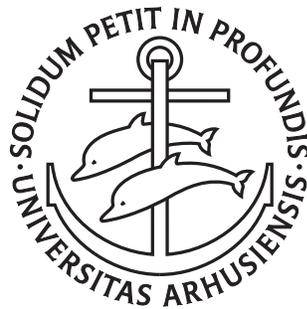# Data Structures: Range Queries and Space Efficiency

Pooya Davoodi

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

Data Structures:

Range Queries and Space Efficiency

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

by
Pooya Davoodi
Last Revision: May 7, 2011

Abstract (English)

We study data structures for variants of range query problems. In particular, we consider (1) the range minimum problem: given a multidimensional array, find the position of the minimum element in a query rectangle; (2) the path minima problem: given a weighted tree, find the edge with minimum weight in a query path; (3) the range diameter problem: given a point set in the plane, find two points that are farthest away in a query rectangle. These and similar problems arise in various applications including document retrieval, genome sequence analysis, OLAP data cubes, network flows, shape-fitting, and clustering.

The three mentioned problems are considered for either static inputs or dynamic inputs. In the static setting, we investigate the space-efficiency of data structures, which is an important aspect in massive data algorithmics. We provide lower bounds on the trade-off between the query time and the space usage of range minimum and range diameter data structures. We also present data structures for these problems to either complement the lower bounds or beating the lower bounds under certain assumptions about inputs. One of the results proves that to answer a multidimensional range minimum query, the product of the number of cells that we need to read and the number of bits stored in indexing data structures is at least equal to the number of elements in input arrays. Another result shows that using at most linear bits in addition to an array, we can achieve constant query time to support two-dimensional range minimum queries.

In the dynamic setting, we present data structures for the path minima problem that support optimal query time for various types of update operations. One of the results presents a comparison-based data structure which answers path minima queries in sub-logarithmic time and supports updating the edge-weights of input trees in optimal logarithmic time. We also prove lower bounds on trade-offs between the query time and the update time of path minima data structures.

Finally, we study the space-efficiency of cardinal tree representations in the dynamic setting, which has practical applications, for example in text indexing structures. We present a succinct dynamic data structure that supports traversing low-arity cardinal trees while answering queries during the traversal.

Abstract (Danish)

Afhandlingen beskæftiger sig med forskellige varianter af range query problemer. Specifikt, så studerer vi (1) range minimum problemet: givet et multidimensionalt array, find positionen af et minimalt element indenfor et query rektangel; (2) sti minima problemet: givet et vægtet træ, find en kant med minimum vægt langs stien mellem to knuder; (3) range diameter problemet: given en punktmængde i planen, rapporter to punkter der er længst fra hinanden indenfor et query rektangel. Disse og beslægtede problemer har anvendelser indenfor mange område, så som dokument genfinding, analyse af genom-sekvenser, OLAP data cubes, netværks-strømninger, formtilpasning, og beregning af klynger i data.

De tre nævnte problem betragtes for enten statiske eller dynamiske input. I det statiske tilfælde, studerer vi pladseffektiviteten af datastrukturer - et vigtigt aspekt indenfor massiv data algoritmik. Vi viser nedre grænser for trade-off'et mellem forespørgselstiden og pladsforbruget for datastrukturer for range minimum og range diameter problemerne. Vi præsenter også datastrukturer for disse problemer, som enten komplimenterer de nedre grænser eller som faktisk er bedre end de nedre grænser under visse antagelser omkring inputtet. Et af resultaterne er at for a kunne svare på multidimensionalle range minimum queries, skal produktet af det antal celler vi læser og størrelsen af en indekseringsdatastruktur i bits være mindst antallet af elementer i inputtet. Et andet resultat vi viser er at man kan understøtte to-dimensionale range minimum queries i konstant tid ved kun at gemme et lineært antal bits ud over input arrayet.

For dynamiske datastrukturer viser vi hvordan man kan opnå optimale query tider for forskellige typer af opdateringer. Et resultat er en sammenligningsbaseret datastruktur, der understøtter sti minima queries i sublogaritmisk tid og opdateringer af kant-vægte i input træet i optimal logaritmisk tid. Vi viser også nedre grænser for trade-offs mellem query tiden og opdateringstiden for sti minima datastrukturer.

Det sidste emne vi studerer er plads-effektiviteten af dynamiske repræsentationer af kardinaltræer, som bl.a. anvendes i tekstindekserings strukturer. Vi præsenterer en succinct dynamisk datastrukturer der understøtter kardinaltræer af lav maksimal grad, og som understøtter forespørgsler under traverseringen af træet.

Abstract (Persian)

در این پایان‌نامه به مطالعه ساختمان‌های داده‌ای می‌پردازیم که در مسائلی کاربرد دارند که در آنها به سوالاتی درباره یک بازه یا محدوده پاسخ داده می‌شوند. مسائلی که مورد بررسی قرار می‌گیرند عبارتند از (۱) کوچکترین در بازه: دردرون یک آرایه چند بعدی، مکان کوچکترین عنصر در داخل یک بازه مستطیلی داده شده را بیابید؛ (۲) کوچکترین در مسیر: در یک درخت که هر یالش دارای یک وزن می‌باشد، یالی که دارای کوچکترین وزن در داخل که مسیر داده شده می‌باشد را بیابید؛ (۳) قطر در بازه: در یک مجموعه از نقاط در صفحه، دو نقطه را که در داخل یک بازه مستطیلی، دورترین فاصله را از هم دارند بیابید. مسائلی از این دست کاربردهای مختلفی دارند، به عنوان نمونه در بازیابی اسناد، تحلیل توالی‌های ژنوم، پایگاه داده‌ها، شبکه‌های جریان، مطابق کردن اشکال، و خوشه‌سازی.

سه مسئله ذکر شده یا برای ورودی های ایستا بررسی می‌شوند و یا برای ورودی‌های پویا. در حالت ایستا، حافظه مصرفی ساختمان‌های داده‌ای مورد بررسی قرار می‌گیرند. ما در مسائل کوچکترین در بازه و قطر در بازه، به ارائه چندین کران پایین و کران بالا برای حافظه مصرفی و زمان اجرایی می‌پردازیم. یکی از نتایجی که در این تحقیق به دست آمد اثبات این است که برای یافتن موقعیت کوچکترین عنصر در یک بازه از آرایه، تعداد خانه‌هایی از حافظه که نیاز به خواندنشان داریم ضربدر اندازه یک ساختمان دادۀ نمایه ساز در مقیاس بیت، حداقل برابر است با تعداد عناصر موجود در آرایه. یکی دیگر از نتایج تحقیق نشان می‌دهد که برای یافتن موقعیت کوچکترین عنصر در یک بازه از آرایه‌های دو بعدی در زمانی با پیچیدگی ثابت، ذخیره تعدادی بیت که با یک تابع خطی از اندازه آرایه ورودی بیان می‌شود کافی است.

در حالت پویا، ما چندین ساختمان داده‌ها برای مسئله کوچکترین در مسیر ارائه می دهیم که همگی بهترین زمان اجرایی ممکن را دارند و انواع مختلفی از بروزرسانی را پشتیبانی می کنند. در یکی از نتایج، ما یک ساختمان داده مقایسه‌ای ارائه می دهیم که کوچکترین وزن در مسیرها را در زمانی کمتر از تابع لگاریتم می‌یابد و بروزرسانی وزنها را در زمان لگاریتمی انجام می‌دهد. ما همچنین به اثبات چندین کران پایین برای زمان بروزرسانی و زمان پاسخگویی مسئله کوچکترین در مسیر می‌پردازیم.

در انتها، نمایاندن درختان کاردینال پویا که دارای کاربردهایی از جمله در نمایه‌سازی متن هستند را از نظر حافظه مصرفی مورد مطالعه قرار می‌دهیم. یک ساختمان داده فشرده پویا ارائه می‌دهیم که پیمایش یک درخت کاردینال با حداکثر درجه پایین را به همراه پاسخگویی به پرسشهایی در طول پیمایش پشتیبانی می‌کند.

Preface

This dissertation is devoted to two aspects of data structures and algorithms: space-efficient data structures and range queries. Data structures with agile queries on static inputs are usually the commencement of space-efficiency and lead to the study of time-space trade-offs in a broad range of computational problems. Furthermore, data structures on dynamic inputs give rise to the analysis of query-update trade-offs. The crossing point of these two types of trade-offs is the subject of succinct dynamic data structures, where we care about all: optimal space, query time, and update time. In this dissertation, I consider these issues of efficiency for three different versions of the range query problem, and for a fundamental non-range query problem.

Structure of the dissertation.

- **Chapter 1 (Introduction)**: In the first chapter, I give a detailed overview of the subject of this dissertation. In particular, I define the different problems that are considered in the next chapters, and survey the literature of those problems. At the end of the chapter, I mention some preliminaries that are essential or useful for the other parts of the dissertation.

- **Chapter 2 (Range Minimum Queries)**: The content of this chapter is based on a paper published in the proceedings of ESA 2010 [BDR10] that was later invited to the special issue of Algorithmica for the conference which is now in press [BDR11a]. This is a joint work with Gerth Stølting Brodal and S. Srinivasa Rao under the title of "On Space Efficient two-dimensional Range Minimum Data Structures".

  In this chapter, we study range minimum query (RMQ) data structures, which support finding the position of the minimum element in a query rectangle within a given array. We provide an optimal time-space trade-off for one-dimensional RMQ indexing data structures (Sections 2.2.1 and 2.2.2). Also the first $O(N)$-bit space two-dimensional RMQ indexing data structure with constant query time for arrays that contain N elements is given (Section 2.2.3). Furthermore, for two-dimensional RMQ indexing data structures, the first time-space trade-off in terms of both lower bounds and upper bounds is proved (Sections 2.2.1 and 2.2.4). Also an information-theoretic lower bound for the two-dimensional RMQ problem is established, and a constant query time encoding two-dimensional RMQ data structure that leaves a gap for the space bound of such structures is given (Section 2.3). Finally, it is shown that the lower bound for the time-space trade-off of the two-dimensional RMQ problem also holds for multidimensional RMQ indexing data structures (Section 2.2.1).

- **Chapter 3 (Path Minima Queries)**: An extended abstract of this chapter is going to appear in the proceedings of WADS 2011 [BDR11b]. This is a joint work with Gerth Stølting Brodal and S. Srinivasa Rao under the title of "Path Minima Queries in Dynamic Weighted Trees".

  In this chapter, it is demonstrated how to answer path minima queries in optimal time under various update operations on trees (Sections 3.2 and 3.3). This problem is considered in different computational models. The difficulty of the dynamic path minima problem is also analyzed by proving some lower bounds for query-update trade-offs by reductions (Section 3.4).

- **Chapter 4 (Range Diameter Queries)**: This chapter consists of some new results recently obtained in a joint work with Michiel Smid and Freek van Walderveen.

  In this chapter, we study the hardness of the range diameter problem, which asks for the furthest points in a query rectangle within a given point set. In particular, we give a reduction from a fundamental data structure problem to the two-dimensional range diameter problem. This reduction implies conditional lower bounds for the space requirement of range diameter data structures (Section 4.2). Also a lower bound for computing the furthest points in two vertically-separated convex polygons is shown (Section 4.3). Finally, the problem is investigated for convex polygons. We show that two-dimensional range diameter queries can be supported more efficiently when point sets are convex and have low-modality (Section 4.4).

- **Chapter 5 (Succinct Representation of Dynamic Cardinal Trees)**: The content of this chapter is based on a paper that is going to appear in the proceedings of TAMC 2011 [DR11]. This is a joint work with S. Srinivasa Rao under the title of "Succinct Dynamic Cardinal Trees with Constant Time Operations for Small Alphabet".

  In this chapter, a succinct dynamic data structure is presented that uses almost optimal space to represent k-ary cardinal trees for polylogarithmic k. The data structure supports the standard navigational operations and a selection of enhanced queries in $O(1)$ time under leaf insertions and deletions performed in $O(1)$ amortized time. In this data structure, the operations are performed in the course of traversing input trees.

Contents

Chapter 1

Introduction

It was a large room. Full of people. All kinds.
And they had all arrived at the same building at more or less the same time.
And they were all free.
And they were all asking themselves the same question:

What is behind that curtain?
— Anderson Laurie, "Born, Never Asked Lyrics", 1982.

For more than three decades, one of the hot areas of research in design and analysis of data structures has been *range query* problems. The goal is to preprocess a set of objects that have some positional relationship to each other, such that we can efficiently report or count all the objects that are within a query range, such as the points that are within a query rectangle for a given point set in the plane. In a general setting, a certain aggregation function that operates on the objects within a query range should be computed efficiently. For example, for a given point set in the plane, an aggregation function finds the top most point within a query range. These types of queries are known as *range aggregate* queries. Also, in a more general setting, each object can be associated with some weight, and aggregation functions operate on the weights of objects that are within a query range.

Range query problems for geometric objects such as points, line segments, polygons, and polyhedra in the *d*-dimensional Euclidean space $\mathbb{R}^d$, are known as geometric range searching in computational geometry [AE99, Afs08b]. The relationship between geometric objects is their relative positions in some coordinate system, and queries ask to report the intersections of objects with a query range. Geometric range searching has many practical applications, for example in database systems (such as multi-key searching [Knu98, Section 6.5]), time-series (temporal) databases [AE99], geographic information systems (such as traffic monitoring [TP04]), statistics [BF79], and automation in VLSI design [SG06]. Some important range searching problems that have attracted the attention of many geometers are dominance queries [CE87, Afs08a], orthogonal range queries [Meh84, PS85, AE99], simplex and halfspace range searching [Wil82, HW87, Mat92, Mat93, Cha10]. These queries are depicted in Figure 1.1 for two dimensional point sets.

In geometric range searching, in addition to the typical queries: reporting, counting and emptiness, range aggregate queries are also studied. For example, for a given point set, consider finding the closest pair of points among the points within a query range.

dominance query	orthogonal range query	simplex range query	half-space range query
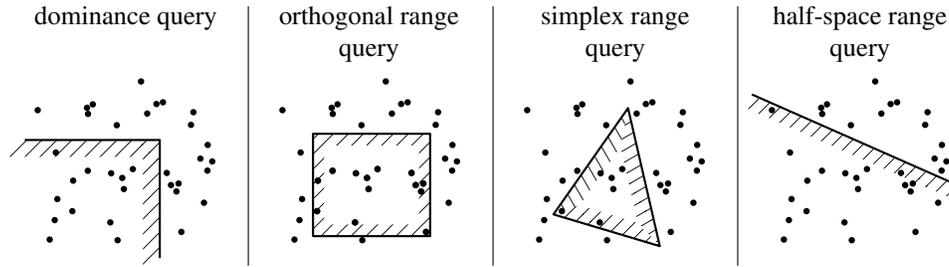
Figure 1.1: Different geometric range searching problems can arise according to different query shapes enclosing the objects. Four different geometric range queries are depicted for two dimensional points.

As another example, for a set of points that each is associated with a numerical weight, queries ask for the point with maximum weight among the points within a given range.

A special case of geometric range searching is when inputs are dense point sets in $\mathbb{R}^d$ such that the points can be placed in the cells of a d-dimensional array (grid). In this case, the inputs are d-dimensional arrays, where each entry of the arrays can maintain a weight assigned to a point. In Section 1.2, we provide an overview of this problem.

Another variant of range queries is when inputs are trees. In this case, objects are vertices of trees, and the relationship among vertices is defined by the edges incident on the vertices. A range in a tree can for example be defined by a path in the tree which is induced by two vertices. The goal is to report or count all vertices (or edges) contained in a range, or to compute a function of them. In a more general setting, a weight is assigned to each vertex (or edge), and then a function of the weights whose corresponding vertices are within a query range has to be computed. In Section 1.3, we sketch this problem.

1.1   Space Efficiency and Succinctness

The amount of data that is collected for various scientific applications is increasing extremely fast. We need efficient algorithms and data structures that can handle massive data. The usual measures of efficiency of data structures are

- *Time*: How long a preprocessing algorithm takes to process an input and construct a data structure, and how much time a query algorithm takes to answer a given query and produce the desired output. If the data structure supports update operations, then the time of those operations also matters.

- *Space*: How much memory a preprocessing algorithm takes to construct a data structure and store it. The most important space bound is the size of the data structure after the preprocessing algorithm is done.

In order to prevent expansion of memory usage, we are usually interested in data structures using linear space. But at the same time, we would like to answer queries as fast as we can, say constant time. For many computational problems, achieving these two goals (linear space and constant query time) at the same time is not an easy

task, and sometimes it is impossible (for example, searching for a number in a list). In particular, there is usually a trade-off between the space bound and the query time. That is, large-sized data structures can support queries fast, whereas small-sized data structures cannot answer queries quickly. A similar trade-off can happen between the preprocessing time and the query time, and also between the query time and the update time. One of the topics of this dissertation is considering these trade-offs for different problems. In particular, we study the time-space trade-offs for range minimum queries in arrays, and range diameter queries in two dimensional point sets, and we also study the query-update trade-offs for path minima queries in trees. These problems will be defined and described in detail in the next sections.

We are usually satisfied with linear space data structures, but sometimes we can aim at making the space bounds even smaller. Indeed, data compression has always been an important field of research in computer science in an attempt to represent data in sublinear space. But, compressing data is not necessarily satisfactory for data structure problems, because we need to answer online queries fast, without decompressing the data structures. In his PhD dissertation, Guy Jacobson wrote: "Small is beautiful. It is good when a piece of data can be made smaller. It is bad, however, when this reduction in size is accompanied by a reduction in accessibility as well, but this is the compromise made in classical data compression. Sometimes such a compromise is unacceptable." [Jac89]. We need to optimize data compression techniques such that they can transform inputs into small-sized data structures, while preserving important functionality such as supporting queries or modifying data. Designing *succinct* data structures introduced by Jacobson [Jac89] is an answer to this problem.

1.1.1 Succinct Data Structures

The goal of succinct data structures is to use the power of the unit-cost word RAM model to store data in a number of bits close to the information-theoretic lower bound, while supporting queries in optimal time. For example, to represent a binary tree containing n nodes, the information-theoretic lower bound is $2n - \Theta(\log n)$ bits. This bound is derived by taking the logarithm of the number of different binary trees of n nodes captured by the Catalan number $C_n = \frac{1}{n+1}\binom{2n}{n}$. For an input binary tree containing n nodes, we would like to design data structures of size close to $2n - \Theta(\log n)$ bits, that can support navigational operations in the tree such as returning the left child of a given node. Many different types of problems have been considered in the literature on succinct data structures such as dictionaries, and pattern matching [DPM05]. Trees are one of the fundamental structures in computing. In the following, we consider succinct representations of trees which is one of the subjects of this dissertation (Chapter 5). The word size of the machine is usually $w = \Theta(\log n)$ bits, where n is the input size, for example for tree representations, n is the number of nodes.

Binary trees. We have to represent an input binary tree containing n nodes in a number of bits close to the information theoretic lower bound $2n - \Theta(\log n)$, and support the navigational operations left-child, right-child, parent for any given node in the tree. In most of the applications, a piece of data of fixed size, termed satellite data, is associated to each node of the tree. We can allocate an array containing n cells and store the satellite data in the cells of the array. Thus, we map the nodes of the tree onto

the integers in $\{1, \ldots, n\}$, and the navigational operations return integers representing their output nodes. Hence, we can use these integers to access the array containing the satellite data.

Consider a bit vector of length $2n + 1$ made by performing a left-to-right level-order (the breadth first) traversal of the tree, and storing 1 bits for the present nodes and 0 bits for the absent children. In the traversal of the tree, each node attains an integer rank according to its appearance in the traversal. Let this rank be the interface of the navigational operations performed on the tree. Let the operation rank(i) return the number of 1s before the position i in the bit vector, and the operation select(i) return the position of the i-th 1 in the bit vector. The navigational operations on the tree can be supported using the following equations:

$$\text{parent}(i) = \text{select}(\lfloor i/2 \rfloor) \, ,$$

$$\text{left-child}(i) = 2 \cdot \text{rank}(i) \, ,$$

$$\text{right-child}(i) = 2 \cdot \text{rank}(i) + 1 \, .$$

It is known that $o(n)$ bits are sufficient to store a data structure that supports rank and select in $O(1)$ time on a bit vector of length n [Mun96]. Thus, a binary tree containing n nodes can be represented with the bit vector along with a rank-select data structure using $2n + o(n)$ bits to support the navigational operations in $O(1)$ time, and accessing the satellite data [Jac89, Cla96, CM96].

Ordinal trees.   An ordinal tree is a rooted tree where each node can have an arbitrary number of children in some order. There is a one-to-one correspondence between ordinal trees containing n nodes and binary trees containing $n - 1$ nodes (Section 1.5.3). Therefore, the information theoretic lower bound to represent ordinal trees containing n nodes is also $2n - \Theta(\log n)$ bits. The operations left-child and right-child mentioned for binary trees are replaced with the operation i-th child, which returns the i-th child of any given node in the tree. Therefore, the navigational operations are i-th child and parent. Every ordinal tree representation can also represent a binary tree within the same time and space complexities to support the navigational operations ([MR01] and Section 1.5.3), but the opposite is not clear. A typical operation on ordinal trees is degree, which returns the number of children of a given node.

Jacobson presented the level-order unary degree sequence (LOUDS) representation of ordinal trees containing n nodes, which supports the navigational operations in $O(1)$ time and can be stored in $2n + o(n)$ bits [Jac89]. A tree is represented by the sequence containing the degrees of the nodes in the level-order traversal, in which each degree d is written as a string of d 1s and a 0.

There are applications in which we need to determine the size of the subtree rooted at each node of the tree [MR01]. Therefore, we would like to design ordinal tree representations that also support the operation subtree-size in addition to the navigational operations. The LOUDS representation supports i-th child in $O(1)$ time because there is a simple relationship between a node and its children. But it cannot support subtree-size in constant time, since it requires to traverse a subtree to determine its subtree size.

Munro and Raman [MR01] overcame the difficulty of subtree-size by representing an ordinal tree of size n using a balanced sequence of parentheses. This sequence is

derived from the depth-first traversal of the tree, by assigning an open parenthesis to a node when we visit the node, and assigning a close parenthesis to a node after we visit the whole subtree of the node. They showed that the representation can be stored in $2n + o(n)$ bits and it can support subtree-size in $O(1)$ time because each subtree is represented as a *contiguous* balanced sequence of parentheses. But the representation requires $\Theta(i)$ time to support i-th child [DPM05]. The representation maps each node of the tree to the rank of the node in the preorder traversal of the tree called the preorder number of the node. This representation is known as *balanced parentheses representation* (BPS). They also show that their data structure can be used to support the subtree size queries in binary trees [MR01, Section 3.2].

Benoit, Demaine, Munro, Raman [BDMR99] introduced the depth-first unary degree sequence (DFUDS) representation which overcomes the difficulty of supporting both subtree-size and i-th child in constant time by combining the virtues of LOUDS and BPS. They create a balanced parentheses sequence containing the unary degree sequence of each node but in the order of the depth-first traversal of the tree, in contrast to LOUDS which is in level-order. That is, the representation of each node contains essentially the same information as in LOUDS, but the nodes are stored in a different order. The DFUDS representation can be stored in $2n + o(n)$ bits and can support the navigational operations, degree, and subtree-size, all in $O(1)$ time [BDM$^+$05].

Cardinal trees. A k-ary cardinal tree is a rooted tree in which each node has at most k children and each edge is labeled by a symbol from a totally ordered set of fixed size k. Another term for k-ary cardinal trees is *tries* with degree k. For each node in such a tree, the edges between the node and its children are sorted increasingly according to their labels. We assume that $k \leq n$. The number of different k-ary cardinal trees containing n nodes is $\mathscr{C}(n,k) = \binom{kn+1}{n}/(kn+1)$ [GKP88, Arr08]. Thus, if k is a non-decreasing function of n, the information theoretic lower bound to represent k-ary cardinal trees is $n(\log e + \log k)$ bits.

We can represent a k-ary cardinal tree using ordinal tree representations. But if we would like to perform operations on the tree that are concerned with the edge-labels, then we need to maintain some information about the edge-labels. One of these operations can be label-child(α) which returns the child of a given node, where the edge between the child and the node is labeled by α. To maintain the edge-labels, for each node of the tree we can make a dictionary structure that contains the edge-labels of all the children of that node. Now, we can perform label-child for a given node using its corresponding dictionary along with the operation i-th child on the tree. In particular, to find label-child(α) of a given node x, we determine whether α exists in the dictionary corresponding to x. If it does, we compute the rank i of α, that is, the number of labels existing in the dictionary which are smaller than α plus one. The desired child is thus i-th child of x.

