, 1972. ACM.

[Ove87] Mark H. Overmars. *Design of Dynamic Data Structures*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1987.

[OvL81] Mark H. Overmars and Jan van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23(2):166 – 204, 1981.

[OW88] Mark H. Overmars and Derick Wood. On rectangular visibility. *J. Algorithms*, 9:372–390, September 1988.

[Pet57] W. W. Peterson. Addressing for random-access storage. *IBM J. Res. Dev.*, 1:130–146, April 1957.

[RS94] Sridhar Ramaswamy and Sairam Subramanian. Path caching (extended abstract): a technique for optimal external searching. In *Proceedings of the 13th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '94, pages 25–35, New York, NY, USA, 1994. ACM.

[SMK$^+$04] Spyros Sioutas, Christos Makris, Nectarios Kitsios, Georgios Lagogiannis, Jannis Tsaknakis, Kostas Tsichlas, and Vasillis Vassiliadis. Geometric retrieval for grid points in the RAM model. *Journal of Universal Computer Science*, 10(9):1325–1353, 2004.

[SR95] Sairam Subramanian and Sridhar Ramaswamy. The p-range tree: a new data structure for range searching in secondary memory. In *Proceedings of the 6th annual ACM-SIAM symposium on Discrete algorithms*, SODA '95, pages 378–387, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics.

[ST99] Betty Salzberg and Vassilis J. Tsotras. Comparison of access methods for time-evolving data. *ACM Comput. Surv.*, 31(2):158–221, 1999.

[Sun89] Rajamani Sundar. Worst-case data structures for the priority queue with attrition. *Inf. Process. Lett.*, 31:69–75, April 1989.

[Tar83] Robert Endre Tarjan. Updating a balanced search tree in o(1) rotations. *Information Processing Letters*, 16(5):253 – 257, 1983.

[Tsa84] Athanasios K. Tsakalidis. Maintaining order in a generalized linked list. *Acta Inf.*, 21:101–112, June 1984.

[Tsa86] Athanasios K Tsakalidis. AVL-trees for localized search. *Inf. Control*, 67(1-3):173–194, 1986.

[Vit08] Jeffrey Scott Vitter. Algorithms and data structures for external memory. *Found. Trends Theor. Comput. Sci.*, 2(4):305–474, 2008.

[VV97] Peter J. Varman and Rakesh M. Verma. An efficient multiversion access structure. *IEEE Trans. on Knowl. and Data Eng.*, 9:391–409, May 1997.

[Wil00] Dan E. Willard. Examining computational geometry, van emde boas trees, and hashing from the perspective of the fusion tree. *SIAM J. Comput.*, 29(3):1030–1049, 2000.

[WL85] Dan E. Willard and George S. Lueker. Adding range restriction capability to dynamic data structures. *J. ACM*, 32:597–617, July 1985.