In an experimental work, Darragh, Cleary, and Witten [DCW93] showed a compact form of representing k-ary cardinal trees named *Bonsai*, by storing $6 + \lceil \log k \rceil$ bits per node. Their method is essentially a way of storing trees in a compact form of hash tables. Their structure supports navigation through cardinal trees and inserting new leaves in $O(1)$ expected time.

Benoit, Demaine, Munro, and Raman [BDMR99] turned the attention of the suc-

cinct community from binary trees to cardinal trees of higher degrees. They represent k-ary cardinal trees containing n nodes using $2n + n\lceil \log k \rceil + o(n)$ bits, and their data structure supports the navigational operations in $O(1)$ time except label-child which takes $O(\log \log k)$ time. Their structure was improved by using static dictionaries that support constant time rank operation [RR99], leading to an improvement for label-child to $O(1)$ time by spending totally $2n + n\lceil \log k \rceil + o(n) + O(\log \log k)$ bits [BDM$^+$05]. The space bound of their structure is $\log \mathscr{C}(n,k) + \Omega(n)$ bits, as k grows. Later, Raman, Raman, and Rao [RRS07] improved the space to $\log \mathscr{C}(n,k) + o(n) + O(\log \log k)$ bits, while supporting all the operations except subtree-size in $O(1)$ time. Finally, Farzan, Raman, and Rao [FRR09] presented a data structure that achieves $O(1)$ time for all the operations including subtree-size using the same amount of space $\log \mathscr{C}(n,k) + o(n) + O(\log \log k)$ bits.

1.1.2   Succinct Dynamic Data Structures

In succinct data structures, we arrange bits of preprocessed data in an optimal-length sequence of memory words, such that queries can be supported efficiently using the sequence. Of course, finding such an arrangement is much easier if the input data is static [Jac89]. Considering the dynamic succinct data structures goes back to 1994, where Brodnik and Munro [BM99] showed how to store a subset of a universe with space bound within a small constant factor of the minimum required and to support membership queries in constant time and insertions and deletions in constant expected amortized time. Other fundamental problems such as dynamic arrays, dynamic partial sums, and dynamic dictionaries have also been considered [DPM05]. In the following we describe dynamic succinct representations of trees.

Dynamic binary trees.   Munro, Raman, and Storm [MRS01] considered succinct representations of dynamic binary trees for the first time. They presented a data structure of size $2n + o(n)$ bits which supports the navigational operations left-child, right-child, parent in $O(1)$ time, and allows inserting a node along an edge, inserting a leaf, deleting a node with one child, and deleting a leaf all in $O(\log^2 n)$ time in the worst case, or in $O(\log \log n)$ amortized time. When constant-sized satellite data is associated with the nodes, their data structure supports accessing the data in $O(1)$ time with $O(\log^3 n)$ worst case update time and $O(\log n)$ amortized update time. Also if the size of satellite data for each node is $\Theta(\log n)$ bits, then updating the structure takes $O(\log^4 n)$ worst case time and $O(\log^2 n)$ amortized time. Of course, the space used to store satellite data is added to the size of the data structure.

Farzan and Munro [FM10] presented a succinct representation of dynamic ordinal trees (described later in this section), which can be used as a representation for binary trees. In fact, every ordinal tree representation also supports navigation and updates in binary trees within the same time and space complexities, using the transformation algorithm mentioned in Section 1.5.3 (see also [MR01, Section 3.2]).

The data structure of [FM10] can represent ordinal trees of size n nodes with satellite data of size $b = O(\log n)$ bits using $2n + bn + o(bn)$ bits. It supports the basic navigational operations parent, first-child, last-child, next-sibling, and previous-sibling, and accessing and modifying satellite data in $O(1)$ time, and supports insertions and deletions in $O(1)$ amortized time (the similar types of insertions and deletions

as in [MRS01]).

Subtree sizes in dynamic binary trees. As mentioned for static binary tree representations, determining the subtree size of a given node is an interesting operation for some applications. Farzan and Munro [FM10] proved that maintaining a binary tree containing n nodes under insertions and deletions of leaves and subtree size queries requires $\Omega(\log n / \log\log n)$ amortized time per operation. They determined the lower bound by a reduction from the subset rank problem as follows.

The subset rank problem is representing a subset $S \subseteq \{1, \ldots, n\}$ under insertions and deletions of elements, and rank which asks for the number of elements that are smaller than a query element. They showed that the subset rank problem for S can be transformed into representing the following binary tree under updates and subtree size queries: consider a left skewed path of length n (where each node has only a left child, except the bottom most one); correspond each node to an element in $\{1, \ldots, n\}$; for each node in the path, if its corresponding element appears in S, give it a right child. The subset rank problem also has the lower bound of $\Omega(\log n / \log\log n)$ amortized time per operation in the cell probe model with words of length polylogarithmic in n [FS89].

To overcome this difficulty of supporting subtree size queries in dynamic setting, Munro, Raman, and Storm [MRS01] introduced the *traversal model* by putting some restrictions on the update operations and subtree size queries. The restriction is that updates are performed in the course of a traversal which starts from the root, moves through the tree by performing the navigational operations at the current node, and ends at any node. Subtree size queries can be answered for every node only after the traversal is completed. In other words, imagine that there is a finger which is initially at the root, and it crawls on the tree using the navigational operations. The update operations can be only performed on the finger, and subtree size queries can be only answered after the navigation ends (the finger stops traveling in the tree). In this model, the update time and the query time can be amortized over the movements of the finger, as well as the time to perform the updates or to answer the queries. To achieve worst case bounds for the update time, they assume that the traversal (the finger) stops at the root.

In the traversal model, the dynamic data structure presented in [MRS01] can support the navigation, accessing satellite data, and subtree size queries in $O(1)$ time using $2n + o(n)$ bits. The update time of the data structure is the same as mentioned before considering subtree size queries.

Raman and Rao [RR03] improved the update time to $O((\log\log n)^{1+\varepsilon})$ amortized for satellite data of size $b = O(\log n)$ bits, while making the traversal model slightly stronger by answering subtree size queries at any time during the traversal (not only on completion of the traversal) but only at the current node. The size of their data structure is $2n + bn + o(n)$ bits.

Dynamic ordinal trees. Farzan and Munro [FM10] showed the difficulty of supporting the enhanced navigational operation i-th child in ordinal trees, in similar to supporting subtree size queries in dynamic binary trees. In particular, they showed that maintaining ordinal trees containing n nodes under insertions and deletions of leaves, accessing satellite data, and the navigational operations i-th child and par-

ent requires $\Omega(\log n/\log\log n)$ amortized time per operation. They proved the lower bound by a reduction from the list representation problem. In the list representation problem, we have to maintain an ordered list of at most n elements from $\{1,\ldots,n\}$ under insertions, deletions, and reporting the element at a given position in the list. The list representation problem also has the lower bound of $\Omega(\log n/\log\log n)$ amortized time per operation in the cell probe model with words of length polylogarithmic in n [FS89].

Gupta, Hon, Shah, and Vitter [GHSV07] presented a framework to dynamize succinct data structures focusing on achieving information-theoretically optimal space along with near-optimal update time and query time for different data structures such as dictionaries and trees. They dynamized the LOUDS representation of ordinal trees, and achieved a dynamic data structure of size $2n+o(n)$ bits to represent ordinal trees containing n nodes that supports the navigational operations in $O(\log\log n)$ time, and the update operations in $O(n^\varepsilon)$ amortized time, for any constant $\varepsilon > 0$.

Sadakane and Navarro [SN10] in an attempt to simplify the existing succinct representations of static ordinal trees, and cutting the redundancy of those representations, proposed a technique to reduce the large number of relevant tree operations to a few primitives. Moreover, they derived a dynamic data structure for ordinal trees that support plenty of operations. Their representation for ordinal trees of size n can be stored in $2n+o(n)$ bits, and supports all the operations in $O(\log n/\log\log n)$ time.

To overcome the difficulty of supporting i-th child in dynamic setting, Farzan and Munro [FM10] utilized a stronger version of the traversal model[1], in which the navigation and queries can be performed at any time at any node, while updates are only performed at the finger. In this model, they presented a dynamic data structure to maintain ordinal trees containing n nodes with satellite data of size $b = O(\log n)$ bits, using $2n+bn+o(bn)$ bits. Their data structure supports the basic navigational operations and the operations i-th child, degree, and subtree-size all in $O(1)$ worst case time, and updates in $O(1)$ amortized time, degree returns the number of children of a given query node.

Cardinal trees.   Apart from the standard operations of dynamic ordinal trees, and label-child for cardinal trees, we would like to have insert-label-leaf(α), which inserts a leaf as a child of a given node, where the edge between the leaf and the node is labeled by α, and also delete-label-leaf(α) defined analogously.

As previously mentioned, Darragh, Cleary, and Witten [DCW93] in an experimental work, presented the Bonsai structure to represent k-ary cardinal trees containing n node using $6n+n\lceil\log k\rceil$ bits. The Bonsai data structure supports the navigational operations and inserting new leaves in $O(1)$ expected time.

In response to the question posed in [MRS01] about representing dynamic k-ary cardinal trees succinctly, Arroyuelo [Arr08] presented a data structure of size $2n+n\log k+o(n\log k)$ bits, that supports standard operations of dynamic cardinal trees containing n nodes in the traversal model, where all the supported operations can be performed at the current node at any time (this model is the same as the one used in [RR03]). His data structure supports the navigational operations, subtree-size, and degree in $O(\log k+\log\log n)$ time and updates in $O((\log k+\log\log n)(1+$

---

[1]In their paper, this model is denoted as the *finger-update* model [FM10]

$(\log k)/(\log(\log k + \log \log n))))$ amortized time. These bounds appear due to using dynamic rank and select data structures for general alphabet size, using searchable partial sums structures, and using dynamic DFUDS representations. This data structure does not support accessing satellite data. For small alphabet size that has practical motivations, in particular $k = (\log n)^{O(1)}$, Arroyuelo's data structure achieves $O(\log \log n)$ query time and $O((\log \log n)^2 / \log \log \log n)$ amortized update time. One of the contributions of this dissertation is improving these bounds to constant (Chapter 5).

The traversal model used in cardinal trees representations has practical applications such as constructing Lempel-Ziv indexes which is used for dynamic compressed full-text indexes; and also constructing suffix trees, if we supplement the data structure with satellite data.

Our contribution. Munro, Raman, and Storm [MRS01] asked for representing k-ary cardinal trees succinctly while they speculated that their techniques for dynamic binary trees may be suitable for k-ary cardinal trees as well. Arroyuelo [Arr08] presented a k-ary cardinal tree structure and mentioned that improving his data structure for small alphabets, such as polylogarithmic in the size of the input tree, is interesting. We achieved this, by giving a succinct representation of dynamic k-ary cardinal trees that uses $2n + n \log k + o(n \log k)$ bits for an input k-ary cardinal tree containing n nodes, where $k = (\log n)^{O(1)}$. Low-arity cardinal trees have practical applications, for example in text indexing [BDMR99, Arr08]. Our data structure works in the traversal model and supports the navigational operations in constant time, insertions and deletions of leaves in amortized constant time, and the following enhanced queries in constant time: determine the subtree size and the degree of the current node of the traversal; verify if the current node is the ancestor of a given query node. This data structure is presented in Chapter 5.

1.1.3 Succinct Indexing Data Structures

In some practical applications, it is needed to keep the original input (raw data) in memory for some reasons unrelated to answering queries, for example someone else needs the raw data for his applications. That is, the input is preprocessed and a data structure is constructed which supports queries, but the input is also maintained even though the data structure does not need the input any more. In these cases, it makes sense to design data structures that utilize the existence of the input. It allows us to make smaller data structures that answer queries by consulting the input, denoted as *indexing data structures*. For example, in information retrieval systems, we can use inverted indexes which essentially store the answer of all the queries in typically $50\% - 300\%$ of the size of the input. This enables the system to answer the queries very fast without consulting the input. On the contrary, we can make an indexing data structure of size about 2%-4% of the size of the original input which answers the queries slower but fast enough (GLIMPSE as a searching tool is an example of such an index [MW94]). According to the nomenclature of succinct data structures, indexing data structures are also denoted as systematic schemes contrary to non-systematic schemes or encoding data structures which do not need the input to answer the queries [GM07].

1.2   Range Queries in Arrays

In this section, we consider the range query problem for input arrays. A d-dimensional array $A[1 \cdots n_1] \times [1 \cdots n_2] \times \cdots \times [1 \cdots n_d]$ is a table of elements, where $A[i_1, i_2, \ldots, i_d]$ denotes an element of the array addressed by the indexes $i_1, i_2, \ldots, i_d$, for $1 \leq i_k \leq n_k$ and $1 \leq k \leq d$. In this section, we consider range query problems for such arrays. An orthogonal d-dimensional range query $q = (i_1, j_1, i_2, j_2, \ldots, i_d, j_d)$ in an array A, spans all the cells in $A[i_1 \cdots j_1] \times [i_2 \cdots j_2] \times \cdots \times [i_d \cdots j_d]$.

In contrast to geometric range searching, where counting the objects within the range, or determining the emptiness of the range all require non-trivial data structures, in the case of arrays, these can be trivially solved by doing calculation on the indexes $i_1, j_1, i_2, j_2, \ldots, i_d, j_d$ (no preprocessing is required). The problem arises when the query asks to compute a function of the elements, where some information about the elements is required to compute the function.

In some versions of the problem, the function to be computed is an arithmetic function such as *sum*: the sum of the elements within the range. In this case the elements of the input array are numbers (integers or reals). In some other versions, the function is a comparison based function such as *minimum* (*maximum*): the smallest (greatest) element within the range; *median*: the element of rank $\lceil s/2 \rceil$ among all the s elements within the range. In this case, the elements of the array are some members of a totally ordered set. There are also some versions in which the function is an equality based function such as *count*: the number of distinct elements within the range. In Section 1.2.1, we will study a version of the problem where the function is minimum, which is one of the subjects of this dissertation (Chapter 2).

Let $N = |A| = n_1 \cdot n_2 \cdots n_d$ be the size, the total number of the elements, of the input array A. There are two naive solutions to answer range queries in A:

- Brute force search: Do not need to store anything during the preprocessing. To answer a query, scan all the elements within the query range, and compute the answer. This solution is not fast in terms of the query time. In particular, it takes at least $O(N)$ query time to just read the elements within the range, in the worst case. Although it essentially does not use any space to store a data structure. The space usage of the solution is $O(N)$ to store the input array.

- Tabulation: During the preprocessing, compute the answer of every possible query and arrange all the answers in a table (which is a two dimensional array). Answer the query by looking up the table at the position corresponding to the query. The size of the table is in the order of the total number of queries, which is as large as $O(N^2)$. A naive preprocessing algorithm takes at least $O(N^3)$ time to make the table by just reading all the elements within each possible query to compute the answer of the query. Though, the query algorithm only takes $O(1)$ time to determine the answer. Notice that the query algorithm does not need to read any element from the input array. Therefore, in this solution, the input array is not stored in the data structure.

1.2.1 Range Minimum Queries

In this problem, the input is a *d*-dimensional array *A*, and the elements of *A* are from a totally ordered set. The query asks for the minimum element among the elements within the query range. Notice that the query can be answered by finding the position of the minimum element within the range, and then returning the element in that position. Indeed, in the literature due to some applications such as simulating suffix tree based algorithms on suffix arrays and LCP arrays [FHS08], the range minimum query denotes the version of the problem, where the query asks for the *position* of the minimum element within the range. In this thesis from now on, we follow the same definition. In the following two sections, we review the one dimensional and multidimensional versions of the problem, and at the end we mention our contribution.

Applications. The RMQ problem for one dimensional arrays has applications in for example, range queries [Sax09], text indexing [AKO04, FMN08, Sad07a], text compression [CPS08], document retrieval [Mut02, Sad07b, VM07], flowgraphs [GT04], and position-restricted pattern matching [ICK$^+$08]. The RMQ problem for two dimensional arrays has applications in computer graphics, image processing (for example, finding the lightest or darkest point in a range), computational biology (for example, finding minimum or maximum number in an alignment tableau or in genome sequence analysis), and databases (for example, range minimum or maximum query in OLAP data cubes [Poo03]).

One dimensional RMQ. In the one dimensional range minimum query problem (1D-RMQ), the input is an array $A[1 \cdots n]$ containing *n* elements. This problem has been studied extensively.

Gabow, Bentley and Tarjan in their seminal paper [GBT84], solved several geometric range searching problems including finding the point with minimum weight in a given query range for a *d*-dimensional point set. Their solution resolves the 1D-RMQ problem with $O(1)$ query time by spending $O(n)$ preprocessing time and space. They recognized the *Cartesian tree* that was introduced by Vuillemin [Vui80] in the context of average time analysis of searching. The Cartesian tree for the array *A* is a binary tree with nodes labeled by the indexes of *A*. The root has label *i*, where $A[i]$ is the minimum element in *A*. The left subtree of the root is a Cartesian tree for the subarray $A[1 \cdots i-1]$, and the right subtree of the root is a Cartesian tree for the subarray $A[i+1 \cdots n]$. This inductive definition implies that the answer to any range minimum query $q = (i, j)$ in *A*, is the label of the lowest common ancestor (LCA) of the nodes labeled by *i* and *j*. The Cartesian tree of *A* can be constructed in $O(n)$ time [Vui80]. The LCA of two nodes in a tree of size *n* can be found in $O(1)$ time using $O(n)$ preprocessing time and space in the word RAM due to Harel and Tarjan [HT84]. In the following, we briefly explain a relationship between the 1D-RMQ problem and the LCA problem.

Equivalence between 1D-RMQ and LCA. The LCA problem is one of the most fundamental problems in computer science [AGKR04] that was introduced by Aho, Hopcroft, and Ullman [AHU73]. The problem has several variants in terms of offline/online and static/dynamic [HT84]. In the online setting, we preprocess an input

tree and then we answer incoming queries one by one by finding the lowest common ancestor of two given nodes. This problem for a static tree of size n was resolved by Harel and Tarjan [HT84] in a complicated data structure of size $O(n)$, constructed in $O(n)$ time that achieves $O(1)$ query time. This structure works in the the word RAM (in the pointer machine, $\Omega(\log \log n)$ query time is required [HT84]). There have been works on simplifying the data structure of [HT84], for example see [SV88].

Gabow, Bentley and Tarjan [GBT84] showed that the LCA problem for a tree can be solved using a 1D-RMQ structure. The input array to the 1D-RMQ problem can be derived by listing the heights of the nodes in an inorder traversal of the tree. Essentially, this reduction and the Cartesian tree solution for the 1D-RMQ together indicate that these two problems, the 1D-RMQ and LCA, are equivalent. Although this is a simple reduction from the LCA problem to the 1D-RMQ problem, but this did not give any solution for the LCA problem (hopefully simpler than [HT84]). Because at that time, there was no $O(1)$-query time data structure to solve the 1D-RMQ problem except the one that relies on LCA in the Cartesian tree.

Berkman and Vishkin [BGSV89] presented a simple LCA PRAM-algorithm by reducing it to a restricted-domain version of the 1D-RMQ problem denoted by ± 1RMQ, in which each element of the array differs by $+1$ or -1 from its adjacent. The array is derived by listing the heights of the nodes traversed in the Euler tour of the tree. Notice that the difference between the heights of each pair of successive nodes in this list is exactly one. They solved the ± 1RMQ with $O(1)$ query time, and $O(n)$ preprocessing time and space, by partitioning the array and precomputing the answers for small subarrays (similar to the structure of [AS87]). Later, the sequential version of their algorithm was demonstrated in a simplified presentation in [BFCP$^+$05] arguing that the difficulty in implementing the LCA algorithms should not prevent us anymore to solve problems through giving algorithms that rely on the LCA problem. See [AGKR04, FH06] for other results on the 1D-RMQ problem.

Succinct 1D-RMQ structures.   The 1D-RMQ data structures can be classified into two different types: *indexing* and *encoding*. Recall that indexing data structures support queries by using both verbatim input arrays and some additional information stored along the inputs. Notice that if all the entries of the input arrays are distinct, explicitly storing an input array takes $\Omega(n \log n)$ bits. The goal is to reduce the *additional space* as much as possible, while supporting the queries fast. See Section 1.1.3 for more details about indexing data structures. On the contrary, encoding data structures must be able to support the queries without consulting the input array. The information theoretic lower bound to represent the 1D-RMQ data structures is $2n - \Theta(\log n)$ bits. This lower bound is derived from the fact that the Cartesian tree of an input array can be constructed by answering some of the 1D-RMQs over the input array, and also all the 1D-RMQs over the input array can be answered using the Cartesian tree of the input array. Since the Cartesian tree is a binary tree, the logarithm of the Catalan number $C_n = \frac{1}{n+1} \binom{2n}{n}$ as the number of binary trees containing n nodes, gives the lower bound.

Sadakane [Sad07b] presented a 1D-RMQ encoding data structure that only uses $4n + o(n)$ bits of space while supporting the queries in $O(1)$ time. Later, Fischer and Heun [FH07] presented an indexing 1D-RMQ data structure of additional

space $2n + o(n)$ bits that supports queries in $O(1)$ time. Recently, Fischer [Fis10] improved the size of the best 1D-RMQ encoding data structure to $2n + o(n)$ bits, which supports the queries in $O(1)$ time. He introduced a new data structure named 2d-Min-Heap instead of using the Cartesian tree.

1D-RMQ in the semigroup model The binary operation $\min(\alpha, \beta)$ is a semigroup operation that returns the minimum of α and β, without giving any information about the position of the minimum. In the following, we illustrate how the 1D-RMQ problem can be considered in the semigroup model.

First consider a related problem, *the partial sums* problem which has been considered extensively in the semigroup model. The partial sums problem asks to preprocess an input array $[x_1, \ldots, x_n]$ of real numbers and answer queries of the form $q_{i,j} = \sum_i^j x_k$. The problem can be trivially solved using linear space with constant query time, by storing the answer of all the prefix queries $q_{1,i}$, and retrieving the answer of $q_{i,j}$ by subtracting the answer of $q_{1,i-1}$ from the answer of $q_{1,j}$.

Yao [Yao82] considered the partial sums problem[2] in a restricted model, where the addition operation $+$ is the only available operation of algorithms to operate on elements. He proved that any data structure stored in m memory cells, where $m \geq n \geq 1$, requires to perform $\Theta(\alpha(m,n) + \frac{n}{m-n+1})$ addition operations during the querying to retrieve the answer of the queries. Yao proved that this lower bound also applies to the partial sums problem in the semigroup model, where the elements x_i are from a semigroup, and the addition operation can be any semigroup operator.

Yao ignored the time needed to find the proper memory cells for answering the queries [Yao82]. Thus, his upper bound does not apply to the RAM. Later, Chazelle and Rosenberg [CR89] showed how to achieve such an upper bound in the RAM, where the elements x_i are from a semigroup. Since the minimum operation is a semigroup operator, their data structure can trivially solve the version of the 1D-RMQ problem, where the queries ask for the minimum element within the range instead of the position of the minimum. Their data structure can also be used to solve the original version of the 1D-RMQ problem as follows. In the input, replace each x_i with (x_i, i) and use the following function as the semigroup operator:

$$f((x_i, i), (x_j, j)) = \begin{cases} (x_i, i) & \text{if } x_i \leq x_j \\ (x_j, j) & \text{otherwise .} \end{cases}$$

This gives a 1D-RMQ data structure of size $O(n)$ that can answer the queries in $O(\alpha(n))$ time.

Multidimensional RMQ. In the d-dimensional RMQ problem, the input is an array $A[1 \cdots n_1] \times [1 \cdots n_2] \times \cdots \times [1 \cdots n_d]$ containing $N = n_1 \cdot n_2 \cdots n_d$ elements.

As previously described, Gabow, Bentley and Tarjan [GBT84] solved the 1D-RMQ problem for an array of size n, in order to accomplish better bounds for several d-dimensional geometric range searching problems including finding the point with minimum weight in a given d-dimensional query rectangule. Their solution for d-dimensional range searching problems can be used to solve the d-dimensional RMQ

---

[2]In his paper, this problem is named as "Interval Query" [Yao82].

Table 1.1: Results for the 1D-RMQ problem for an input array of n elements. The parameter c is an integer, where $1 \le c \le n$. The last three rows present results for succinct indexing data structures, which are stored in addition to the input array. Our results appear in boldface.

Reference	Space	Query Time
[GBT84, BGSV89, AGKR04] [BFCP$^+$05, FH06]	$O(n)$	$O(1)$
[Sad07b]	$4n + o(n)$ bits	$O(1)$
[Fis10]	$2n + o(n)$ bits	$O(1)$
[FH07]	$2n + o(n)$ bits	$O(1)$
Section 2.2.2	$O(n/c)$ bits	$O(c)$
Section 2.2.1	$O(n/c)$ bits	$\Omega(c)$

problem for arrays, by transforming each element of the input array A to a point placed in a cell of a d-dimensional grid in $\mathbb{R}^d$. This gives a data structure that supports RMQs in $O(\log^{d-1} N)$ query time with $O(N \log^{d-1} N)$ preprocessing time and space.

Chazelle and Rosenberg [CR89] in the semigroup model, considered the d-dimensional *partial-sum* problem defined as follows[3]. Preprocess a d-dimensional array A containing N elements from a semigroup, and then given a d-dimensional rectangular range query, compute the sum of all the elements within the range. They presented a data structure of size $O(N)$ that works in the word RAM, and supports the queries in $O((\alpha(N))^d)$ time after spending $O(N)$ preprocessing time. More precisely, for $O(M)$ preprocessing time and space, they achieve $O((\alpha(M,N))^d)$ query time for any M, where $M \ge 14^d N$. The term $\alpha(M,N)$ is the two-parameter inverse of Ackermann's function (Section 1.5.5). Notice that minimum is a semigroup function, and therefore their solution also solves the d-dimensional RMQ problem, and improves the time and space bounds of the solution of [GBT84].

Amir, Fischer, and Lewenstein [AFL07] considered the two dimensional version of the problem, the 2D-RMQ problem. They improved the query time $O((\alpha(N))^2)$ of [CR89] to $O(1)$ time by spending more preprocessing time and space, specifically $O(N \log^{(k+1)} N)$ preprocessing time and $O(kN)$ space, for any $k > 1$, where $\log^{(k+1)} N$ is the result of applying the function $\log$ for $k+1$ times on N.

For the 2D-RMQ problem that has an n by n input array, Demaine, Landau, and Weimann [DLW09a] proved that any data structure that can support queries without accessing the input array at the querying stage, requires $\Omega(n^2 \log n)$ bits of space and preprocessing time. This lower bound is derived by taking the logarithm of the number of different input arrays with respect to 2D-RMQs, where two arrays are considered different only if there exists at least a 2D-RMQ with different answers in those two arrays. They showed that the number of such different arrays is $\Omega((\frac{n}{4}!)^{n/4})$. This lower bound is extended in this thesis to $\Omega(mn \log m)$ for m by n input arrays, where $m \le n$ (Section 2.3.2).

Recently, Atallah and Yuan [AY10] achieved $O(N)$ preprocessing time and space while supporting d-dimensional RMQs in $O(1)$ time.

---

[3]The upper bound result of this paper did not appear in the journal version [CR91]

Table 1.2: Results for the 2D-RMQ problem for an m by n input array containing N elements, where $m \leq n$. The parameter c is an integer, where $1 \leq c \leq N$. The lower bound of [DLW09a] is for an n by n input array. The last three rows present results for succinct indexing data structures, which are stored in addition to the input array. Our results appear in boldface.

Reference	Preprocessing time	Space	Query time
[GBT84]	$O(N \log N)$	$O(N \log N)$	$O(\log N)$
[CR89]	$O(N)$	$O(N)$	$O((\alpha(N))^2)$
[AFL07]	$O(N \log^{(k+1)} N)$	$O(kN)$	$O(1)$
[AY10]	$O(N)$	$O(N)$	$O(1)$
[DLW09a]	-	$\Omega(n^2 \log n)$ bits	-
Section 2.3.1	$O(N)$	$O(mn \cdot \min\{m, \log n\})$ bits	$O(1)$
Section 2.3.2	-	$\Omega(mn \log m)$ bits	-
Section 2.2.3	$O(N)$	$O(N)$ bits	$O(1)$
Section 2.2.4	$O(N)$	$O(N/c)$ bits	$O(c \log^2 c)$
Section 2.2.1	-	$O(N/c)$ bits	$\Omega(c)$

Table 1.3: Results for the d-dimensional RMQ problem for an input array containing N elements. The parameter c is an integer, where $1 \leq c \leq N$. The last row presents a lower bound for succinct indexing data structures, which are stored in addition to the input array. Our results appear in boldface.

Reference	Preprocessing time	Space	Query time
[GBT84]	$O(N \log^{d-1} N)$	$O(N \log^{d-1} N)$	$O(\log^{d-1} N)$
[CR89]	$O(N)$	$O(N)$	$O((\alpha(N))^d)$
[AY10]	$O(N)$	$O(N)$	$O(1)$
Section 2.2.1	-	$O(N/c)$ bits	$\Omega(c)$

Dynamic RMQ structures. The dynamic 1D-RMQ problem asks for dynamic data structures that support 1D-RMQs under updating the elements of the input arrays at arbitrary positions. In the following, we mention known lower bounds for this problem. Let t_u be the time to update an entry of the array using a dynamic 1D-RMQ data structure.

It can be proved that on a *cell probe* model with word size b bits, 1D-RMQs on an array of size n require $\Omega(\log n / \log(t_u b \log n))$ time. For example, the update time $(\log n)^{O(1)}$ implies query time $\Omega(\log n / \log \log n)$. This lower bound is derived by reduction from the priority search trees problem defined as follows. We have to maintain a set $S \subseteq \{1, \ldots, n\}$ under insertions and deletions where we assign a weight from $\{1, \ldots, n\}$ to each element by the insertions, and a query for a given element $e \in \{1, \ldots, n\}$, asks for the lowest weight among all the weights whose elements are in S and are not larger than e.

The priority search trees problem can be solved using a dynamic 1D-RMQ structure as follows. Represent the set S with an array $A[1 \cdots n]$ initially containing $+\infty$ in all the entries. To insert an element e with weight w into S, perform $A[e] = w$,

and to delete an element e from S, perform $A[e] = +\infty$. A query for a given element e can be answered by the 1D-RMQ for the range $A[1 \cdots e]$. The above mentioned lower bound was proved for the priority search trees problem by Alstrup, Husfeldt, and Rauhe [AHR98, Section 2.2], which then holds for the dynamic 1D-RMQ problem as well.

Brodal, Chaudhuri, and Radhakrishnan [BCR96] considered the problem of maintaining the minimum of a set under insertions and deletions. They proved a lower bound in the *comparison model*, for a slightly more difficult version of the problem in which the delete operation is provided with the location of the element that is going to be deleted. It is obvious that a dynamic 1D-RMQ structure can also solve this problem by showing almost the same reduction used above for the priority search trees problem. Thus, the following lower bound is derived by applying the lower bound proved in [BCR96] for the minimum maintenance problem. The lower bound implies that if updates perform at most t_u comparisons then 1D-RMQs require at least $n/(e2^{2t_u}) - 1$ comparisons. For example, if 1D-RMQs use $(\log n)^{O(1)}$ comparisons then updates require $\Omega(\log n)$ comparisons.

Consider a version of the 1D-RMQ problem in which queries ask for the minimum element within a given range, instead of asking for the position of the minimum. The partial sums problem in the *semigroup model* is a generalization of this version of the 1D-RMQ problem. For the dynamic version of the partial sums problem in the semigroup model, Pătrașcu and Demaine [PD06] proved the lower bounds $t_q \log(t_u/t_q) = \Omega(\log n)$ and $t_u \log(t_q/t_u) = \Omega(\log n)$, where t_q and t_u denote the query time and update time respectively. For example, the update time $O(\log n)$ implies the query time $\Omega(\log n)$ and vice versa.

In Section 1.3.1, we will consider a generalized version of the dynamic 1D-RMQ problem, denoted as the *dynamic path minima* problem. Not only the above lower bounds hold for the path minima problem in the corresponding models, but also the upper bounds for the path minima problem hold for the dynamic 1D-RMQ problem. One of the results of this thesis is achieving optimal upper bounds for the dynamic 1D-RMQ problem through solving the path minima problem.

Another version of the dynamic 1D-RMQ problem is when we want to maintain a 1D-RMQ data structure under insertions and deletions of elements at arbitrary positions in the input array. As far as we know, there is no specific result for this problem. Recall that the standard data structure to solve the 1D-RMQ problem is a Cartesian tree with an LCA structure constructed for it. Bialynicka-Birula and Grossi [BBG06] showed how to maintain a Cartesian tree under insertions and deletions performed in $O(\log n)$ amortized time. Each update operation changes the topology of the Cartesian tree drastically, and this becomes an obstacle in maintaining the LCA structure under such changes in the tree, although it is known that we can maintain a constant-query time LCA structure under insertions and deletions of nodes performed in $O(1)$ time [CH05]. Notice that maintaining the input array of the 1D-RMQ problem explicitly under insertions and deletions, which shift the entries appropriately, is not interesting due to the difficulty of the list representation problem [FS89].

Our contributions.   We list our results for the RMQ problem that are presented in Chapter 2. We investigate the problem in terms of both indexing data structures and

Table 1.4: Results for the dynamic 1D-RMQ problem. Upper bounds are shown at the top, and lower bounds are shown at the bottom. The lower bounds should be read like a conditional sentence, for example, the result from [BCR96] states that if the query time is $\log^{O(1)} n$, the update time $\Omega(\log n)$ is required. Our results appear in boldface.

Ref.	Preprocessing time	Query time	Space	Update time	Model
[ST83]	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$	Semigroup
Section 2.4	$O(n)$	$O(\frac{\log n}{\log\log n})$	$O(n)$	$O(\log n)$	Comparison
Section 2.4	$O(n)$	$O(\frac{\log n}{\log\log n})$	$O(n)$	$O(\frac{\log n}{\log\log n})$	RAM
[PD06]	-	$\Omega(\log n)$	-	$O(\log n)$	Semigroup
[BCR96]	-	$\log^{O(1)} n$	-	$\Omega(\log n)$	Comparison
[AHR98]	-	$\Omega(\frac{\log n}{\log\log n})$	-	$\log^{O(1)} n$	RAM

encoding data structures.

- An optimal time-space trade-off for 1D-RMQ indexing data structures (Sections 2.2.1 and 2.2.2): for an input array containing *n* elements, we show that using an indexing data structure of size $O(n/c)$ bits, the query time $\Theta(c)$ is the best and sufficient, for any *c* where $1 \le c \le n$.

- The first $O(N)$-bit space 2D-RMQ indexing data structure with constant query time, for an input array containing *N* elements (Section 2.2.3).

- The first time-space trade-off for 2D-RMQ indexing data structures in terms of both lower bounds and upper bounds (Sections 2.2.1 and 2.2.4): for an input array containing *N* elements, we prove that with an indexing data structure of size $O(N/c)$ bits, $\Omega(c)$ query time is the best possible, while our structure supports the queries in $O(c \log^2 c)$ time, for any *c* where $1 \le c \le n$.

- An information-theoretic lower bound for the 2D-RMQ problem, and a constant query time encoding 2D-RMQ data structure that leaves a gap for the space bound of such structures (Section 2.3): the gap is proved to be between $\Omega(mn \log m)$ and $O(mn \log n)$ bits, for *m* by *n* input arrays.

- The first time-space trade-off for multidimensional RMQ indexing data structures in terms of lower bounds (Section 2.2.1): generalizing the proof of the lower bound for the 2D-RMQ problem, for an input array containing *N* elements, we prove that with an indexing data structure of size $O(N/c)$ bits, $\Omega(c)$ query time is the best possible.

For the indexing data structures, we prove the lower bounds in a non-uniform cell probe model, and we achieve the upper bounds using partitioning and tabulation techniques. For the encoding data structures, the lower bound for the 2D-RMQ problem

is derived by a counting argument, and the upper bound is given by encoding the input arrays with the rank of the elements in the arrays and making an indexing data structure for it.

1.3   Range Queries in Trees

In this section, we consider the range query problems in trees. A range query $q = (u, v)$ asks a question about all the nodes along the path $(u, \ldots, v)$ between the nodes u and v of T. Notice that in every tree, there is exactly one path between every two nodes. A version of the problem is when the input tree is a rooted tree. This opens the door to queries dependent on the hierarchical structure of the input tree such as LCA queries.

A simple range query may ask for the number of nodes along the query path. But in a common generalization of the range query problem in trees, each edge of T is assigned a weight, where the weight can be a number or a character from a universe. In this case, queries ask to compute a function of the weights whose corresponding edges are within the query range. Notice that the weights can be assigned to the nodes instead of the edges, but this does not make the problem very different. Similar to the range query problem for arrays, the function to be computed for weights can be an arithmetic-based function such as sum, a comparison-based function such as minimum, or a equality-based function such as counting the number of distinct weights.

The range query problem for trees is a generalization of the range query problem for one dimensional arrays. This comes from the fact that if a tree is one path that contains all the nodes of the tree, then the tree can be seen as a one dimensional array that contains the edge-weights. This is particularly interesting from the lower bound point of view, where we prove the hardness of the range query problem for trees. Because this generalization implies that the range query problem for trees is at least as hard as the range query problem for one dimensional arrays.

Similar to the range query problem for arrays, the brute force search and the tabulation method are two naive solutions for the range query problem for trees. In the following, we explain these two solutions for an input tree containing n nodes and a range query $q = (u, v)$.

- Brute force search: Do not need to store anything during preprocessing T. To answer the query q, traverse the tree starting from u and ending at v. While scanning the path $(u, \ldots, v)$, compute the answer of q. This solution is not fast in terms of the query time. In particular, it takes at least $O(n)$ query time to just access the nodes within the range, in the worst case. Although it essentially does not use any space to store a data structure. The space usage of the solution is $O(n)$ to store the input tree explicitly using pointers.

- Tabulation: During preprocessing of T, compute the answer of every possible query and arrange all the answers in a table (which is a two dimensional array). Answer the query q by looking up the table at the position corresponding to q. The size of the table is in the order of the total number of queries, which is as large as $O(n^2)$. A naive preprocessing algorithm takes $\Theta(n^3)$ time to make the table by just traversing the tree to compute the answer of each query. Though, a reasonable query algorithm only takes $O(1)$ time to determine the answer from

the table. Notice that a query algorithm does not need to traverse T or access any node in T. Therefore, in this solution, the tree is not stored in the data structure.

1.3.1 Path Minima Queries

The path minima problem is to preprocess an input tree in which each edge is assigned a weight from a totally ordered set, such that we can find the edge with minimum weight along a given query path. This query is denoted as pathmin[4]. Another type of query is when we have to find the minimum weight instead of finding the edge with minimum weight in a given path. This type of queries can be answered by first finding the answer of the corresponding path minima queries, and then returning the weight associated to the found edge. Therefore, if we have access to the edge-weights of the input trees, this type of queries is at least simpler the path minima queries.

Offline setting. In the offline setting of the path minima problem, m queries are given in advance together with a tree of size n. This problem is a generalization of the minimum spanning tree verification problem. In the comparison model, the problem can be solved in $O(m+n)$ time using $O(n)$ space [Kin97]. See [Tar78, Kom85, CR91, DRT92] for other results on the static offline version in different models.

Online setting. Chazelle [Cha87] considered the online version of the path minima problem. For input trees containing n nodes, when the edge-weights are from a semi-group, he solved the problem with $O(\alpha(n))$ query time, and $O(n)$ preprocessing time and space (see Section 1.5.5 for the definition of $\alpha(n)$ and its other variants). Later, in an attempt at giving a trade-off between the preprocessing time and the query time, Alon and Shieber showed that with $O(nk\alpha_k(n))$ preprocessing time and $O(n)$ space, a query can be answered in $4k-1$ semigroup operations [AS87]. In the comparison model, where the only operations allowed on the edge-weigths are comparisons, Pettie [Pet06] achieved a slight decrease in the preprocessing time $O(nk\log\alpha_k(n))$ with $4k-1$ query complexity.

The problem has also the following known lower bounds which imply that the above upper bounds are almost optimal. Alon and Shieber [AS87] proved that to answer the queries using $4k-1$ semigroup operations, $\Omega(n\alpha_{2k}(n))$ preprocessing time is required. Notice that $\alpha_{2k}(n) = o(\alpha_k(n))$ for $k > 1$ (Section 1.5.5). Pettie [Pet06] proved a slightly higher lower bound in the comparison model, which improves the lower bound of [AS87] but yet leaves a small gap in both the comparison model and the semigroup model. He defines a function λ_k, where $\alpha_k(n) < \lambda_{2k-1}(n) < \alpha_{k-1}(n)$, for $k \geq 1$ as n grows. He states that for k query time, $\Omega(n\log\lambda_k n)$ preprocessing time is required. When only $O(n)$ preprocessing time is allowed, an $\Omega(\alpha(n))$ query lower bound is known in both models [AS87, Pet06].

The above lower bounds give trade-offs between the preprocessing time and the query time of the data structures. But also limiting the space of the data structures gives lower bounds for the query time. As mentioned in Section 1.2.1, a special case of the path minima problem is the 1D-RMQ problem. Indeed, the 1D-RMQ problem for an input array can be solved by solving the path minima problem for an input tree

---

[4]In [DLW09a], the path minima problem is named "Bottleneck Edge Query problem" (BEQ).

Table 1.5: Results for the static version of the path minima problem in both the offline and online settings. The lower bounds should be read like a conditional sentence, for example, the result from [Pet06] states that if the preprocessing time is $O(n)$, the query time $\Omega(\alpha(n))$ is required. The size of the input tree is denoted by n, and k and c are arbitrary parameters, where $1 \leq k \leq \alpha(n)$ and $1 \leq c \leq n$. In the last row of the table, t denotes the number of bits required to explicitly store the input tree. Note that the semigroup lower bounds of [CR91, AS87] are in stronger models.

Ref.	Time		Space	Model
	Offline for m queries			
[Kom85]	$O(m+n)$ comparisons		$-$	Comparison
[Kin97]	$O(m+n)$		$O(n)$	Comparison
[DRT92]	$O(m+n)$		$O(n)$	RAM
[Tar78]	$\Omega((m+n)\cdot\alpha(m+n,n))$		$-$	Semigroup
[CR91]	$\Omega((m+n)\cdot\alpha(m+n,n))$		$-$	Faithful Semigroup

Ref.	Preprocessing time	Query time	Space	Model
	Online			
[Cha87]	$O(n)$	$O(\alpha(n))$	$O(n)$	Semigroup
[AS87]	$O(n\cdot k\cdot\alpha_k(n))$	$4k-1$	$O(n\cdot k\cdot\alpha_k(n))$	Semigroup
[DLW09a]	$O(n\log^{(k)}n)$	$4(k-1)$	$O(n)$	Comparison
[Yao82]	$-$	$\Omega(\alpha(n))$	$O(n)$	Semigroup
[AS87]	$\Omega(n\cdot\alpha_{2k}(n))$	$4k-1$	-	Commutative Semigroup
[AS87]	$\Omega(n)$	$O(\alpha(n))$	-	Commutative Semigroup
[Pet06]	$\Omega(n)$	$O(\alpha(n))$	-	Comparison
[Pet06]	$\Omega(n\log\lambda_k n)$	k	-	Comparison
[BDR11a]	-	$\Omega(c)$	$O(n/c)+t$ bits	RAM

that is a path and has the entries of the array as its edge-weights in the appropriate order. Thus, the following two lower bounds hold for the path minima problem by reduction from the 1D-RMQ problem: (1) When the edge-weights are from a semigroup, every path minima data structure of size $O(n)$, requires $\Omega(\alpha(n))$ query time [Yao82]; (2) In the cell probe model, every indexing data structure of size $O(n/c)$ bits that solves the path minima problem, requires $\Omega(c)$ query time [BDR11a] (notice that for the latter lower bound and generally for indexing data structures, we assume that inputs, here the trees including their edge-weights, are given in a read-only array).

As mentioned in Section 1.2.1, Cartesian trees are a standard data structure to solve the 1D-RMQ problem. Interestingly, Cartesian trees can be used to solve the path minima problem as well. However, while it is known that in the comparison model, linear time is enough to construct a Cartesian tree of an array using an incremental fashion [GBT84], but $\Omega(n\log n)$ comparisons are required to construct a Cartesian tree of a

tree containing *n* nodes. This lower bound is derived from a reduction from the sorting problem to making a Cartesian tree of a star-tree containing *n* nodes [DLW09a]. It is known that for an input tree containing *n* nodes in the comparison model, $O(n \log^{(k)} n)$ preprocessing time is sufficient to answer the path minima queries in at most $4(k-1)$ comparisons, for $k \geq 1$, where $\log^{(k)} n$ is the *k*-th iterated version of the logarithm function [BMN+04, DLW09a]. Notice that this upper bound is worse than the one of [Pet06] but it is achieved by a simpler algorithm.

Dynamic weighted trees.   The dynamic version of the path minima problem can be defined by providing some update operations performed on the input tree. We may be interested in updating the edge-weights of the tree, and/or interested in updating the topology of the input tree. In the latter case, we can think of insertions and deletions of nodes in the input tree, or in a more general setting, we are given a collection of trees (a forest) as the input and we can update this forest by linking and cutting the trees, or making new trees. In the following, we define a set of update operations such that providing various subsets of them defines different variants of the dynamic path minima problem.

- update(*e*,*w*): change the weight of the edge *e* to *w*.

- insert(*e*,*v*,*w*): split the edge $e = (u_1, u_2)$ by inserting the node *v* along it. The new edge (u_1, v) has weight *w*, and (u_2, v) has the old weight of *e*.

- insert-leaf(*u*,*v*,*w*): add the node *v* and the edge (u, v) with weight *w*.

- contract(*e*): delete the edge $e = (u, v)$, and contract *u* and *v* to one node.

- delete-leaf(*v*): delete both the leaf *v* and the edge incident to it.

- link(*u*,*v*,*w*): add the edge (u, v) with weight *w* to the forest, where *u* and *v* are in two different trees.

- cut(*e*): delete the edge *e* from the forest.

Notice that to use the operations link and cut, we should also provide an operation that can make new trees, for example one that can make a new single-node-tree that contains a given new node. Different variants of the dynamic path minima problem have been considered in the literature, where none of them consider the operation update, although link and cut together can update an edge-weight. In the following, we mention these variants.

In their seminal paper, Sleator and Tarjan [ST83] solved the most general version of the dynamic path minima problem, in which all the above update operations are provided. They presented a data structure denoted as *dynamic trees* or *link-cut trees*. When the edge-weights are from a semigroup, their data structure supports many operations including link, cut, root and evert all in $O(\log n)$ amortized time. The operation root finds the root of the tree containing a given node, and evert changes the root of the tree containing a given node such that the node becomes the root, by turning the tree "inside out". This data structure can solve all the variants of the path minima problem, however it does this through supporting the expensive operations root and

Table 1.6: Results for the dynamic path minima problem. The table is divided into four parts corresponding to four variants of the problem according to the supported update operations that are mentioned at the top of each part. For the first two parts, the input is a tree containing n nodes. For the last two parts, the input is a collection of trees totally containing n nodes. The lower bounds should be read like a conditional sentence. For example, the last row of the table states that if the query time is $O(\log n/(\log\log n)^2)$, then $(\log n)^{\Omega(\log\log n)}$ update time is required. For the first row of the table, k is an arbitrary parameter, where $1 \leq k \leq \alpha(n)$. For the results of [DLW09a], u denotes the size of the universe from which the edge-weights are. Our results appear in boldface.

Reference	Preprocessing time	Query time	Space	Update time	Model
update operations: insert-leaf, delete-leaf					
Section 3.3	$O(nk\alpha_k(n))$	$4k$	$O(nk\alpha_k(n))$	$O(k\alpha_k(n))$	Semigroup
Section 3.3	$O(n)$	$O(\alpha(n))$	$O(n)$	$O(1)$	Semigroup
[DLW09a]	$O(n\log n)$	$O(1)$	$O(n)$	$O(\log n)$	Comparison
[DLW09a]	$O(n\log n)$	$O(1)$	$O(n)$	$O(\log\log u)$	RAM
[AH00, KS08], **Section 3.3**	$O(n)$	$O(1)$	$O(n)$	$O(1)$	RAM
update operations: update, insert, insert-leaf, contract					
Section 3.2	$O(n)$	$O(\frac{\log n}{\log\log n})$	$O(n)$	$O(\log n)$	Comparison
Section 3.2	$O(n)$	$O(\frac{\log n}{\log\log n})$	$O(n)$	$O(\frac{\log n}{\log\log n})$	RAM
[PD06]	-	$\Omega(\log n)$	-	$O(\log n)$	Semigroup
[PD06]	-	$O(\log n)$	-	$\Omega(\log n)$	Semigroup
[BCR96]	-	$\log^{O(1)} n$	-	$\Omega(\log n)$	Comparison
[AHR98, Section 2.2]	-	$\Omega(\frac{\log n}{\log\log n})$	-	$\log^{O(1)} n$	RAM
update operations: link					
[AH00, KS08]	$O(n)$	$O(\alpha(n))$	$O(n)$	$O(1)$	RAM
update operations: link, cut					
[ST83, Section 5]	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$	Semigroup
Section 3.4	−	$\Omega(\log n)$	−	$O(\log n)$	Cell Probe
Section 3.4	−	$\Omega(\frac{\log n}{\log\log n})$	−	$(\log n)^{O(1)}$	Cell Probe
Section 3.4	−	$O(\frac{\log n}{\log\log n})$	−	$\Omega(\log^{1+\varepsilon} n)$	Cell Probe
Section 3.4	−	$O(\frac{\log n}{(\log\log n)^2})$	−	$(\log n)^{\Omega(\log\log n)}$	Cell Probe

evert, where it might not be necessary. The difficulty of these two operations will be described in Section 3.4.

Another variant of the dynamic path minima problem is denoted as *incremental trees*, in which the input is a forest and the only available update operation is link. In the offline setting, where a sequence of m queries and updates are given in advance, Tarjan [Tar79a, Section 6] solved the incremental trees problem under the following two restrictions: (1) the input contains rooted trees, thus link adds an edge between a root and a node in two different trees; (2) each query path is between a node and the root of the tree containing that node. Tarjan solved this problem in $O((m+n) \cdot \alpha(m+n,n))$ time using $O(m+n)$ space in the semigroup model. If only the sequence of updates is

known in advance and not the queries, for rooted trees and the same restricted type of queries, every operation can be performed in $O(1)$ time in the RAM model [Har85].

Alstrup and Holm [AH00] considered the incremental trees problem in the online setting for rooted trees. They achieved $O(\alpha(n))$ query time for arbitrary queries, and $O(1)$ amortized update time using $O(n)$ space in the RAM model. Kaplan and Shafrir extended this result to unrooted trees [KS08] within the same complexities.

Another variant of the path minima problem is when only the update operations insert-leaf and delete-leaf are provided. Alstrup and Holm [AH00] also presented a linear space data structure that solves this version of the problem with $O(1)$ query and update time in the RAM model (see also [KS08, DLW09a]). In the comparison model, $O(1)$ query time and $O(\log n)$ amortized update time with $O(n)$ space is achieved via maintaining the Cartesian tree of the input tree by using dynamic trees of Sleator and Tarjan [DLW09a].

Our contributions. We itemize our contributions to the dynamic path minima problem. We study three different variants of the problem, and for each one we obtain the following results.

- Optimal query time data structures that support updating the edge-weights (Section 3.2): For an input tree containing n nodes, we show that the query time $\Theta(\log n/\log\log n)$ can be achieved if we want to update the edge-weights in $\Theta(\log n)$ comparisons or $O(\log n/\log\log n)$ RAM operations. Moreover our data structures support the operations insert, insert-leaf, contract, delete-leaf with the same update time.

- An optimal query time data structure that supports insertions and deletions of leaves (Section 3.3): For an input tree containing n nodes, we show that the path minima queries can be answered with at most $4k - 1$ comparisons under insertions and deletions of leaves with $O(k\alpha_k(n))$ amortized comparisons. This result also holds in the semigroup model. In the word RAM model, we present another approach to achieve $O(1)$ query/update amortized time for this problem.

- Lower bounds for query-update trade-offs (Section 3.4): We prove two lower bounds for the query-update trade-offs of the dynamic path minima problem when the update operations are link and cut. These lower bounds are derived by reductions. The difficulty of maintaining an input tree under updating the edge-weights for path minima queries is also shown by a reduction from the dynamic 1D-RMQ problem.

1.4 Geometric Range Aggregate Queries

In range aggregate query problems for geometric objects, the goal is to preprocess a set of geometric objects, and compute a certain aggregation function that operates on the objects within a query range. Aggregation function are either geometric or non-geometric. For example, for a given set of line segments, queries ask for the number of intersection points of all the segments that are contained within a given range. Non-geometric aggregation functions can be defined when, for example, weights are
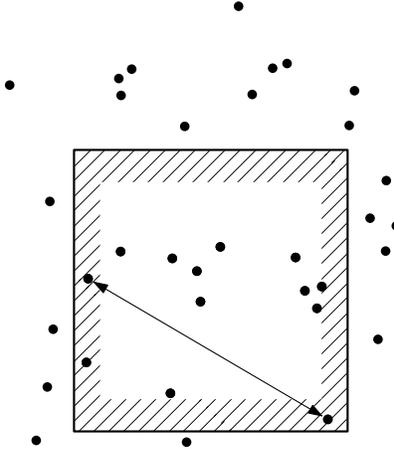
Figure 1.2: The answer to a range diameter query is a pair of points that are farthest away in a given orthogonal range.

associated to input objects. Then aggregation functions, such as minimum or median, operate on the weights of the objects that are within a query range.

A standard technique to solve range aggregate query problems is given an input, divide the input into disjoint partitions, aggregate each part separately and then compute the final result by further aggregation of the partial results. This technique can answer aggregation functions like count and maximum, but cannot support functions like *median*, which cannot be computed by aggregating partial results[5] [Gup05]. In this section, we consider the geometric aggregation function *diameter* which has similar difficulties like median.

1.4.1   Range Diameter Queries

In this section, we consider the two dimensional range diameter problem defined as follows. The goal is to preprocess a set of n points from $\mathbb{R}^2$, such that we can efficiently find the two furthest points within an orthogonal range query (Figure 1.2). We mainly focus on the space efficiency of data structures and not their construction time. In the following, we first describe a simple solution to the range diameter problem using *tabulation*, and then we mention the known results on the problem.

The problem can be naively solved by tabulating the answer of all possible queries, in rank space where each point is placed in a cell of an n by n grid according to the rank of each of its coordinates. This technique gives $O(\log n)$ query time using $O(n^4)$ space as follows. Consider the point set in rank space. Each orthogonal range query is defined by two corners of the range. Thus, the total number of queries is $O(n^4)$. Compute the answer of each query offline by looking at the original point set (not in rank space), and store all the answers in a table in some canonical order. The size of this table is $O(n^4)$ and it can be built in $O(n^6)$ time naively or in $O(n^5 \log n)$ time using diameter algorithms running in $\Theta(n \log n)$ time for a point set of size n [PS91]. The furthest points within each orthogonal range query can be found by first doing a

---

[5]These types of functions are denoted as holistic functions in [Gup05].

predecessor search in rank space in $O(\log n)$ time to find the borders of the range, and then looking the answer up in the table in $O(1)$ time.

Gupta, Janardan, Kumar, and Smid [JGKS08] mentioned that standard geometric range searching structures can be used to decrease the space from $O(n^4)$ to $O(n^2 \log n)$ while answering queries in $O(\log^4 n)$ time. This can be achieved as follows. Construct a two-level range tree structure for the point set [dBCvK08]. For each node u in the secondary structure (for Y-coordinates) of the range tree, let S_u be the set of points in the subtree of u. During the preprocessing, for every pair of nodes u and v in the secondary structure, store the two furthest points of $S_u \cup S_v$ in a table. The size of the table is $O(n^2 \log n)$. The range tree partitions every orthogonal query into $O(\log^2 n)$ subqueries that each corresponds with a node in the secondary structure of the range tree. To answer a query, consider every pair of subqueries of the query. For every pair of subqueries, find the furthest points by looking up in the table in $O(1)$ time. Thus, the query time of the data structure is $O(\log^4 n)$ derived from the number of pairs of the subqueries, and the space usage is $O(n^2 \log n)$.

Gupta, Janardan, Kumar, and Smid [JGKS08] also improved the above solution by giving a time-space trade-off for the problem. They presented a data structure that supports queries in $O(k \log^5 n)$ time using $O((n + (n/k)^2) \log^2 n)$ space, for a parameter k where $1 \leq k \leq n$. To achieve this result, they utilized furthest-point Voronoi diagrams and point location structures [Kir83]. A furthest-point Voronoi diagram of a point set P of size n partitions the plane into at most n parts, where each part corresponds to one point in P. A part D corresponding to a point p has the property that every point in $\mathbb{R}^2$ which is also in D has p as the furthest point among all the points in P.

In the following, we describe the data structure of [JGKS08] that achieves the mentioned trade-off. This data structure is similar to the first data structure of [JGKS08] using range trees that was described above. Following the preprocessing algorithm of the first data structure, for every pair of nodes u and v in the secondary structure, store the two furthest points of $S_u \cup S_v$ in a table, only if the size of both S_u and S_v is at least k, where k is a parameter. For every node u in the secondary structure of the range tree, construct the furthest-point Voronoi diagram of S_u along with a point location structure for its subdivision. To answer a query, consider every pair of subqueries of the query. For every pair of subqueries, if both of them have at least k points, find the furthest points by looking up the table in $O(1)$ time. If at least one of S_u and S_v has less than k points, say S_u, then for each point p in S_u, search for the furthest point to p in S_v using the furthest-point Voronoi diagram constructed for S_v in $O(k \log n)$ time. Thus, the query time of the data structure is $O(k \log^5 n)$ derived from the number of pairs of the subqueries. The size of the table is $O((n/k)^2 \log^2 n)$, and the space used to store all the furthest-point Voronoi diagrams and point location structures is $O(n \log^2 n)$.

Our contributions. We make a step forward to show the difficulty of range diameter queries. We demonstrate relations between the range diameter query problem and fundamental data structure problems suggesting lower bounds for the space usage. We also describe that we can overcome the difficulty of range diameter queries under assumptions for input point sets. Our results are as follows:

- Relation to the set intersection problem (Section 4.2): We prove that supporting range diameter queries for a point set in the plane is as hard as verifying the

disjointness of two sets among a given collection of sets. The latter problem is conjectured to have quadratic space complexity for constant query time. Assuming the conjecture in a stronger model, our reduction suggests the same lower bound for range diameter queries.

- A lower bound for computing the furthest points in two vertically-separated convex polygons (Section 4.3): We prove that for any independent representation of two given convex polygons that are vertically separated, any algorithm needs nearly linear time in the number of vertices of the smallest polygon. This lower bound is derived by a reduction from the asymmetric set disjointness in communication complexity.

- Supporting range diameter queries in logarithmic time (Section 4.4): We show that for input convex polygons of size n, $O(\log n)$ query time can be obtained using only $O(n + m \log m)$ space, where m is the total modality of a convex polygon defined as the sum over all vertices of the number of local maxima in the sequence of distances from each vertex to the other vertices.

1.5   Preliminaries

In this section, we present definitions, algorithms, and data structures that are used in other parts of the dissertation.

1.5.1   Computational Models

Range queries can be considered in different computational models depending on the various constraints that can be defined on the input data elements, memory access methods, and the primitive operations of the machine running the algorithms.

Real RAM.   In computational geometry, the standard model of computation is the real RAM [PS85], where the input values (for example, the coordinates of points) are unbounded real numbers, and the arithmetic operations ($+$, $-$, $\times$, $/$), comparisons, k-th root, trigonometric functions, exponential function (that is, e^x), and the logarithm function are available. In the real RAM, the real numbers are stored in a randomly accessible memory. The geometric range query problems such as the range diameter problem, are usually defined in this model.

Pointer machine.   A more restrictive model denoted as the pointer machine model was proposed by Tarjan [Tar79b] to understand the inherent power of pointer manipulation in terms of considering lower bounds for the complexity of list processing problems. In this model, any kind of memory address calculation is forbidden, and data is accessed by following pointers. Chazelle [Cha88] considered variants of this model to solve a number of problems in multidimensional range searching. Most of the range searching data structures can be described in this model [AE99].

Word RAM. Apart from restricting the memory access methods, we can restrict the data elements to make them more realistic. Instead of working with unbounded real numbers in the real RAM, we can assume that the data elements are integers, or even more constrained, any number or symbol that can be represented in a memory cell of size w bits. The latter one suggests the realistic *word RAM* model in which data elements are integers or floats and can be represented in a machine word. In this model, the operations are standard RAM operations (with or without bitwise operations), and memory cells can be accessed randomly.

The geometric range query problems can be considered in this model. For example the input can be a set of points, where their coordinates are integers of size w bits. The word size of the model usually changes with the problem size. For example, if n is the number of points in the input point set, we assume that $w \geq \log n$ or $w = O(\log n)$ bits.

The word RAM is the typical model used to study the range query problem on arrays. In this model, each element of an array is stored in a memory word of size w bits. The elements of the array can be restricted depending on the application. For example, each element can be a member of a totally ordered set of size at most 2^w. In this case, only the comparison operations can be performed on the elements. In a less constrained model, the input elements of the array can be numbers (w-bit integers or floating points), and any of the arithmetic operations, comparisons or bit-wise operations may be allowed.

Comparison-based model. Although the advanced operations of the word RAM model make it realistic and practical, but we are also interested to consider models that are simple and clean. An example is the comparison-based model. In this model, the data elements are from a partially or totally ordered set, and only comparisons are allowed to be performed on the data elements. Comparison-based algorithms can utilize the power of the pointer machine or RAM to find the memory cells. Standard sorting algorithms such as *quick sort* are examples of comparison-based algorithms.

Decision tree model. A fundamental model from a theoretical perspective, which is intellectually attractive [Hag98], is the decision tree model that is used to prove lower bounds for the complexity of the number of comparisons to solve computational problems (for example, sorting [AHU87]). In this model, the nodes of a decision tree describe the branching operations (comparisons) performed by an algorithm, and each leaf of the tree determines an output of the algorithm.

Semigroup model. Lueker [Lue78] speculated that the decision tree model cannot demonstrate the difficulty of the multidimensional orthogonal range queries, and another computational model should be considered to prove good lower bounds for such a problem. Then, Fredman [Fre81] proposed the semigroup model in which he proved a lower bound for the mentioned problem. A semigroup is a set of elements together with an associative binary operation, for example, the elements could be numbers and the operations be $+$, $*$, or the minimum of two numbers. The range query problem for input arrays can be studied in the semigroup model. In this case, we assume that the elements of the input arrays are from a semigroup, and every algorithm is only allowed to use the semigroup operation on the elements of the array. Indeed, the algorithm is

oblivious to the actual semigroup operation that might be used, but it should give the correct output for all of the semigroup operations.

Cell probe model. Another model that is used to prove strong lower bounds for computational problems is the cell problem model. In this model, computation is free, and we only pay for accessing memory (reads or writes). More precisely, we store data structures in memory cells of size w bits, which is a parameter of the model. The query time is counted as the number of cells probed by a query algorithm. The model allows memory reads or writes to depend arbitrarily on past cell probes. The model is non-uniform, that is, for different values of input parameter n, we can have different algorithms.

1.5.2 Cartesian Trees

Vuillemin [Vui80] introduced *Cartesian trees* in the context of average time analysis of searching. Let A be an array containing n elements from a totally ordered set. The Cartesian tree of A is a binary tree with nodes labeled by the indexes of A. The root has label i, where $A[i]$ is the minimum element in A. The left subtree of the root is a Cartesian tree for the subarray $A[1 \cdots i-1]$, and the right subtree of the root is a Cartesian tree for the subarray $A[i+1 \cdots n]$. This inductive definition implies that the answer to any range minimum query $q = (i, j)$ in A, is the label of the lowest common ancestor (LCA) of the nodes labelled by i and j. Vuillemin also showed how to construct a Cartesian tree in linear time [Vui80].

Now we consider the Cartesian tree of a tree. Let T be a tree (not necessarily binary or rooted) containing n nodes and each edge of T is assigned a weight from a totally ordered set. The Cartesian tree of T is a binary tree, where its root corresponds to the edge e of T with minimum weight, and the two children of the root correspond to the Cartesian trees of the two components made by deleting e from T [Vui80]. The internal nodes of the Cartesian tree are the edges of T, and the leaves of the Cartesian tree are the nodes of T. This definition implies that the answer to any path minima query between two nodes u and v in T, is determined by the LCA of the leaves corresponding to u and v in the Cartesian tree (Figure 1.3) [DLW09a].

It has been proved that an $O(n)$ space data structure to maintain the Cartesian tree of a tree with n nodes, can be constructed in $O(n \log n)$ time and comparisons to support path minima queries in $O(1)$ time ([Net99, Section 3.3.2], [BMN$^+$04, Section 2], and [DLW09a, Theorem 2]). Since the construction of the Cartesian tree of a star tree (that is, a tree where one node is adjacent to all remaining nodes) corresponds to sorting the edges, it follows that an explicit construction of a Cartesian tree will require $\Omega(n \log n)$ comparisons [DLW09a].

Maintaining Cartesian trees under inserting leaves. The above discussion explains how to support path minima queries in a static tree, using the Cartesian tree of the tree. When the tree is dynamic, in the sense that we can insert and delete new leaves in the tree, maintaining the Cartesian tree of the tree under these update operations is of interest. Indeed, the path minima problem for dynamic weighted trees in different computational models is one of the subjects of this dissertations (see Sections 1.3.1 and 3.3 for further details).
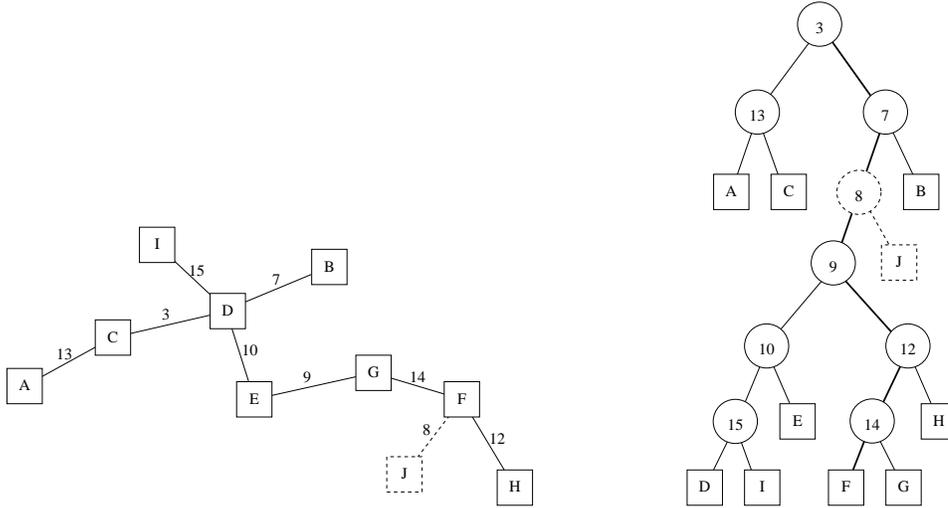
Figure 1.3: A tree T (left) and the Cartesian tree of T (right). Inserting the new leaf J as a neighbor of F with an edge of weight 8 in T causes 8 to be inserted as a node in the Cartesian tree on the path from F to the the root such that heap order is preserved.

Here, we describe an algorithm presented in [DLW09b] that maintains the Cartesian tree of a tree T under inserting new leaves into T. In particular, they show how to maintain a Cartesian tree of T containing n nodes under insertion of a new leaf in T in $O(\log n)$ worst case time (Figure 1.4). Recall that every edge of T is an internal node in the Cartesian tree, and every node (including leaves) of T is a leaf in the Cartesian tree. Let ℓ be a new leaf which we want to insert into T as a new child of v. Therefore, v is a leaf and the edge $e = (\ell, v)$ is a new node in the Cartesian tree. In the Cartesian tree, the edge e should be inserted in the path from v to the root such that the weights along the path remain heap ordered, and ℓ becomes a child of e. Hence, the problem is reduced to finding the appropriate location to insert e into the Cartesian tree.

Demaine, Landau, and Weimann [DLW09b] showed how to use the link-cut trees of Sleator and Tarjan [ST83] to partition a Cartesian tree into disjoint paths such that e can be located and inserted in the Cartesian tree by using a constant number of query, link, and cut operations. The following lemma states their result[6].

Lemma 1.1 ([DLW09a]) *The Cartesian tree of a tree with n nodes can be maintained in a data structure of size $O(n)$ that can be constructed in $O(n \log n)$ time, and supports* pathmin *in $O(1)$ time and* insert-leaf *in $O(\log n)$ time.*

1.5.3   Trees: Transformations and Decompositions

In this section, we describe different aspects of trees that are used in various parts of this dissertation. First, we demonstrate a bijection function from the set of binary

---

[6]Their algorithm only appears in the online manuscript of the paper [DLW09b]. The conference version of the paper states this result without describing how to achieve it [DLW09a].
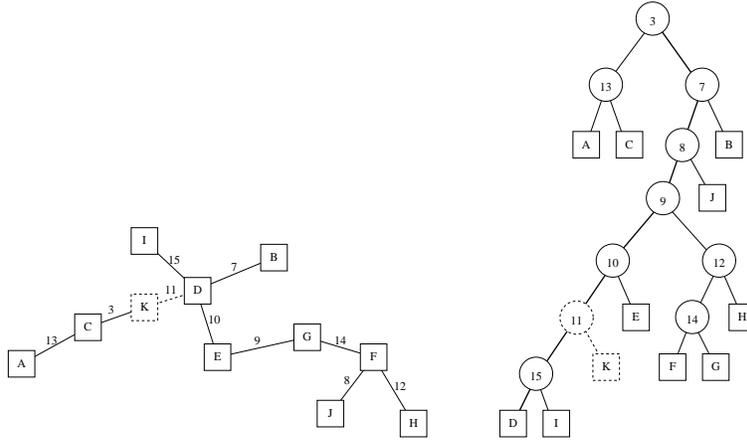
Figure 1.4: In the tree T (left), a new node K is inserted on the edge (C,D) with weight 3. The new edge (K,D) has weight $11 \geq 3$. In the Cartesian tree (right), the new edge with weight 11 is inserted as a node on the path from D to the node LCA(C,D)="3" such that heap order is preserved.

trees to the set of ordinal trees. As mentioned in Section 1.1.1, this one-to-one correspondence is used to show the information theoretic lower bound of representing the ordinal trees. We also explain how we can utilize a representation of ordinal trees to represent binary trees. This can be used in succinct data structures as mentioned in Section 1.1.1. Then, we give another transformation from rooted trees to binary trees that is usually used in data structures that deal with rooted trees. This transformation shows that, for the sake of simplicity, we can build our data structures for binary trees instead of rooted trees while supporting the operations on the rooted trees. We use this transformation for our path minima data structure for rooted trees in Chapter 3. We explain a known decomposition of trees, in which each node has degree at most 3. This decomposition is used to design divide and conquer data structures for trees as we do in Chapter 3. At the end, we describe an algorithm to split a binary tree into constant number of binary trees under satisfying some properties. The split algorithms are usually used to amortize the insertion operation in tree data structures such as ours in Chapter 3.

One-to-one correspondence between ordinal trees and binary trees.   There exists a one-to-one correspondence between ordinal trees (ordered rooted trees) and binary trees. More precisely, every ordinal tree containing n nodes can be transformed into a binary tree containing $n-1$ nodes, and vice versa. This shows that the total number of ordinal trees containing n nodes equals the total number of binary trees containing $n-1$ nodes. In the following, we describe these two transformations.

Let T be an ordinal tree that we want to transform into a binary tree T'. For each node u in T, we make a corresponding node u' in T'. We start from the root of T by putting the corresponding root in T'. Let u_i be the i-th child of an arbitrary node u in the tree. For each node u, u_1 becomes the left child of u', u_2 becomes the right child of u'_1, u_3 becomes the right child of u'_2, and so on. In other words, all the siblings of the

first child of u from left-to-right are placed in a right-skewed path from top-to-bottom as the right subtree of the node corresponding to the first child of u. The result is a binary tree containing n nodes whose root never has a right child. Therefore, it can be removed from the binary tree, and the tree has $n-1$ nodes.

The above transformation is indeed a bijective function. Let T' be an arbitrary binary tree containing $n-1$ nodes. In the following we transform it to an ordinal tree T containing n nodes, which then can be transformed back into T' using the above transformation. For each node u' in T', we make a corresponding node u in T. First add a new root r' to T' such that the previous root becomes the left child of r'. Make a root r for T corresponding to r'. We start from the left child of r'. For each node u' in T', if u' is a left child, u becomes the first child of u''s parent. If u' is a right child of a node v', u becomes the immediate sibling of v to the right.

Representing binary trees with ordinal trees. In the following, we explain how to use the basic navigational operations of an ordinal tree to support the navigational operations in a binary tree.

Every binary tree T can be transformed to an ordinal tree T', such that each navigational operation on T can be performed using a constant number of basic navigational operations on T'. The transformation is as follows. For each node v in T, we make a corresponding node in T' denoted as v'. We make T' by making a dummy root, and putting the root of T as the first child of the dummy root. Then starting from children of the root of T, for every node v in T, if v is the left child of its parent u, we make v' as the first child of u', and if v is the right child of u, we make v' as the next sibling of u'. Therefore, the left child of a given node u in T can be determined by finding the first child of u'. The right child of a given node u in T can be determined by finding the next sibling of u'. The parent of a given node v in T, can be determined by first verifying whether the node is a left child or a right child. This can be done by finding the first child of the parent of v' denoted as u'. If u' is the same as v', then v is a left child and we return the parent of v' as the parent of v. If u' is not the same as v', then v is a right child and we return the previous sibling of v' as the parent of v.

Transforming rooted trees into binary trees. We first describe how to transform a rooted tree into a binary tree, and then we show how to perform rooted trees operations on the binary tree. The transformation is due to [Fre85]. Let T be the rooted tree that we want to transform into a binary tree T'. For every node u in the rooted tree with d children $v_1, v_2, \ldots, v_d$, where $d \geq 2$, we represent u by d nodes $w_1, w_2, \ldots, w_d$ in the binary tree. We make w_{i+1} the right child of w_i with the edge-weight $+\infty$, thus we make a path of length d nodes. Then, we make v_i the left child of w_i with the weight of the edge (u, v_i), and we replace u by w_1 as the left child of the parent of u. The right child of w_d is empty, which acts as a place holder for inserting new children for u.

The operation insert-leaf(u,v,w), which adds the new node v and the new edge (u, v) weighted with w, can be performed by doing insert-leaf(w_d, w_{d+1}, ∞) for the left child of w_d, and insert-leaf(w_{d+1}, v, w) for the right child of w_{d+1}. The operation insert(e, v, w), which splits the edge $e = (u_1, u_2)$ by inserting the node v along it, such that the new edge (u_1, v) has weight w, and (u_2, v) has the old weight of e, can be performed by a single insert operation on the binary tree.

Micro-macro decomposition of a binary tree. We utilize the algorithm presented in [ASS97] to partition a tree. Given a binary tree T with n nodes and a parameter x, where $1 \le x \le n$, we decompose the set of nodes in T into $O(n/x)$ disjoint subsets, each of size at most x, where each subset induces a subtree of T called a *micro tree*. Furthermore, the division is constructed such that at most two nodes in a micro tree are adjacent to nodes in other micro trees that are denote by *boundary nodes*. If a micro tree has two boundary nodes, then one of the nodes is the root. We define a macro tree consisting of all the boundary nodes, such that it contains an edge between two nodes if either they are in the same micro tree or there is an edge between them in T.

Lemma 1.2 (**[ASS97]**) *Given a rooted binary tree T with n nodes and a parameter x, where $1 \le x \le n$. A partitioning of T into micro trees can be performed in $O(n)$ time that satisfies: (1) each micro tree contains at most x nodes, (2) there are $O(n/x)$ micro trees, and (3) each micro tree has at most two boundary nodes.*

Splitting a binary tree. Given a binary tree T with n nodes and at most two boundary nodes, we decompose the set of nodes of T into at most four disjoint subsets inducing four subtrees of T in $O(n)$ time, where each subtree has at most $1 + 2n/3$ nodes, and at most two boundary nodes including the old boundary nodes. Note that T can also be decomposed by using Lemma 1.2, but then the old boundary nodes do not necessarily remain as the boundary nodes of the newly created subtrees. Therefore, we present another algorithm for this decomposition as follows.

We first find a *centroid edge e* of T, i.e., an edge whose removal partitions T into two trees of size at most $1 + 2n/3$ each. It is well-known that for a given non-empty binary tree, such an edge exists and can be found in $O(n)$ time [Cha82]. We remove e from T to obtain two components T_1 and T_2, each of size at most $1 + 2n/3$. We should maintain the property that each component has at most two boundary nodes. Let c_1 and c_2 be the incident nodes of e, existing in T_1 and T_2 respectively. There are three cases based on the location of the boundary node(s) of T: (1) Let b be the only boundary node of T. Assume w.l.o.g. that b is in T_1. Then T_1 is a micro tree with two boundary nodes b and c_1, and T_2 is a micro tree with one boundary node c_2. (2) If nodes b_1 and b_2 are the two boundary nodes of T, and if b_1 is in one component and b_2 is in the other component, then each component has two boundary nodes. (3) If nodes b_1 and b_2 are the two boundary nodes of T, and if w.l.o.g. both of b_1 and b_2 are in T_1. The component T_2 has one boundary node c_2. The component T_1 has three boundary nodes c_1, b_1, and b_2. Then we split T_1 into three components as follows. Let $(c_1, \ldots, b, x_1, \ldots, b_1)$ be the path from c_1 to b_1, and $(c_1, \ldots, b, x_2, \ldots, b_2)$ be the path from c_1 to b_2, where b is the last common node in these two paths. We remove the edges (b, x_1) and (b, x_2) which partition T_1 into three components. Each component has exactly two boundary nodes which are the pairs (b_1, x_1), (b_2, x_2), and (c_1, b).

1.5.4   Q-heap

Fredman and Willard [FW94] presented a dynamic data structure denoted as Q-heap. This data structure for a small enough set of integers, supports the operations search (unsuccessful search returns the predecessor), rank (which returns the number

of elements in the set that are less than the query element), insert, and delete in constant time in the RAM model. The following lemma summarizes this data structure.

Lemma 1.3 ([FW94]) *For any integer $n < 2^b$, and a set of m b-bit integers, where b is the word size of the machine and m is at most $(\log n)^{1/4}$, there exists a data structure of size $O(m)$ supporting insertion, deletion, and rank operations in $O(1)$ worst case time using a lookup table of size $O(n)$ constructed in $O(n)$ time.*

1.5.5 Inverse-Ackermann Function

The inverse-Ackermann function denoted by α has been defined variously [AS87, DLW09a, Tar79a]. In this dissertation, the definitions that are provided here are used which are based on the definitions in [DLW09a, Niv09]. First, we define a sequence of functions denoted as the inverse-Ackermann hierarchy for integers $n \geq 1$:

$$
\begin{aligned}
\alpha_0(n) &= \lceil n/2 \rceil \\
\alpha_k(1) &= 0 && \text{for } k \geq 1 \\
\alpha_k(n) &= 1 + \alpha_k(\alpha_{k-1}(n)) && \text{for } k \geq 1 .
\end{aligned}
$$

In other words,

$$
\begin{aligned}
\alpha_0(n) &= \lceil n/2 \rceil \\
\alpha_k^{(1)}(n) &= \alpha_k(n) && \text{for } k \geq 0 \\
\alpha_k^{(j)}(n) &= \alpha_k(\alpha_k^{(j-1)}(n)) && \text{for } k \geq 0, j \geq 2 \\
\alpha_k(n) &= \min\{j \mid \alpha_{k-1}^{(j)}(n) \leq 1\} && \text{for } k \geq 1 .
\end{aligned}
$$

Note that

$$
\begin{aligned}
\alpha_1(n) &= \lceil \log_2 n \rceil \\
\alpha_2(n) &= \log^* n \\
\alpha_3(n) &= \log^{**} n .
\end{aligned}
$$

Indeed, $\alpha_k(n) = \log^{**\cdots*} n$, where the $*$ is repeated $k-1$ times in the superscript for $k \geq 2$. The inverse-Ackermann function is defined as:

$$
\alpha(n) = \min\{k \mid \alpha_k(n) \leq 3\} .
$$

The two-parameter version of the inverse-Ackermann function for integers $m, n \geq 1$ is defined as follows:

$$
\alpha(m,n) = \min\{k : \alpha_k(n) \leq 3 + m/n\} .
$$

This definition of the function satisfies $\alpha(m,n) \leq \alpha(n)$ for every m and n. Further details can be found in [Niv09, Sei06].

Chapter 2

Range Minimum Queries

Abstract. The range minimum query (RMQ) problem for multidimensional arrays is considered in this chapter. This is a problem that asks "where is the minimum in an orthogonal range within an array?". The problem is studied for input arrays with different dimensionality. We address some questions regarding the RMQ problem such as: how fast can we answer a query with a compressed indexing data structure? and how many bits do we require to encode a two dimensional input array?

For input arrays (in any dimensions) containing N elements, we prove that if an indexing data structure stores $O(N/c)$ bits, it requires $\Omega(c)$ time to support RMQs, for any c where $1 \leq c \leq N$, in the cell probe model. For one-dimensional input arrays, we show that this lower bound is tight up to a constant factor. For two-dimensional input arrays, we present an RMQ indexing data structure supporting queries in $O(c \log^2 c)$ time using additional space $O(N/c)$ bits. For $O(1)$ query time, this gives an optimal space indexing data structure, and it leaves a gap for super constant query time.

We also prove that to encode an m by n input array, where $m \leq n$, $\Omega(mn \log m)$ bits is required to support RMQs. This information-theoretic lower bound is smaller than the trivial upper bound $O(mn \log n)$ bits. We also present an encoding data structure that supports RMQs in $O(1)$ time using $O(mn \cdot \min\{m, \log n\})$ bits.

We also give optimal query time data structures that can maintain one dimensional input arrays containing n elements under updating their entries. These data structures achieve $\Theta(\log n / \log \log n)$ query time, and obtains $\Theta(\log n)$ update time in the comparison model and $O(\log n / \log \log n)$ update time in the RAM model.

2.1   Introduction

In this chapter, we consider the range minimum query (RMQ) problem for d-dimensional arrays. We preprocess an array $A[1 \cdots n_1] \times [1 \cdots n_2] \times \cdots \times [1 \cdots n_d]$ containing $N = n_1 \cdot n_2 \cdots n_d$ elements from a totally ordered set, into an RMQ data structure. An RMQ data structure supports queries asking for the *position* of the minimum element within a given orthogonal d-dimensional range query $q = (i_1, j_1, i_2, j_2, \ldots, i_d, j_d)$ that spans all the cells in $A[i_1 \cdots j_1] \times [i_2 \cdots j_2] \times \cdots \times [i_d \cdots j_d]$. Without loss of generality, we assume that all the entries of A are distinct (identical entries of A are ordered lexicographically by their index). In this chapter, a matrix denotes a two dimensional array. We also consider the RMQ problem for one dimensional arrays when updating the entries of the arrays is also possible.

2.1.1   Our Contributions

We study RMQ indexing data structures for input arrays of any dimension. We prove a lower bound for the trade-off between the number of bits required to store an indexing data structure and its query time. In particular, for an input array containing N elements, we prove that $\Omega(N/c)$ bits is required to store an indexing data structure supporting RMQs in $O(c)$ time, for any c where $1 \le c \le N$. This lower bound is proved in a non-uniform *cell probe* model, and the proof is similar to the proof of Theorem 3.1 of Golynski [Gol07]. This result appears in Theorem 2.1 of Section 2.2.1.

For the 1D-RMQ problem, where the input is a one dimensional array, we present a simple indexing data structure of size $O(N/c)$ bits that supports 1D-RMQs in $O(c)$ time. This implies that the lower bound proved in Theorem 2.1 of Section 2.2.1 is tight for the 1D-RMQ problem within a constant factor. Notice that this data structure brings the space below the information-theoretic lower bound $2N + o(N)$ bits which is for encoding 1D-RMQ data structures. We mention this upper bound in Section 2.2.2.

For the 2D-RMQ problem, where the input is a two dimensional array containing N elements, we present an indexing data structure of size $O(N/c)$ bits that supports RMQs in $O(c \log^2 c)$ time. This trade-off gives the optimal additional space bound $O(N)$ bits for $O(1)$ query time. For super constant query times, this leaves a gap to the additional space of RMQ data structures. To present this data structure, we first show how to achieve $O(1)$ query time using additional space $O(N)$ bits in Section 2.2.3. Then, using this data structure, we build a data structure that achieves the claimed trade-off in Section 2.2.4.

We also consider the issue of encoding data structures for the 2D-RMQ problem. In particular, we prove an information-theoretic lower bound $\Omega(mn \cdot \log m)$ bits to encode an m by n input array such that the encoding information can be used to answer 2D-RMQs. Notice that the input arrays can be stored explicitly using $O(mn \log n)$ bits. Thus, we introduce a gap to the encoding space. Recall that Demaine, Landau, and Weimann [DLW09a] proved that $\Omega(nn \cdot \log n)$ bits is required when the input is an n by n array. Indeed, our lower bound is an extension of theirs with a similar argument. We also show that $O(mn \cdot \min(m, \log n))$ bits is sufficient to support 2D-RMQs in $O(1)$ time. Section 2.3 is devoted to this issue.

We finish this chapter by giving two data structures that achieve $\Theta(\log n / \log \log n)$ query time for the 1D-RMQ problem under updating the entries of input arrays

Table 2.1: Our contribution to the *d*-dimensional RMQ problem for arrays containing *N* elements. The parameter *c* is an integer, where $1 \leq c \leq N$. The lower bounds should be read like a conditional sentence, e.g., the result of Theorem 2.1 states that if the additional space is $O(N/c)$ bits, then $\Omega(c)$ query time is required. For the results of the last two rows, the input is an *m* by *n* array, where $m \leq n$.

Reference	dimension	Query time	Space (bits)	Prep. time
Theorem 2.1	$d \geq 1$	$\Omega(c)$	$O(N/c)$	-
Theorem 2.2	$d = 1$	$O(c)$	$O(N/c)$	$O(N)$
Theorem 2.3	$d = 2$	$O(1)$	$O(N)$	$O(N)$
Theorem 2.4	$d = 2$	$O(c \log^2 c)$	$O(N/c)$	$O(N)$
Theorem 2.5	$d = 2$	-	$\Omega(mn \cdot \log m)$	-
Section 2.3.1	$d = 2$	$O(1)$	$O(mn \cdot \min(m, \log n))$	$O(N)$

Table 2.2: Our contribution to the dynamic 1D-RMQ problem for arrays containing *n* elements.

Ref.	Preprocessing time	Query time	Space	Update time	Model
Theorem 2.6	$O(n)$	$O(\frac{\log n}{\log \log n})$	$O(n)$	$O(\log n)$	Comparison
Theorem 2.7	$O(n)$	$O(\frac{\log n}{\log \log n})$	$O(n)$	$O(\frac{\log n}{\log \log n})$	RAM

containing *n* elements (Section 2.4). The first data structure supports the updates in $\Theta(\log n / \log \log n)$ time in the comparison model (Theorem 2.6), and the second data structure obtains $O(\log n)$ update time in the RAM model (Theorem 2.7).

2.2 Indexing Data Structures

In this section, we study RMQ indexing data structures. First, we prove a lower bound for any dimensional input arrays. Then we show that the lower bound is tight for the 1D-RMQ problem. We also present an indexing data structure that supports 2D-RMQs in constant time using optimal space. At the end, we give a 2D-RMQ indexing data structure that achieves a time-space trade-off using the constant query time data structure.

2.2.1 Lower Bound

We prove a lower bound for the query time of the 1D-RMQ problem where the input is a one dimensional array of *n* elements, and then we show that the bound also holds for the RMQ problem in any dimension. The proof is in the non-uniform cell probe model [Mil]. In this model, computation is free, and time is counted as the number of cells accessed (probed) by the query algorithm. The algorithm is also allowed to be non-uniform, i.e., for different values of input parameter *n*, we can have different
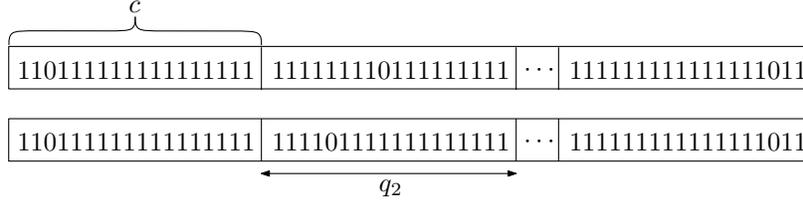
Figure 2.1: Two arrays from $\mathscr{C}$, each one has n/c blocks. In this example $c = 18$. The query q_2 has different answers for these arrays.

algorithms.

For integers n and c, where $1 \leq c \leq n$, we define a set of arrays $\mathscr{C}$, and a set of queries $\mathscr{Q}$. W.l.o.g., we assume that c divides n. We will argue that for any 1D-RMQ algorithm which has access to an index of size n/c bits (in addition to the input array A), there exists an array in $\mathscr{C}$ and a query in $\mathscr{Q}$ for which the algorithm performs $\Omega(c)$ probes into A.

Definition 2.1 *Let n and c be two integers, where $1 \leq c \leq n$ and c divides n. The set $\mathscr{C}$ contains the arrays $A[1 \cdots n]$ such that the elements of A are from the set $\{0,1\}$, and in each block $A[(i-1)c+1 \cdots ic]$ for all $1 \leq i \leq n/c$, there is exactly a single zero element (Figure 2.1).*

The number of possible data structures of size n/c bits is $2^{n/c}$, and the number of arrays in $\mathscr{C}$ is $c^{n/c}$. By the pigeonhole principle, for any algorithm $\mathscr{G}$ there exists a data structure $D_{\mathscr{G}}$ which is shared by at least $(\frac{c}{2})^{n/c}$ input arrays in $\mathscr{C}$. Let $\mathscr{C}_{D_{\mathscr{G}}} \subseteq \mathscr{C}$ be the set of these inputs.

Definition 2.2 *Let $q_i = [(i-1)c+1 \cdots ic]$. The set $\mathscr{Q} = \{q_i \mid 1 \leq i \leq n/c\}$ contains n/c queries, each covering a distinct block of A.*

For algorithm $\mathscr{G}$ and data structure $D_{\mathscr{G}}$, we define a binary decision tree capturing the behavior of $\mathscr{G}$ on the inputs from $\mathscr{C}_{D_{\mathscr{G}}}$ to answer a query $q \in \mathscr{Q}$.

Definition 2.3 *Let $\mathscr{G}$ be a deterministic algorithm. For each query $q \in \mathscr{Q}$, we define a binary decision tree $T_q(D_{\mathscr{G}})$. Each internal node of $T_q(D_{\mathscr{G}})$ represents a probe into a cell of the input arrays from $\mathscr{C}_{D_{\mathscr{G}}}$. The left and right edges correspond to the output of the probe: left for reading a zero and right for reading a one. Each leaf is labelled with the answer to q, i.e., the position of the zero within the block covered by q.*

For each algorithm $\mathscr{G}$, we have defined n/c binary trees depicting the probes of the algorithm into the inputs from $\mathscr{C}_{D_{\mathscr{G}}}$ to answer the n/c queries in $\mathscr{Q}$. Note that the answers to all these n/c queries uniquely determine the input. We compose all the n/c binary trees into a single binary tree $T_{\mathscr{Q}}(D_{\mathscr{G}})$ in which every leaf determines a particular input. To obtain $T_{\mathscr{Q}}(D_{\mathscr{G}})$, we first replace each leaf of $T_{q_1}(D_{\mathscr{G}})$ with the whole $T_{q_2}(D_{\mathscr{G}})$, and then replace each leaf of the obtained tree with $T_{q_3}(D_{\mathscr{G}})$, and so on (Figure 2.2). Every leaf of $T_{\mathscr{Q}}(D_{\mathscr{G}})$ is labelled with the answers to all the n/c queries in $\mathscr{Q}$ which were replaced on the path from the root to the leaf. Every two input arrays in $\mathscr{C}_{D_{\mathscr{G}}}$ correspond to different leaves of $T_{\mathscr{Q}}(D_{\mathscr{G}})$. Otherwise the answers to all the
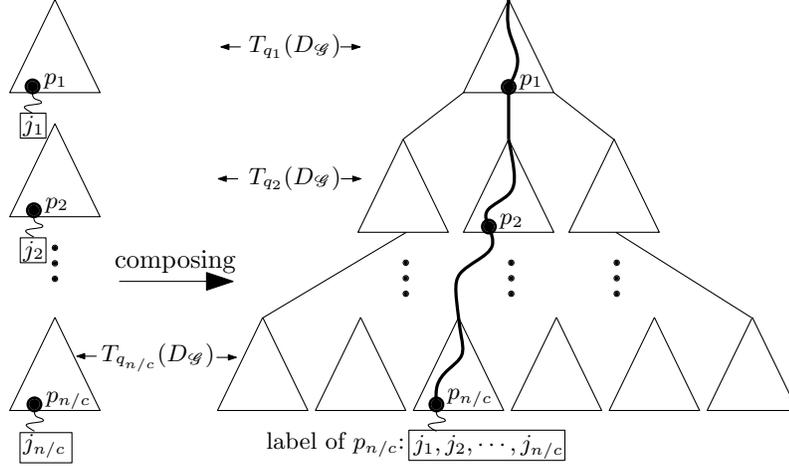
Figure 2.2: Composing the n/c decision trees to obtain the large decision tree $T_{\mathcal{Q}}(D_{\mathcal{G}})$. Each leaf is labeled with a vector of positions of zeros in the input.

queries in $\mathcal{Q}$ are the same for both the inputs which is a contradiction. Therefore, the number of leaves of $T_{\mathcal{Q}}(D_{\mathcal{G}})$ is at least $(\frac{c}{2})^{n/c}$, the minimum number of inputs in $\mathcal{C}_{D_{\mathcal{G}}}$.

We next prune $T_{\mathcal{Q}}(D_{\mathcal{G}})$ as follows: First we remove all nodes not reachable by any input from $\mathcal{C}_{D_{\mathcal{G}}}$. Then we repeatedly replace all nodes of degree one with their single child. Since the inputs from $\mathcal{C}_{D_{\mathcal{G}}}$ correspond to only reachable leaves, the number of leaves becomes equal to the number of inputs from $\mathcal{C}_{D_{\mathcal{G}}}$ which is at least $(\frac{c}{2})^{n/c}$. Note that the result of a *repeated* probe is known already, because the probe has been performed before. Therefore, before pruning, one child of the node corresponding to a repeated probe is unreachable, and after pruning where all the unreachable nodes are pruned, there is no repeated probe on a root to leaf path. Every path from the root to a leaf has at most n/c left edges (zero probes), since the number of zero elements in each input from $\mathcal{C}$ is n/c. Each of these paths represents a binary sequence of length at most d containing at most n/c zeros, where d is the depth of $T_{\mathcal{Q}}(D_{\mathcal{G}})$ after pruning. By padding each of these sequences with further 0s and 1s, we can ensure that each sequence has length exactly $d + n/c$ and contains exactly n/c zeros. The number of such binary sequences is $\binom{d+n/c}{n/c}$, which becomes an upper bound for the number of leaves in the tree after pruning.

Lemma 2.1 *For all n and c, where $1 \le c \le n$, the worst case number of probes required to answer a query in $\mathcal{Q}$ over the inputs from $\mathcal{C}$ using a data structure of size n/c bits is $\Omega(c)$.*

Proof. First, we prove a lower bound for d, the depth of $T_{\mathcal{Q}}(D_{\mathcal{G}})$ after pruning. Then, we divide the lower bound by n/c, the number of binary trees, to prove the lower bound for the number of probes.

In the above discussion, we obtained the following upper bound for the number of leaves of $T_{\mathcal{Q}}(D_{\mathcal{G}})$ after pruning.

$$\binom{d + \frac{n}{c}}{\frac{n}{c}} = \frac{(d + \frac{n}{c})!}{(\frac{n}{c})! \cdot (d + \frac{n}{c} - \frac{n}{c})!} \le \frac{(d + \frac{n}{c})^{(\frac{n}{c})}}{(\frac{n}{c})!} \ .$$

Comparing this upper bound with the lower bound for the number of leaves of $T_{\mathscr{Q}}(D_{\mathscr{G}})$, we have

$$\left(\frac{c}{2}\right)^{n/c} \leq \frac{(d+\frac{n}{c})^{\left(\frac{n}{c}\right)}}{\left(\frac{n}{c}\right)!} \ .$$

By Stirling's formula, we obtain the following:

$$\frac{c}{2} \leq \frac{(d+\frac{n}{c})e}{\frac{n}{c}} \ ,$$

and therefore $d \geq n(\frac{1}{2e} - \frac{1}{c})$. For any arbitrary algorithm $\mathscr{G}$, the depth d of $T_{\mathscr{Q}}(D_{\mathscr{G}})$ is at most the sum of the depths of the n/c binary trees composed into $T_{\mathscr{Q}}(D_{\mathscr{G}})$. By the pigeonhole principle, there exists an input $x \in \mathscr{C}_{D_{\mathscr{G}}}$ and an i, where $1 \leq i \leq n/c$, such that the query q_i on x requires at least $d/(n/c) = \Omega(c)$ probes into the array maintaining the input.   □

Theorem 2.1 *Any algorithm that uses N/c bits additional space to solve the RMQ problem for an input array of size N (in any dimension), requires $\Omega(c)$ query time, for any c, where $1 \leq c \leq N$.*

Proof. Lemma 2.1 gives the lower bound for the 1D-RMQ problem. The proof for the 2D-RMQ is a simple extension of the proof of Lemma 2.1. The set $\mathscr{C}$ consists of matrices, each composed of mn/c submatrices $[ic_1 + 1 \cdots (i+1)c_1] \times [jc_2 + 1 \cdots (j + 1)c_2]$ of size c_1 by c_2, for $1 \leq i < m/c_1$ and $1 \leq j < n/c_2$, where $c = c_1 \cdot c_2$ (w.l.o.g., assuming that c_1 divides m, and c_2 divides n). Each submatrix has exactly one zero element, and all the others are one. There are N/c queries in $\mathscr{Q}$, each one asks for the minimum of each submatrix. As in the proof of Lemma 2.1, we can argue that there exists a query requiring $\Omega(c)$ probes by utilizing the methods of decision trees, composing and pruning them, and bounding the number of leaves. The proof can be generalized straightforwardly to higher dimensional versions of the RMQ problem. □

2.2.2   Tightness of the Lower Bound in One Dimension

We show that the lower bound proved in Theorem 2.1 is tight for the 1D-RMQ problem.

Theorem 2.2 *The 1D-RMQ problem for a one dimensional input array of size n is solved in $O(n)$ preprocessing time and optimal $O(c)$ query time using $O(n/c)$ additional bits.*

Proof. Partition the input array into n/c blocks of size c. Construct a 1D-RMQ encoding structure $\mathscr{D}$ for the list of n/c block minima (minimum elements of the blocks) in $O(n/c)$ bits [Sad07b]. The query is decomposed into three subqueries q_ℓ, q_m, and q_r (see Figure 2.3). The subquery q_m contains all the blocks fully spanned by the query. To solve q_m, we first find the block containing the answer by querying the data structure $\mathscr{D}$ in $O(1)$ time, and then scan that block in $O(c)$ time to find the answer. Each of the subqueries q_ℓ and q_r, which is contained within a single block, is answered in $O(c)$ time by scanning the respective block.   □
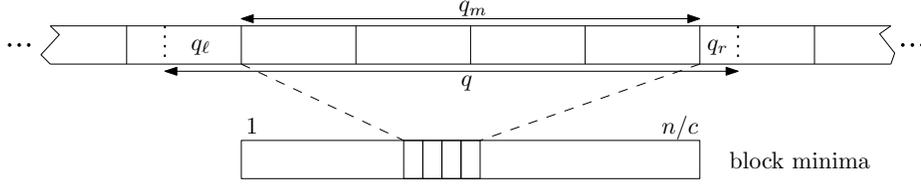
Figure 2.3: The input is partitioned into n/c blocks of size c. The 1D-RMQ encoding structure $\mathscr{D}$ of size $O(n/c)$ bits is built for the list of the block minima. The query q is divided into three subqueries q_ℓ, q_m, and q_r.

2.2.3 Constant Query Time with Optimal Space in Two Dimensions

We present an indexing data structure that supports 2D-RMQs in $O(1)$ time using additional space $O(N)$ bits. This structure forms the basis of another data structure in Section 2.2.4 to achieve a time-space trade-off for the 2D-RMQ indexing data structures.

Preliminaries. We introduce some terminology that we use to describe an indexing data structure for the 2D-RMQ problem in the following sections. A *block* is a rectangular range in a matrix. Let B be a block of size m' by n'. For the block B, the list $\mathrm{MinColList}[1 \cdots n']$ contains the minimum element of each column of B and $\mathrm{MinRowList}[1 \cdots m']$ contains the minimum element of each row of B. For integer ℓ where $1 \le \ell \le m'/2$, let $\mathrm{TopSuffixes}(B, \ell)$ be the set of blocks $B[m'/2 - i\ell + 1 \cdots m'/2] \times [1 \cdots n']$, and $\mathrm{BottomPrefixes}(B, \ell)$ be the set of blocks $B[m'/2 + 1 \cdots m' - (i-1)\ell] \times [1 \cdots n']$, for $1 \le i \le m'/(2\ell)$ (w.l.o.g., assuming that 2ℓ divides m').

Data structure and querying. In the following, we present an indexing data structure of size $O(N)$ bits achieving $O(1)$ query time to solve the 2D-RMQ problem for an m by n input matrix M of size $N = m \cdot n$. The basic idea of the construction is to solve the problem with four levels of recursion, reducing the queries to subqueries of size $\log\log m$ by $\log\log n$, which are solved by a tabulation idea of Atallah and Yuan [AY10]. We partition the input matrix M into $m/\log m$ blocks $\mathscr{B} = \{b_1, \ldots, b_{m/\log m}\}$ of size $\log m$ by n by cutting the input matrix at every $\log m$'th row. If a query is contained in a block b_i, the problem is solved recursively for this block. Otherwise, the query q is divided into subqueries q_1, q_2 and q_3 such that q_1 is contained in b_j and q_3 is contained in b_k, and q_2 spans over $b_{j+1}, \ldots, b_{k-1}$ vertically, where $1 \le j < k \le m/\log m$ (see Figure 2.4). Since q_1 and q_3 are range minimum queries in the submatrices b_j and b_k respectively, they are answered recursively. The subquery q_2 is handled as described below. Lastly, the answers to q_1, q_2 and q_3, which are indices into three matrix elements, are used to find the index of the smallest element in q.

A binary tree structure is utilized to answer q_2. This binary tree has $m/\log m$ leaves, one for each block in $\mathscr{B}$. Without loss of generality, we assume that $m/\log m$ is a power of 2. Each leaf maintains a 1D-RMQ structure [Sad07b] for MinColList of its corresponding block b_i. Each internal node v with $2k$ leaf descendants corresponds to a submatrix M composed of $2k$ consecutive blocks of $\mathscr{B}$, for $1 \le k \le m/(2\log m)$.
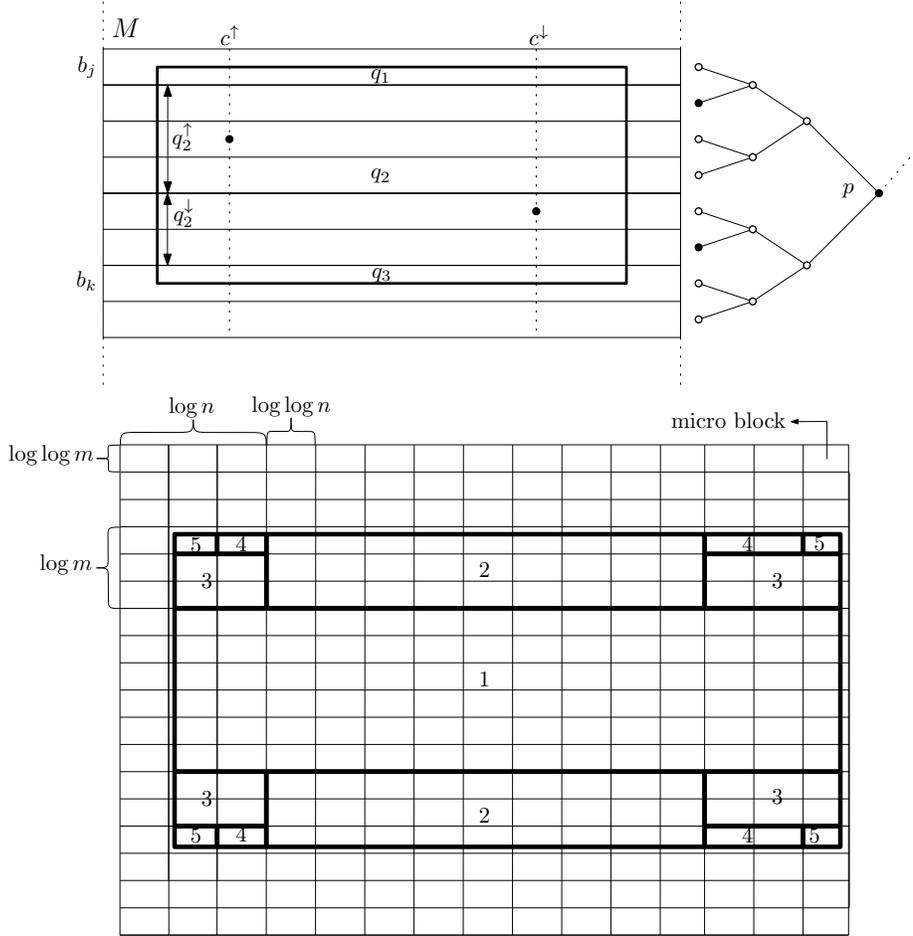
Figure 2.4: Top: Partitioning the input and building the binary tree structure. The node p is the LCA of the leaves corresponding to b_{j+1} and b_{k-1}. The columns $c^\uparrow$ and $c^\downarrow$, which contain the answers to $q_2^\uparrow$ and $q_2^\downarrow$ respectively, are found using the 1D-RMQ structure stored in p. The minimum element in each of the columns $c^\uparrow$ and $c^\downarrow$ is found using the 1D-RMQ structure constructed for that column. Bottom: The numbers 1,2,3,4, and 5 on the subqueries depict the recursion level that answer the corresponding subqueries.

These $2k$ blocks correspond to the $2k$ leaf descendants of v. Note that each of the sets TopSuffixes(M, log m) and BottomPrefixes(M, log m) contains k blocks. For each of these $2k$ blocks, the internal node v stores a 1D-RMQ structure that is constructed for the MinColList of the block.

We also construct a 1D-RMQ structure for each of the rows and columns of the input matrix M.

In the binary tree structure, let p be the lowest common ancestor of the leaves corresponding to b_{j+1} and b_{k-1}, and let M be the submatrix corresponding to p. The subquery q_2 is composed of the top part $q_2^\uparrow$ and the bottom part $q_2^\downarrow$, where $q_2^\uparrow$ and $q_2^\downarrow$ are two blocks in the sets TopSuffixes(M, log m) and BottomPrefixes(M, log m), respectively. Two of the 1D-RMQ structures maintained in p, are constructed for MinColLists of $q_2^\uparrow$

and $q_2^\downarrow$. These 1D-RMQ structures are utilized to find two columns $c^\uparrow$ and $c^\downarrow$ containing the answer to $q_2^\uparrow$ and $q_2^\downarrow$. The 1D-RMQ structures constructed for these two columns are utilized to find the answer to $q_2^\uparrow$ and $q_2^\downarrow$. Then the answer to q_2 is determined by comparing the smallest element in $q_2^\uparrow$ and $q_2^\downarrow$.

In the second level of the recursion, each block of $\mathscr{B}$ is partitioned into blocks of size $\log m$ by $\log n$. The recursion continues for two more levels until the size of each block is $\log\log m$ by $\log\log n$. In the binary tree structures built for all the four recursion levels, we construct the 1D-RMQ structures for the appropriate MinColLists and MinRowLists respectively. The blocks that are used to make MinRowLists are defined similarly to TopSuffixes and BottomPrefixes, but for left suffixes and right prefixes respectively. In the second and fourth levels of recursion, where the binary tree structure gives two rows containing the minimum elements of $q_2^\uparrow$ and $q_2^\downarrow$, the 1D-RMQ structures constructed for the rows of the matrix are used to answer $q_2^\uparrow$ and $q_2^\downarrow$. Similar to the first level of the recursion, in the third level, where the binary tree structure gives two columns containing the minimum elements of $q_2^\uparrow$ and $q_2^\downarrow$, the 1D-RMQ structures constructed for the columns of the matrix are used to answer $q_2^\uparrow$ and $q_2^\downarrow$.

We solve the 2D-RMQ problem for a block of size $\log\log m$ by $\log\log n$ using the table lookup method given by Atallah and Yuan [AY10]. Their method preprocesses the block by making at most $c'G$ comparisons, for a constant c', where $G = \log\log m \cdot \log\log n$, such that any 2D-RMQ can be answered by performing four probes into the block. Each block is represented by a *block type* which is a binary sequence of length $c'G$, encoding the results of the comparisons. The lookup table has $2^{c'G}$ rows, one for each possible block type, and G^2 columns, one for each possible query within a block. Each cell of the table contains four indices to address the four probes into the block. The block types of all the blocks of size G in the matrix are stored in another table T. The query within a block is answered by first recognizing the block type using T, and then checking the lookup table to obtain the four indices. Comparing the results of these four probes gives the answer to the query. For further details, we refer the reader to [AY10].

Theorem 2.3 *The 2D-RMQ problem for an m by n matrix of size $N = m \cdot n$ is solved in $O(N)$ preprocessing time and $O(1)$ query time using $O(N)$ bits additional space.*

Proof. We first consider the query time. The subquery q_2 is answered in $O(1)$ time by using a constant query time LCA structure [HT84], querying the 1D-RMQ structures in constant time [Sad07b], and performing $O(1)$ probes into the input matrix. The number of recursion levels is four, and for each level, we perform at most four recursive subqueries (see Figure 2.4). In the last level, the subqueries contained in blocks of size G are also answered in $O(1)$ time by using the lookup table and performing $O(1)$ probes into the matrix. Therefore the query q is answered in total $O(1)$ time.

We bound the space of the data structure as follows. The depth of the binary tree, in the first recursion level, is $O(\log(m/\log m))$. Each level of the tree has $O(m/\log m)$ 1D-RMQ structures for MinColLists of size n elements. Since a 1D-RMQ structure of a list of n elements is stored in $O(n)$ bits [Sad07b], the binary tree can be stored in $O(n \cdot m/\log m \cdot \log(m/\log m)) = O(N)$ bits. Since the number of recursion levels is $O(1)$, the binary trees in all the recursion levels are stored in $O(N)$ bits. The space used by the $m+n$ 1D-RMQ structures constructed for the columns and rows of M is $O(N)$ bits.

Since $G = o(\log N)$, then $G \leq c'' \log N$ for any constant $c'' > 0$, and sufficiently large N. We can therefore bound the size of the lookup table by $O(2^{c'c'' \log N} G^2 \log G) = o(N)$ bits when $c'' < 1/c'$. The size of table T is $O(N/G \cdot \log(2^{c'G})) = O(N)$ bits. Hence the total additional space is $O(N)$ bits.

Finally, we consider the preprocessing time. In the binary tree, in the first level of the recursion, each leaf maintains a 1D-RMQ structure constructed for a MinColList of size n elements. These $m/\log m$ lists are constructed in $O(N)$ time by scanning the whole matrix. Each MinColList in the internal nodes is constructed by comparing the elements of two MinColLists built in the lower level in $O(n)$ time. Therefore constructing these lists, for the whole tree, takes $O(N + n \cdot m/\log m \cdot \log(m/\log m)) = O(N)$ time. Since a 1D-RMQ structure can be constructed in linear time [Sad07b], the 1D-RMQ structures in all the nodes of the binary tree are constructed in total $O(N)$ time. The LCA structure is also constructed in linear time [HT84]. Therefore the binary tree is built in $O(N)$ time. Since the number of recursion levels is $O(1)$, all the binary trees are built in $O(N)$ time. The lookup table and table T are also constructed in $O(N)$ time, see [AY10, Sections 3.2 and 5]. $\qquad\square$

Corollary 2.1 *The query algorithm performs at most 38 probes into the input to solve the query.*

Proof. As shown at the top of Figure 2.4, the subquery q_2 is answered by comparing the smallest elements in $q_2^{\uparrow}$ and $q_2^{\downarrow}$. To find these two smallest elements, the algorithm performs two probes into the input. For each of the subqueries solved in different recursion levels, shown at the bottom of Figure 2.4, at most two probes are performed. As described earlier, to solve the subqueries contained in blocks of size $\log\log m$ by $\log\log n$, four probes are performed. Therefore, the total number of probes in the recursion levels is the sum: $2 + 2 \cdot 2 + 4 \cdot 2 + 4 \cdot 2 + 4 \cdot 4 = 38$. $\qquad\square$

2.2.4   Time-Space Trade-off in Two Dimensions

We now describe how to use the data structure of Section 2.2.3 to achieve a trade-off between the additional space usage and the query time. We present an indexing data structure of size $O(N/c \cdot \log c)$ bits additional space solving the 2D-RMQ problem in $O(c \log c)$ query time and $O(N)$ preprocessing time, where $1 \leq c \leq N$. The input matrix is divided into N/c blocks of size 2^i by $c/2^i$, for each integer i in the range $[0 \cdots \log c]$; w.l.o.g., assuming that c is a power of 2. Let M_i be the matrix of size N/c containing the minimum elements of the blocks of size 2^i by $c/2^i$. Let D_i be the linear space data structure of Section 2.2.3 applied to the matrix M_i using $O(N/c)$ bits. Each D_i handles a different ratio between the number of rows and the number of columns of the blocks. Note that the matrices M_i are constructed temporarily during the preprocessing and are not maintained in the data structure.

A query q is resolved by answering $\log c + 1$ subqueries. Let q_i be the maximal subquery of q spanning blocks of size 2^i by $c/2^i$ for $0 \leq i \leq \log c$. The minimum elements of the blocks spanned by q_i assemble a query over M_i which has the same answer as q_i. Therefore, q_i is answered by using D_i. Note that whenever the algorithm wants to perform a probe into a cell of M_i, a corresponding block of size c of the input is
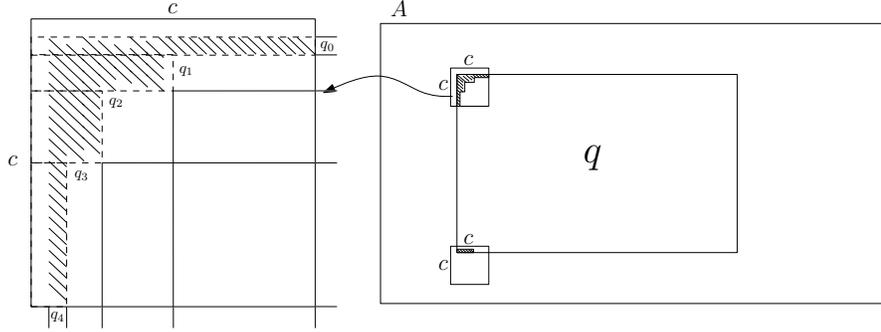
Figure 2.5: Right: The white area of the query q contains the subqueries which completely span the blocks of size 2^i by $c/2^i$. Left: A corner of q which is contained in a block of size c by c. The shaded area contains $O(c \log c)$ elements.

searched for the minimum (since M_i is not explicitly stored in the data structure). The subqueries q_i overlap each other. Altogether, they compose q except for $O(c \log c)$ elements in each of the four corners of q (see the proof of Theorem 2.4). We search these corners for the minimum element. Eventually, we compare the minimum elements of all the subqueries to find the answer to q (see Figure 2.5).

Theorem 2.4 *The 2D-RMQ problem for a matrix of size N is solved in $O(N)$ preprocessing time and $O(c \log^2 c)$ query time using $O(N/c)$ bits additional space.*

Proof. The number of linear space data structures D_i is $\log c + 1$. Each data structure D_i requires $O(N/c)$ bits. Therefore, the total additional space is $O(\log c \cdot N/c)$ bits.

The number of subqueries q_i is $\log c + 1$. Each q_i is answered by using D_i in $O(1)$ query time in addition to the $O(1)$ probes into M_i. Since each probe into M_i can be performed by $O(c)$ probes into the input matrix, the query q_i can be answered in $O(c)$ time. Each of the four corners of the query q not covered by the q_i queries, is contained in the union of at most $\log c + 1$ blocks, at most one block of each size 2^i by $c/2^i$ for $0 \le i \le \log c$ (see Figure 2.5). The four corners are searched in $O(c \log c)$ time for the minimum element. In the end, the minimum elements of the subqueries are compared in $O(\log c)$ time to answer q. Consequently, the total query time is $O(c \log c)$.

Each D_i is constructed in $O(N/c)$ time (Section 2.2.3) after building the matrix M_i. To be able to make all M_i efficiently, we first construct an $O(N)$-bit space data structure of Section 2.2.3 for the input matrix in $O(N)$ time. Then, M_i is built in $O(N/c)$ time by querying a block of the input matrix in $O(1)$ time for each element of M_i. Therefore, the total preprocessing time is $O(\log c \cdot N/c + N) = O(N)$. Substituting the parameter c by $c \log c$ gives the claimed bounds. $\square$

2.3 Encoding Data Structures in Two Dimensions

We examine the RMQ encoding data structures for m by n input arrays. Without loss of generality, we assume that $m \le n$. First we present a simple data structure of

size $O(mn \cdot \min(m, \log n))$ bits that supports 2D-RMQs in $O(1)$ time without consulting the input. Then, we prove the information-theoretic lower bound $\Omega(mn \log m)$ bits for the 2D-RMQ problem. This leaves a gap for the space requirement of such data structures.

2.3.1   Upper Bound

The algorithm described in Section 2.2.3 can preprocess an m by n input matrix A of size $N = m \cdot n$ into a data structure of size $O(N)$ bits in $O(N)$ time. But the query algorithm in Section 2.2.3 is required to perform some probes into the input matrix. Since A is not accessible in the encoding model, we store another matrix maintaining the rank of all the N elements using $O(N \log N) = O(N \log n)$ bits. Whenever the algorithm wants to perform a probe into A, it does it into the rank matrix. Therefore the problem can be solved in the encoding model using $O(N \log n)$ preprocessing time (to sort A) and $O(1)$ query time using space $O(N \log n)$ bits.

Another solution in the encoding model is the following. For each of the n columns of A, we build a 1D-RMQ structure using space $O(m)$ bits [Sad07b], in total using $O(mn) = O(N)$ bits. Furthermore, for each possible pair of rows (i_1, i_2), $i_1 \le i_2$, we construct a 1D-RMQ structure for the MinColList L_{i_1,i_2} of $A[i_1 \cdots i_2] \times [1 \cdots n]$, i.e., $L_{i_1,i_2}[j] = \min_{i_1 \le i \le i_2} A[i,j]$, using space $O(n)$ bits. Note that we only store the 1D-RMQ structure for L_{i_1,i_2}, but not L_{i_1,i_2} itself. In total we use space $O(m^2 n) = O(Nm)$ bits. The column j containing the answer to a query $q = [i_1 \cdots i_2] \times [j_1 \cdots j_2]$ is found by querying for the range $[j_1 \cdots j_2]$ in the 1D-RMQ structure for L_{i_1,i_2}. The query q is answered by querying for the range $[i_1 \cdots i_2]$ in the 1D-RMQ structure for column j. Since both 1D-RMQ queries take $O(1)$ time, the total query time is $O(1)$.

Selecting the most space efficient solution of the above two solutions gives an encoding structure of size $O(N \cdot \min\{m, \log n\})$ bits with $O(1)$ query time.

2.3.2   Lower Bound

To prove a lower bound for the space required in the encoding model, we generate a large class of input matrices which are distinguishable by the queries. We consider two matrices A_1 and A_2 *different* if there exists a 2D-RMQ with different answer for A_1 and A_2. We present a set of $\Omega((m!)^n)$ matrices which are pairwise different. The elements of the matrices are from the set $\{1, \ldots, mn\}$. In every matrix A of the set, the smallest mn' elements of A are placed in two parts $A' = A[1 \cdots m/2] \times [1 \cdots n']$ and A'' containing all the anti-diagonals of length $m/2$ within the block $A[m/2+1 \cdots m] \times [n' + 1 \cdots n]$ where $n' = \lfloor (n - m/2 + 1)/2 \rfloor$, w.l.o.g., assuming that m is even (see Figure 2.6). The odd numbers from the set $\{1, \ldots, mn'\}$ are placed in A' in increasing order from left to right and then top to bottom, i.e. $A'[i,j] = 2((i-1)n' + j) - 1$. The even numbers of $\{1, \ldots, mn'\}$ are placed in A'' such that the elements of each anti-diagonal are not sorted but are larger than the elements of the anti-diagonals to the right. The total number of matrices constructed by permuting the $m/2$ elements of each of the n' anti-diagonals of A'' is $(\frac{m}{2}!)^{n'}$.

For any two matrices A_1 and A_2 in the set, there exists an index $[i_2, j_2]$ in the anti-diagonals of A'' such that $A_1[i_2, j_2] \ne A_2[i_2, j_2]$. Without loss of generality, assume that $A_1[i_2, j_2] < A_2[i_2, j_2]$. Let $[i_1, j_1]$ be the index of an arbitrary odd number in A' be-
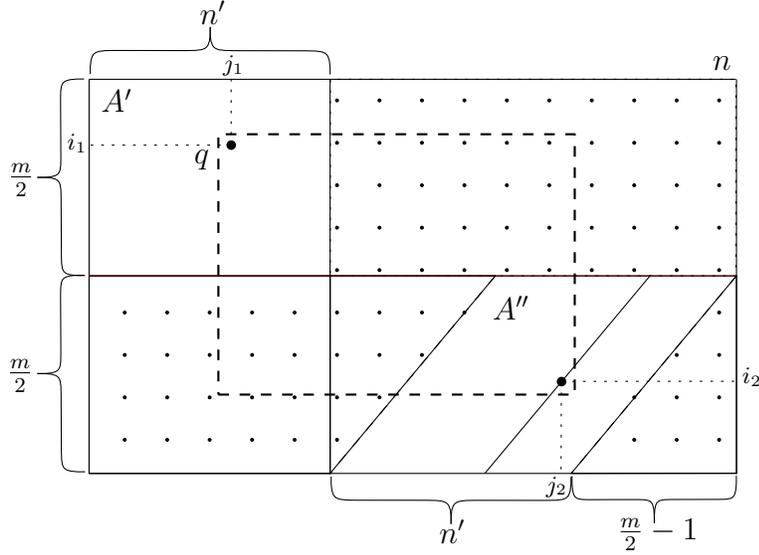
Figure 2.6: All the elements in the dotted area are greater than the elements in the white area. The rectangle drawn with dashed line shows the query q. The smallest entry in $A' \cap q$ is $A[i_1, j_1]$ and the smallest entry in $A'' \cap q$ is $A[i_2, j_2]$.

tween $A_1[i_2, j_2]$ and $A_2[i_2, j_2]$. The query $q = [i_1 \cdots i_2] \times [j_1 \cdots j_2]$ has different answers for A_1 and A_2: For A_1 the answer is $[i_2, j_2]$ whereas the answer for A_2 is $[i_1, j_1]$ (see Figure 2.6). It follows that any two matrices in the set are different.

Theorem 2.5 *The minimum space required to store an encoding data structure for the 2D-RMQ problem is $\Omega(mn \log m)$ bits, assuming that $m \leq n$.*

Proof. Since the number of different matrices in the set is $(\frac{m}{2}!)^{n'}$, the space for a data structure encoding these matrices is $\Omega(\log(\frac{m}{2}!)^{n'}) = \Omega(mn' \log m) = \Omega(mn \log m)$ bits, since $n' \geq (n - \frac{m}{2})/2 \geq n/4$, where the last inequality follows from $m \leq n$. $\quad\square$

2.4 Dynamic Structures in One Dimension

We state our result for the dynamic version of the 1D-RMQ problem, where the elements of input arrays can be updated. This problem is a special case of a variant of the dynamic path minima problem, in which we have to maintain a weighted tree under updating the edge-weights while supporting queries that ask for the edge with minimum weight along a given path. Further details about the path minima problem and its variants can be found in Section 1.3.1 and Chapter 3. Obviously, any data structure that solves this variant of the dynamic path minima problem can also solve the dynamic 1D-RMQ problem for an array A by transforming A to a path, where the i-th edge on the path obtains the weight with the same value as $A[i]$. Therefore, our data structures presented in Section 3.2 can also solve the dynamic 1D-RMQ problem. As was shown in Section 1.2.1, the dynamic 1D-RMQ problem has known lower bounds in different computational models. These lower bounds imply that our path minima dynamic

data structures achieve optimal query time for the dynamic 1D-RMQ problem in the comparison model and the the word RAM. We state our results in the following two theorems which are immediate consequences of Theorems 3.1 and 3.2 respectively.

Theorem 2.6 *In the comparison model, there exists a linear space data structure that supports RMQ queries over one dimensional arrays containing n elements in* $\Theta(\log n / \log \log n)$ *query time, and obtains* $\Theta(\log n)$ *amortized update time.*

Theorem 2.7 *In the RAM model, there exists a linear space data structure that supports RMQ queries over one dimensional arrays containing n elements in* $\Theta(\log n / \log \log n)$ *query time, and obtains* $O(\log n / \log \log n)$ *amortized update time.*

2.5   Open Problems

The lower bound of Theorem 2.1 closed the issue of indexing data structures for the 1D-RMQ problem. In other words, for an input array containing N elements, if we want to support 1D-RMQs in $O(c)$ time, for any parameter c where $1 \leq c \leq N$, we know that the best possible space bound $\Theta(N/c)$ bits for the size of an indexing data structure is achieved by the data structure of Section 2.2.2.

But for the 2D-RMQ problem, there is still an interesting question asking about the best possible query time using an indexing data structure of size $O(N/c)$ bits. The current best upper bound gives $O(c \log^2 c)$ (Theorem 2.4), and the current best lower bound implies $\Omega(c)$ (Theorem 2.1). The answer might be somewhere in between, say $\Theta(c \log c)$.

Another appealing question is: how much can we compress an encoding of an m by n input array, where $m \leq n$, such that we can still answer 2D-RMQs using the encoding? The best lower bound proved in Theorem 2.5 implies $\Omega(mn \cdot \log m)$ bits, while it is far from the best upper bound $O(mn \cdot \min(m, \log n))$ bits (Theorem 2.3.1) for small values of m, such as $m = O(\log n)$.

The whole subject of indexing and encoding data structures for the multidimensional RMQ problem has still many open problems. For an input array containing N elements, the best space bound is $O(N)$ words due to the indexing data structure of [AY10] which supports RMQs in $O(1)$ time. The lower bound that we proved in Theorem 2.1, leaves a gap for the size of the constant query time RMQ indexing data structures.

Chapter 3

Path Minima Queries

Abstract. In this chapter, we study the path minima query problem. We are interested in finding an edge with minimum weight along a given path of a tree with weighted edges. We address some questions such as: how fast can we answer a path minima query if we allow updating the edge-weights? how fast can we insert and delete leaves in a tree while answering path minima queries? and does the dynamic trees of Sleator and Tarjan give the best possible query time for the path minima query problem for input dynamic forests?

For input trees containing n nodes, we present a comparison-based data structure that supports path minima queries in $\Theta(\log n / \log \log n)$ time under updating the edge-weights performed in $\Theta(\log n)$ time. When the edge-weights are integers in the word RAM, we improve the update time to $O(\log n / \log \log n)$. Both of these data structures support inserting a node on an edge, inserting a leaf, and contracting edges.

We also show that if we are not interested in updating the edge-weights, and we only want to insert and delete leaves, it is significantly easier to support path minima queries in the semigroup model, for which we have to compute the product of the edges-weights within a given query path, where the edge-weights are from a semigroup and the product is the semigroup operation. We present a data structure that supports these queries in $\Theta(\alpha(n))$ time with $O(1)$ insertion and deletion amortized time. We also show that this data structure can be extended such that queries can be supported using $2k$ semigroup operations and insertion and deletion of leaves can be performed in $O(k\alpha_k(n))$ semigroup operations, for a parameter k, where $k \geq 1$.

For collections of trees that can be updated by linking the trees and cutting the edges, we prove lower bounds for the query time of the path minima problem through reductions. These lower bounds provide different trade-offs between the query time and the update time of the path minima problem. A consequence of these lower bounds is that the dynamic trees of Sleator and Tarjan do not achieve the best possible path minima query time with polylogarithmic time for link and cut.

3.1 Introduction

In this chapter, we consider variants of the dynamic path minima problem for input weighted trees. We preprocess a collection of input trees containing totally n nodes, where each edge is associated with a weight from a totally ordered set. Our data structures should support pathmin queries asking for the edge with minimum weight along a query path in a tree. Different data structures can also support various subsets of the update operations: update, insert, insert-leaf, contract, delete-leaf, link, and cut (see Section 1.3.1 for the definition of these operations).

We consider the following three variants of the dynamic path minima problem:

- An input consists of a tree, and the operations pathmin, update, insert, insert-leaf, and contract should be supported. We can support the operations on unrooted trees by choosing an arbitrary node as the root, designing data structures for rooted trees. Moreover, we can design data structures for binary trees due to existence of the transformation from rooted trees to binary trees (Section 1.5.3).

- An input consists of a tree, and the operations pathmin, insert-leaf, and delete-leaf should be supported. The above discussion about unrooted trees, rooted trees, and binary trees also apply to this variant of the problem.

- An input consists of a collection of trees, and the operations pathmin, link, and cut should be supported.

We distinguish between three types of algorithms in path minima data structures, depending on how algorithms perform computation on the edge-weights. In other words, we study the dynamic path minima problem in three different models:

- The comparison model: The only allowed operations on the edge-weights are comparisons.

- The word RAM model: Any standard RAM operations are allowed on the edge-weights.

- The semigroup model: The edge-weights are from a semigroup. In this model, pathmin queries ask for the product of the edge-weights on a query path, where the product is the operator of the semigroup.

Except for computations on the edge-weights, our algorithms are in the unit-cost RAM model with word size $\Theta(\log n)$ bits.

3.1.1 Our Contributions

We present dynamic data structures for two variants of the dynamic path minima problem. In Section 3.2.1, for an input tree containing n nodes, we demonstrate that in the comparison model, we can support path minima queries in $\Theta(\log n / \log \log n)$ time while performing the operations update, insert, insert-leaf, and contract in $\Theta(\log n)$ amortized time. We also improve this data structure in the word RAM model such that it can support the same update operations in $O(\log n / \log \log n)$ time (Section 3.2.2).

The optimality of the query times and the comparison-based update time are proved in Section 3.4.

In Section 3.3, we show that the query time and the update time of the data structures of Section 3.2, can be improved if we only allow the operations insert-leaf and delete-leaf. This will be shown through dynamizing the path minima data structure given in [AS87]. This shows that in the semigroup model, path minima queries can be supported in $\Theta(\alpha(n))$ semigroup operations while updating the data structure by inserting and deleting leaves takes $O(1)$ amortized semigroup operations (Section 3.4). We also give another data structure that achieves a trade-off between $4k-1$ query time and $O(nk\alpha_k(n))$ update time for the same update operations, for a parameter k, where $k \geq 1$ (Section 3.3). We also show another approach to obtain $O(1)$ time for all the operations in the RAM model, which was already obtained in [AH00, KS08] (see Section 3.3.3).

In Section 3.4, for an input collection of trees totally containing n nodes, we prove that if we want to support link and cut in polylogarithmic time, we cannot hope for answering path minima queries in faster than $\Omega(\log n / \log\log n)$ time in the cell probe model. We also show that for logarithmic time for link and cut, the $\Theta(\log n)$ query time achieved by the dynamic trees of Sleator and Tarjan is the best possible. Furthermore, we prove that with sub-logarithmic query time, obtaining logarithmic time for link and cut is impossible.

3.2 Data Structures for Dynamic Weights

We present two data structures for the dynamic path minima problem for an input binary tree containing n nodes. These structures support the operations pathmin, update, insert, insert-leaf, and contract. The first data structure is in the comparison model and achieves $\Theta(\log n / \log\log n)$ query time, $\Theta(\log n)$ time for update, and $O(\log n)$ amortized time for insert, insert-leaf, and contract. The second data structure is in the RAM model and achieves $O(\log n / \log\log n)$ for all the above operations by utilizing Q-heaps [FW94] (Section 1.5.4). Both of the structures are similar to the ones in [KS08]. In the following, we first describe the comparison based structure, and then we explain how to convert it to the RAM structure.

3.2.1 In the Comparison Model

We present a dynamic path minima data structure in the comparison model. First, we describe how to preprocess an input binary tree to a data structure, and then we demonstrate how to support the operations. In the preprocessing algorithm, we make a recursive micro-macro decomposition of input trees (Section 1.5.3). Within each micro tree, we precompute the answer of all possible queries. The topology structure of each micro tree and its edge-weights determine a *type* for the micro tree. We also maintain the edge-weights of each micro tree in a searchable data structure under insertions and deletions of new edge-weights to facilitate updating the type of a micro tree after performing an update operation on it. Path minima queries are supported recursively.

Let n be the size of an input tree. By choosing the size of each micro tree to be $O(\log^{\varepsilon} n)$ in the decomposition, for small enough ε, we can make linear-sized

Table 3.1: Our contribution to the dynamic path minima problem. The table is divided into three parts corresponding to three variants of the problem according to the supported update operations that are mentioned at the top of each part. For the first two parts, the input is a tree containing n nodes. For the last part (with the update operations link and cut), the input is a collection of trees totally containing n nodes. The lower bounds should be read like a conditional sentence. For example, the last row of the table states that if the query time is $O(\log n/(\log\log n)^2)$, then $(\log n)^{\Omega(\log\log n)}$ update time is required. For the first row of the table, k is an arbitrary parameter, where $1 \le k \le \alpha(n)$.

Reference	Preprocessing time	Query time	Space	Update time	Model
update operations: insert-leaf, delete-leaf					
Theorem 3.3	$O(nk\alpha_k(n))$	$4k$	$O(nk\alpha_k(n))$	$O(k\alpha_k(n))$	Semigroup
Theorem 3.4	$O(n)$	$\Theta(\alpha(n))$	$O(n)$	$O(1)$	Semigroup
Theorem 3.5 [AH00, KS08]	$O(n)$	$O(1)$	$O(n)$	$O(1)$	RAM
update operations: update, insert, insert-leaf, contract					
Theorem 3.1	$O(n)$	$\Theta(\frac{\log n}{\log\log n})$	$O(n)$	$\Theta(\log n)$	Comparison
Theorem 3.2	$O(n)$	$\Theta(\frac{\log n}{\log\log n})$	$O(n)$	$O(\frac{\log n}{\log\log n})$	RAM
update operations: link, cut					
Section 3.4	–	$\Omega(\log n)$	–	$O(\log n)$	Cell Probe
Section 3.4	–	$\Omega(\frac{\log n}{\log\log n})$	–	$(\log n)^{O(1)}$	Cell Probe
Section 3.4	–	$O(\frac{\log n}{\log\log n})$	–	$\Omega(\log^{1+\varepsilon} n)$	Cell Probe
Section 3.4	–	$O(\frac{\log n}{(\log\log n)^2})$	–	$(\log n)^{\Omega(\log\log n)}$	Cell Probe

lookup tables and we obtain $O(\log n/\log\log n)$ levels of decomposition. Then queries and updates can be supported in $O(\log n/\log\log n)$ levels. At each level, a query takes constant time, and an update takes $O(\log\log n)$ time due to searching for an edge-weight in a micro tree. Therefore, queries take $O(\log n/\log\log n)$ time and updates take $O(\log n)$ time. Amortization for updates is also used to overcome exceeding the limit of the size of micro trees after insertions and deletions. In the following, we describe the details of our decomposition algorithm, the data structure, the lookup tables, and the query and update algorithms.

Decomposition. Let T be the input binary tree. We decompose T into micro trees using Lemma 1.2, such that each micro tree has size $O(\log^{\varepsilon} n)$ and at most two boundary nodes. Each micro tree is contracted to a super-node. A new tree T_1' of size $O(n/\log^{\varepsilon} n)$ is built containing these super-nodes. For each path between the root of a micro tree and the root of its parent micro tree, we put a super-edge between the corresponding super-nodes in T_1'. The weight of this super-edge in T_1' is the minimum weight along the corresponding path (Figure 3.1). We let T_1 be a binarized version of T_1' (Sec-
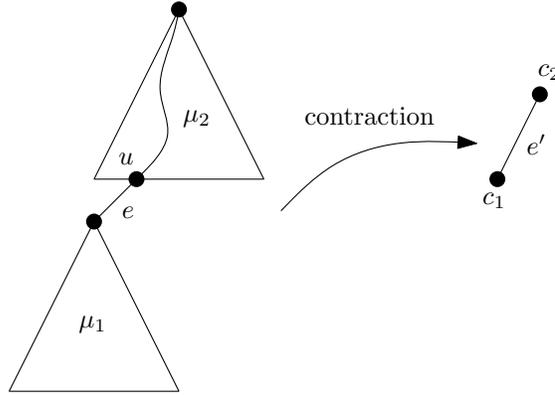
Figure 3.1: The micro trees μ_1 and μ_2 are contracted to the nodes c_1 and c_2 respectively. The weight of the edge e' is calculated as follows: $w(e') = \min\{w(e), w(\mathrm{pathmin}(u, r(\mu_2)))\}$.

tion 1.5.3). The decomposition continues recursively on T_1. In level i, the tree T_{i-1} is decomposed, and the tree T_i is built, for $i = 1, \ldots, \ell$, where T_0 denotes T and ℓ is the number of recursive levels. The size of the micro trees in all the levels and also the size of T_ℓ is $O(\log^\varepsilon n)$, for some constant ε, where $0 < \varepsilon < 1$. The number of recursive levels, ℓ, is $O(\log n / \log \log n)$.

Data structure. The data structure consists of the following parts:

- We explicitly store all the trees $T_0, \ldots, T_\ell$.

- For each node in T_i, we store a pointer to the micro tree of T_i containing that node, and the local ID (insertion time) in the micro tree.

- We represent each micro tree μ with the tuple $(s_\mu, p_\mu, r_\mu, |\mu|)$ of size $o(\log n)$ bits, where s_μ, p_μ, and r_μ are arrays defined as follows. The array s_μ is the binary encoding of the topology of μ. The array p_μ maintains the local IDs of the nodes within μ, and enables us to find a given node inside μ. The array r_μ maintains the rank of the edge-weights according to the preorder traversal of μ.

- For each micro tree, we store a balanced binary search tree containing all the weights of the micro tree. This allows us to find the rank of a new weight within the micro tree under insertion in $O(\log(\log^\varepsilon n))$ time.

- For each micro tree μ of T_i, we store an array of pointers that point to the original nodes in T_i given the local IDs.

Precomputed tables. We use lookup tables to perform each of the following operations within a micro tree μ: pathmin, update, insert, insert-leaf, contract, LCA, root and child-ancestor, where root returns the local ID of the root of μ; LCA returns the local ID of the lowest common ancestor of two given nodes in μ; and child-ancestor(u, v) returns the local ID of the child of u that is also an ancestor of v (if such

a child does not exist, returns null). Tables are indexed by the micro tree representation $(s_\mu, p_\mu, r_\mu, |\mu|)$ and the arguments of the corresponding operation. To perform update, insert, and insert-leaf within μ, we find the rank of the new weight among the edge-weights of μ using its balanced binary search tree in $O(\log|\mu|) = O(\log\log n)$ time. This rank becomes an index for the corresponding tables. The following lemma shows that the operations can be supported using the tables of size $o(n)$ bits.

Lemma 3.1 *Within a micro tree of size $O(\log^\varepsilon n)$, we can support* pathmin*,* LCA*, root, child-ancestor, and moving a subtree inside the tree in $O(1)$ time. The operations* update*,* insert*,* insert-leaf*, and* contract *can be supported in $O(\log\log n)$ time using the balanced binary search tree of the micro tree and precomputed tables of total size $o(n)$ bits that can be constructed in $o(n)$ time.*

Proof. Let μ be the micro tree. The size of the lookup table used to perform pathmin is analyzed as follows. Each entry of the table is a pointer to an edge of μ which can be stored using $O(\log\log n)$ bits. The index to the table consists of (i) $(s_\mu, p_\mu, r_\mu, |\mu|)$, and (ii) two indexes in the range $[1 \cdots |\mu|]$ which represent two pointers to query nodes. The number of different arrays s_μ is $2^{|\mu|}$. The number of different arrays p_μ and r_μ is $O(|\mu|!)$. Therefore, the table is stored in $O(2^{|\mu|} \cdot (|\mu|!) \cdot (|\mu|^3) \cdot (\log|\mu|)) = o(n)$ bits.

In the lookup table used for update-weight, each entry is an array r_μ which maintains the rank of the edge-weights of μ after updating a weight. The index to the table consists of (i) $(s_\mu, p_\mu, r_\mu, |\mu|)$, (ii) an index in the range $[1 \cdots |\mu|]$ to an edge to be updated, and (iii) the rank of the new weight. Therefore, the table can be stored in $O(2^{|\mu|} \cdot (|\mu|!) \cdot (|\mu|^4) \cdot (\log|\mu|)) = o(n)$ bits.

In the lookup table used for add-leaf, each entry is a four-tuple $(s_\mu, p_\mu, r_\mu, |\mu|)$ which maintains the representation of μ after adding the new leaf. The index to the table consists of (i) $(s_\mu, p_\mu, r_\mu, |\mu|)$, (ii) an index in the range $[1 \cdots |\mu|]$ to a vertex adjacent to the new edge, and (iii) the rank of the new weight. Therefore, the table can be stored in $O(2^{|\mu|} \cdot (|\mu|!) \cdot (|\mu|^4) \cdot (\log|\mu|)) = o(n)$ bits.

The size of the other two tables used for LCA and child-ancestor is analyzed similarly. Since the total number of entries in all the tables is less than $o(2^{|\mu|^2})$ and each entry can be computed in time $O(|\mu|)$, all the tables can be constructed in $o(n)$ time.
□

Supporting queries. The query pathmin(u,v) can be answered using the lookup tables, if u and v are in the same micro tree in T. When u and v are not in the same micro tree, we divide the query into subqueries according to our recursive decomposition as follows. Let c be the LCA of u and v in T. There are three micro trees in T that each one contains one of u, v, and c. Each of these three micro trees contains a subquery that can be answered using the lookup tables. In the next level, we consider three micro trees of T_1, each one contains a super-node corresponding to one of the three micro trees that we considered in the previous level. Then, we solve the remaining parts of the query that are within these three micro trees. This query algorithm continues for $k \leq \ell$ levels, until the two micro trees containing u and v are in the same micro tree (Figure 3.2). In our implementation, we first compute the LCA node of each level
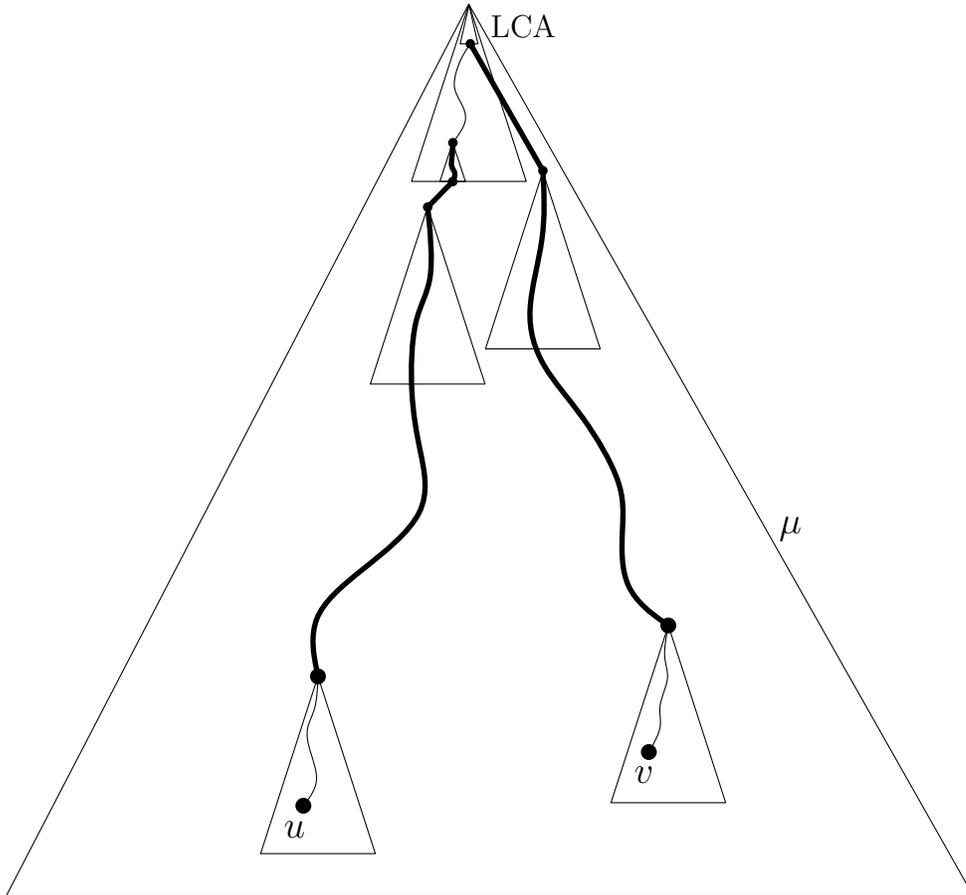
Figure 3.2: Decomposition of the path minima query between the nodes u and v. The micro tree μ in some level of the decomposition contains u, v, and the LCA of u and v. Every thick path is a subquery which is answered by precomputation within μ. Every thin path is a subquery which is answered recursively.

when we return from the previous level. In this way, we can avoid to construct an LCA structure for each T_i. In each level, the three subqueries within the micro trees can be answered in $O(1)$ time using Lemma 3.1. Thus, we achieve $O(\log n / \log \log n)$ query time.

Updating the edge-weights. We perform $\mathsf{update}(e, w)$ by updating the data structure in all the ℓ levels. Without loss of generality, assume that $e = (u, v)$, where u is the parent of v. Let μ be the micro tree in T_0 that contains v. We start to update from the first level, where the tree is T: (1) Update the weight of e in T. (2) If v is not the root of μ, then we update μ using Lemma 3.1. If v is the root of μ, i.e., e connects μ to its parent micro tree, we do not need to update any micro tree. (3) Perform $\mathsf{check\text{-}update}(\mu)$ which recursively updates the edge-weights in T_1 between μ and its child micro trees as follows. We check if $\mathsf{pathmin}$ along the path between the root of μ and the root of each child micro tree of μ needs to be updated. We can check this using $\mathsf{pathmin}$ within μ. If this is the case, for each one, we go to the next level

and perform the three-step procedure on T_1 recursively. Since each micro tree has at most one boundary node that is not the root, then at most one of the child micro trees of μ can propagate the update to the next level, and therefore the number of updates does not grow exponentially. Step 2 takes $O(\log\log n)$ time, and thus update takes totally $O(\log n)$ time in the worst case.

Insertion. We perform insert(e, v, w) using a three-step procedure similar to update. Let μ be the micro tree in T that contains u_2, where $e = (u_1, u_2)$ and u_1 is the parent of u_2. We start from the first level, where the tree is T: (1) To handle insert in the transformed binary tree, we first insert v along e in μ. Note that if u_2 is the root of μ, then v is inserted as the new root of μ. This can be done in $O(\log\log n)$ time using Lemma 3.1. (2) If $|\mu|$ exceeds the maximum limit $3\log^\varepsilon n$, then we split μ into $k \leq 4$ new micro trees, each of size at most $2\log^\varepsilon n + 1$ (see Section 1.5.3). These k micro trees are contracted to nodes that should be in T_1. One of the new micro trees that contains the root of μ corresponds to the node that is already in T_1 for μ. The other $k-1$ new micro trees are contracted and inserted into T_1 with appropriate edge-weights, using insert recursively. Let μ' be the new micro tree that contains the boundary node of μ which is not the root of μ. We perform check-update(μ') to recursively update the edge weights in T_1 between μ' and its child micro trees. (3) Otherwise, i.e., if $|\mu|$ does not exceed the maximum limit, we do check-update(μ) to recursively update the edge weights in T_1 between μ and its child micro trees, which takes $O(\log n)$ time.

To perform insert-leaf(u, v, w), we use the algorithm of insert with the following changes. In step (1), we insert v as a child of u. This can be done in $O(\log\log n)$ time. The step (3) is not required.

A sequence of n insertions into T_0, can at most create $O(n/\log^\varepsilon n)$ micro trees (since any created micro tree needs at least $\log^\varepsilon n$) node insertions before it splits again). Since the number of nodes in $T_0, T_1, \ldots, T_\ell$ is geometrically decreasing, the total number of micro tree splits is $O(n/\log^\varepsilon n)$. Because each micro tree split takes $O(\log^\varepsilon n)$ time, the amortized time per insertion is $O(1)$ for handling micro splits. Thus, both insert and insert-leaf can be performed in $O(\log n)$ amortized time.

Edge contraction. We perform contract(e) by marking v as contracted and updating the weight of e to ∞ by performing update. When the number of marked edges exceeds half of all the edges, we build the whole structure from scratch using insert-leaf for the nodes that are not marked and the edges that do not have weight of ∞. Thus, the amortized deletion time is the same as insertion time.

Theorem 3.1 *There exists a dynamic path minima data structure for an input tree of n nodes in the comparison model, supporting* pathmin *in $O(\log n/\log\log n)$ time,* update *in $O(\log n)$ time,* insert, insert-leaf, *and* contract *in $O(\log n/\log\log n)$ amortized time using $O(n)$ space.*

3.2.2   In the RAM Model

We present a dynamic path minima data structure in the RAM model. Indeed, we exploit the RAM model for improving the time to support updating the edge-weights

to $O(\log n / \log \log n)$ from $O(\log n)$ in the data structure of Theorem 3.1. The bottleneck in our comparison based data structure is that we maintain a balanced binary search tree for the edge-weights within a micro tree. This search tree is used to find the rank of a weight among the weights that are currently in the micro tree. The search for a weight in this search tree takes $O(\log \log n)$ time. Instead of this search tree, in the RAM model, we can maintain the edge weights of the micro tree in a Q-heap [FW94] to find the rank of a weight in $O(1)$ time (Lemma 1.3). We obtain the following data structure.

Theorem 3.2 *There exists a dynamic path minima data structure for an input tree of n nodes in the RAM model, which supports* pathmin *and* update *in* $O(\log n / \log \log n)$ *time, and* insert, insert-leaf *and* contract *in* $O(\log n / \log \log n)$ *amortized time using* $O(n)$ *space.*

3.3 Data structures for Dynamic Leaves

We present data structures that support path minima queries under inserting and deleting leaves. First, we consider the problem in the semigroup model, where the edge-weights are from a semigroup. Of course, such data structures work in the comparison model as well (see the explanation about the 1D-RMQ problem in the semigroup model in Section 1.2.1). We also consider the problem in the RAM model, and we give another approach to achieve the known upper bound $O(1)$ time for all the operations [AH00, KS08]. Notice that, the data structures of this section do not support updating of existing edge-weights.

3.3.1 Query-Update Trade-off in the Semigroup Model

In the semigroup model, we show how to maintain trees containing n nodes to support path minima queries using $4k - 1$ semigroup operations under insertions and deletions of leaves performed in $O(k\alpha_k(n))$ semigroup operations, for a parameter k, where $1 \leq k \leq \alpha(n)$. Our data structure is a dynamized version of the path minima structure of Alon and Schieber [AS87]. They showed how to support path minima queries in $4k - 1$ semigroup operations after $O(nk\alpha_k(n))$ preprocessing time [AS87]. We first demonstrate the data structure, and then we explain how it supports queries and updates. At the end, we analyze the complexities of the operations.

Data structure. Every pathmin(u, v) query can be reduced to two subqueries pathmin(c, u) and pathmin(c, v), where c is the LCA of u and v. Therefore, we construct a dynamic LCA data structure for the tree [CH05], and we only consider queries pathmin(u, v), where u is an ancestor of v. Let $k > 0$ be an input parameter to the preprocessing algorithm.

The data structure is recursive. We partition an input tree into micro trees using the micro-macro decomposition explained in Section 1.5.3 with parameter k. Also, we partition each micro tree into smaller micro trees recursively with parameter k. The decomposition of micro trees into smaller micro trees continues until the size of each micro tree becomes constant. In the last level, we precompute the answer of all possible queries within each micro tree. Recall that in this decomposition, every micro tree

has at most two boundary nodes (Lemma 1.2). Also, we partition the macro tree using the micro-macro decomposition recursively with parameter $k-1$. The data structures for the micro trees and the macro tree are constructed recursively with appropriate parameter.

Every query $q = \mathsf{pathmin}(u,v)$ is denoted as a *small query* if both of u and v lie within the same micro tree, and is denoted as a *large query* if u and v do not lie within the same micro tree. Notice that u is an ancestor of v by assumption. We consider large queries, while small queries are supported recursively. The decomposition divides q into at most three subqueries, each one of three different types: (1) a small subquery between v and the root of the micro tree containing v; (2) a small subquery between u and a boundary node of the micro tree containing u; (3) a large subquery that is between the root of the micro tree containing v and a boundary node of the micro tree containing u.

Recall that small queries are supported recursively. But queries of types 1 and 2 are some special cases of small queries (one side of them is either at a root or a boundary node) that can be answered in constant time by *precomputing* the answer of all such queries. Notice that no semigroup operation is required to answer each of these queries. Queries of type 3 are recursively supported using the macro tree with parameter $k-1$. Hence, we divide a large query into at most three subqueries, one of which is of type 3, and then we divide the subquery of type 3 into at most three subqueries *recursively*.

In addition to the number of semigroup operations, finding the proper memory cells is also a factor that affects the query time. A small query that lies within a micro tree may become a large query at some level of the decomposition. We need to find this level, without going through all the levels. To find this level quickly, we make a data structure as follows.

Consider the decomposition of the input tree T into micro trees, and each micro tree recursively into smaller micro trees. We build a tree L_T corresponding to such a decomposition. Every node in L_T corresponds to a micro tree in the decomposition. The root of L_T corresponds to the tree T. Each subtree of the root of L_T corresponds to the decomposition of each micro tree of the first level of the decomposition recursively. We construct a dynamic LCA data structure [CH05] for L_T. Using this data structure, we can find the top most level of the decomposition in which a small query splits into subqueries.

The size of micro trees and macro trees.   We set the size of the micro trees and the macro tree in the decomposition, such that we achieve the claimed bounds. Recall that k is the input parameter to the preprocessing algorithm.

The input tree is partitioned into $O(n/\alpha_{k-1}(n))$ micro trees, each of size $O(\alpha_{k-1}(n))$. Hence, the size of the macro tree is $O(n/\alpha_{k-1}(n))$. In the second level, each micro tree is partitioned recursively into $O(\alpha_{k-1}(n)/\alpha_{k-1}(\alpha_{k-1}(n)))$ micro trees, each of size $O(\alpha_{k-1}(\alpha_{k-1}(n)))$. In the last level, the size of each micro tree is constant. Hence, the number of levels is $O(\alpha_k(n))$ (Section 1.5.5).

As previously described, the partitioning of the input tree with parameter k is performed recursively on the macro tree with parameter $k-1$. Let $n' = O(n/\alpha_{k-1}(n))$ denote the size of the macro tree. Therefore, the macro tree is partitioned

into $O(n'/\alpha_{k-2}(n'))$ micro trees, each of size $O(\alpha_{k-2}(n'))$.

Base data structure. Our recursive data structure is built on base data structures in the last level of the recursion. More precisely, the data structures that we make for the macro trees are constructed recursively with parameter $k-1$. In the last level of the recursion where $k = 1$, we build a base data structure. This data structure has size $O(n \log n)$ and supports path minima queries using at most 1 semigroup operations, and performs updates in $O(\log n)$ amortized time. In the following, we present the data structure.

We decompose the input tree T into two micro trees of size $n/3$ and $2n/3$. We recursively decompose each micro tree into two micro trees similarly. Hence, the number of levels of the decomposition is $O(\log n)$. There is no macro tree in the decomposition. We precompute the answer of queries of type 1 and 2 within each micro tree. Also, we precompute the answer of all possible queries within the micro trees of the last level. We also make a dynamic LCA data structure for T, and a dynamic LCA data structure for L_T.

Our decomposition partitions a query into at most two subqueries, each of type 1 or 2. Therefore, a query can be answered in at most 1 semigroup operations. The appropriate level of the decomposition, in which two micro trees contain the subqueries can be found using the LCA data structure of L_T in constant time.

Suppose that we want to insert a new leaf ℓ adjacent to a node u in T such that the weight of the new edge between ℓ and u is w. We insert ℓ into each of the micro trees containing u in all the levels of the decomposition. Let μ be the micro tree containing u in level i of the decomposition. We precompute and store the answer of the query $\mathsf{pathmin}(r_\mu, \ell)$, where r_μ is the root of μ. Notice that this query is of type 1. The answer is determined by $\min\{w, \mathrm{weight}(\mathsf{pathmin}(r_\mu, u))\}$. We precompute this answer for all the levels in $O(\log n)$ time.

After inserting ℓ, if the size of a micro tree in some level i exceeds the threshold, then we split it into a constant number of smaller micro trees (see Section 1.5.3). Then at each level below level i, the appropriate micro trees split. If $i = 1$, it can be shown that the total cost of the splits in all the levels is $O(n \log n)$. We amortize the cost of all the splits over the linear number of insertions performed to make a micro tree full. Thus, the cost of a split is $O(\log n)$ amortized, and an insertion takes $O(\log n)$ amortized time.

Theorem 3.3 *For an input tree containing n nodes, there exists a data structure in the semigroup model using $O(nk\alpha_k(n))$ space, that supports path minima queries in $4k-1$ semigroup operations, and performs insertions and deletions of leaves in $O(k\alpha_k(n))$ amortized update time, for a parameter k, where $1 \le k \le \alpha(n)$.*

Proof. We prove the theorem by induction on k, that is, we assume that the bounds hold for $k' < k$, and then we prove them for k. The base of the induction is $k = 1$, which was proved by the base structure.

Query time. First we consider queries of form $\mathsf{pathmin}(u, v)$, where u is an ancestor of v. As explained before, the decomposition divides a query recursively into at most three subqueries. The number of recursive levels in the division of queries is at

most k. Subqueries of types 1 and 2 are supported without performing any semigroup operations. The number of semigroup operations to answer a query is derived from the equation $t_q(n,k) = 2 + t_q(n/\alpha_{k-1}(n), k-1)$, where $t_q(n,1) = 1$ (see the base structure). Therefore, the query time is $2k - 1$.

To answer a general query pathmin(u,v), we combine the answer of the subqueries pathmin(c,u) and pathmin(c,v), where c is the LCA of u and v. Each subquery is answered in $2k - 1$ operation. Therefore, the overall query time is $4k - 1$.

Insertion. We perform the insertion of a new leaf into the input tree in three stages: (1) we insert the leaf into the appropriate micro tree μ (recall that $|\mu| = O(\alpha_{k-1}(n))$); (2) we split μ if its size exceeds the threshold; and (3) we insert the new boundary nodes into the macro tree (notice that splitting μ results in making new boundary nodes). The cost of the first stage is calculated recursively. We amortize the cost of the second stage over all the insertions performed in μ, as described later. The cost of the third stage is $O((k-1)\alpha_{k-1}(n/\alpha_{k-1}(n)))$ by induction on k. We amortize this cost over all the insertions into μ, and we obtain $O(k)$ amortized time. In the following, we show that the amortized cost of a split is constant, and then we add up the costs of all the three stages.

We split μ into a constant number of smaller micro trees in linear time, using the algorithm described in Section 1.5.3. Then, the appropriate micro trees contained in μ at the level below should also split. The number of such micro trees which split is constant, say $c > 0$. Let n' be the size of μ. Thus, the cost of splitting μ is $t_{\text{SPLIT}}(n') = O(n') + c \cdot t_{\text{SPLIT}}(\alpha_{k-1}(n'))$. It can be shown that $t_{\text{SPLIT}}(n', k) = O(n')$. We amortize this cost over linear insertions into μ. Therefore, the cost of split is $O(1)$ amortized.

Now, we add up the cost of all the three stages using the equation $t_{\text{INS}}(n,k) = t_{\text{INS}}(\alpha_{k-1}(n), k) + O(1) + O(k)$. It can be shown that $t_{\text{INS}}(n,k) = O(k\alpha_k(n))$. Also, updating the LCA data structures can be done in constant time. Thus, the overall insertion time is $O(k\alpha_k(n))$.

Space. The data structure for the input tree consists of the data structures constructed for each micro tree of the first level (which have size $O(\alpha_{k-1}(n))$). It also includes the data structure of the macro tree in the first level, which has size $O(n/\alpha_{k-1}(n) \cdot (k-1) \cdot \alpha_{k-1}((n/\alpha_{k-1}(n)))) = O(nk)$ derived from the induction hypothesis on k. Therefore the overall space is calculated by the equation $S(n,k) = (n/\alpha_{k-1}(n)) \cdot S(\alpha_{k-1}(n), k) + O(nk)$. It can be shown that $S(n,k) = O(nk\alpha_k(n))$.

□

3.3.2   Constant Update Time in the Semigroup Model

We describe a linear space data structure which supports queries in $O(\alpha(n))$ time, and insertions and deletions of leaves in $O(1)$ amortized time for an input tree containing n nodes. Notice that if we simply replace k with $\alpha(n)$ in the complexities of the structure of Theorem 3.3, we obtain $O(\alpha(n))$ query time, $O(\alpha(n))$ update time, and $O(n\alpha(n))$ space. Here, we improve the update time and space to $O(1)$ and $O(n)$ respectively.

We decompose the input tree T into $O(n/\alpha^2(n))$ micro trees of size $O(\alpha^2(n))$ using the micro-macro decomposition of Lemma 1.2. The macro tree which has

size $O(n/\alpha^2(n))$ is represented using the structure of Theorem 3.3 for $k = \alpha(n)$. Recall that the data structure of Theorem 3.3 supports path minima queries and updates within the macro tree in $O(\alpha(n))$ time. We then decompose each of the micro trees into micro-micro trees of size $O(\alpha(n))$ with a macro-micro tree of size $O(\alpha(n))$. Path minima queries within a micro tree μ of size $\alpha^2(n)$ can be supported in $O(\alpha(n))$ by traversing the appropriate micro-micro trees and the macro-micro tree.

Inserting a leaf into μ can be supported in $O(1)$ time, by simply performing the insertion into the appropriate micro-micro tree. If the size of a micro-micro tree exceeds its maximum threshold, we split it and insert new leaves into the macro-micro tree all in $O(\alpha(n))$ time. If the size of a micro tree exceeds its maximum threshold, we again split it and insert new leaves into the structure of the macro tree which supports $O(\alpha(n))$ insertion time. Therefore, the insertion time is constant amortized.

The required space to store the data structure of the macro tree is $O(n/\alpha^2(n) \cdot \alpha(n)) = O(n)$. The micro-micro trees and the macro-micro trees are stored explicitly using $O(n)$ space.

Theorem 3.4 *For an input tree of n nodes, there exists a dynamic path minima data structure in the semigroup model using $O(n)$ space, that supports* pathmin *in $O(\alpha(n))$ time,* insert-leaf *and* delete-leaf *in amortized $O(1)$ time.*

3.3.3 Constant Query and Update Time in the RAM Model

We present a dynamic data structure in the RAM model that supports path minima queries in $O(1)$ time, and insertions and deletions of leaves in $O(1)$ amortized time. This is not a new result due to [AH00, KS08], but here, it is achieved by using another approach.

We decompose the tree into micro trees of size $O(\log n)$ with a macro tree of size $O(n/\log n)$ using the micro-macro decomposition of Lemma 1.2. We again decompose each micro tree into micro-micro trees of size $O(\log \log n)$ with a macro-micro tree of size $O(\log n/\log \log n)$. The operations within each micro-micro tree is supported using precomputed tables and Q-heaps. We do not store any representation for the micro trees. We represent the macro tree and each macro-micro.

The decomposition induces a division of queries to subqueries that are contained within a micro-micro tree, a macro-micro tree or the macro tree. The subqueries within the micro-micro trees are answered using precomputed tables in $O(1)$ time. The subqueries within the macro-micro trees and the macro tree are answered using the corresponding Cartesian trees in $O(1)$ time. Since the total number of subqueries is $O(1)$, we can compute the answer in $O(1)$ time.

To perform insert-leaf, we add the new leaf into the appropriate micro-micro tree using precomputed tables. Then, if the size of the micro tree exceeds its maximum threshold, we split it into at most four micro trees (Section 1.5.3). The new boundary nodes are inserted into the appropriate macro-micro tree (described later). If the size of the macro-micro tree also exceeds its maximum limit, we split it into at most four macro-micro trees similarly. During the split of a macro-micro tree that we remove at most three edges, we also split the micro-micro trees containing these three edges and distribute the micro-micro trees among the new macro-micro trees. Then the nodes that connect the new macro-micro trees to each other, are inserted into the macro tree.

We have shown that new leaves can be inserted into Cartesian trees in logarithmic time (Lemma 1.1). The following lemma shows that, during the split, the new nodes can be inserted into the macro-micro trees as leaves, which is similarly true for the macro tree. This is our main observation.

Lemma 3.2 *When a micro-micro tree is split, we can insert the new boundary nodes by performing* insert-leaf *using the Cartesian tree of the corresponding macro-micro tree.*

Proof. Lemma 1.1 explains how we can insert a leaf into the Cartesian tree in logarithmic time. We only need to show that the new boundary nodes can be inserted by performing the same operation insert-leaf. Let b_1 and b_2 be the two boundary nodes of a micro-micro tree, and let x be a new boundary node as a result of splitting the micro-micro tree. Recall that the edge (b_1, b_2) in the macro-micro tree is the path minima along the path between b_1 and b_2 in the micro-micro tree. Let w be the weight of (b_1, b_2). If x is not on the path between b_1 and b_2, then it is a leaf in the macro-micro tree. Otherwise, x splits the edge (b_1, b_2) into two edges (b_1, x) and (b_2, x). Obviously the weight of one of these two edges, w.l.o.g. (b_1, x), is equal to w, and the other one has a weight w', where $w' \geq w$. Consider the subtree S of the Cartesian tree rooted at the child of (b_1, b_2) that has b_2 as a descendant. Then x is a leaf, adjacent to b_2, in the part of the macro-micro tree corresponding to S. Thus, (b_2, x) can be inserted into the Cartesian tree as a leaf.   □

The precomputed tables and the representation of the micro-micro trees are similar to Section 3.2.1. To perform delete-leaf, we simply mark the deleted leaves because they have no effect on the result of future operations. Using global rebuilding for delete-leaf and the amortized analysis, we achieve the following data structure.

We represent the Cartesian trees by using the comparison based structure of [DLW09a], which supports $O(1)$ query time and logarithmic leaf insertion and deletion time. Therefore, Lemma 3.2 allows us to achieve the following.

Theorem 3.5 *There exists a dynamic path minima data structure for an input tree of n nodes using $O(n)$ space that supports* pathmin *in $O(1)$ time, and supports* insert-leaf *and* delete-leaf *in amortized $O(1)$ time.*

3.4   Lower bounds

In this section, we show some lower bounds for the query time and update time of two variants of the dynamic path minima problem by giving reductions from other problems. First, we draw attention to the fact that the dynamic 1D-RMQ problem defined in Section 1.2.1, can be trivially solved by the dynamic path minima problem using only the operations pathmin and update. The lower bounds proved by this reduction in different models, show the optimality of some of our data structures. Then, we consider the variant of the dynamic path minima problem, where the operations link and cut are available. The proved lower bounds for this problem show that the dynamic trees of Sleator and Tarjan achieves the best possible results only if we want fast update times, say as fast as query times, but not if we want a query time faster than the update time. In the following, let t_q denote the query time.

Reduction from dynamic 1D-RMQ. As mentioned above, the dynamic 1D-RMQ problem defined in Section 1.2.1, can be trivially solved by the dynamic path minima problem using only the operations pathmin and update. Indeed, the dynamic 1D-RMQ problem is a special case of the dynamic path minima problem, where the input tree is a path. Therefore, all the three lower bounds mentioned in Section 1.2.1, for the dynamic 1D-RMQ problem also apply to the variant of the dynamic path minima problem, where the only supported update operation is update as follows. Let t_u denote the running time of update.

In the cell probe model with word size b bits, path minima queries on a tree containing n nodes require $\Omega(\log n / \log(t_u b \log n))$ time, where t_u is the time for update. For example, this proves that update time $(\log n)^{O(1)}$ implies query time $\Omega(\log n / \log \log n)$.

In the comparison model, if update performs at most t_u comparisons, then path minima queries require at least $n/(e2^{2t_u}) - 1$ comparisons. For example, this proves that if path minima queries use $(\log n)^{O(1)}$ comparisons then updates require $\Omega(\log n)$ comparisons.

In the semigroup model, we obtain the lower bounds $t_q \log(t_u/t_q) = \Omega(\log n)$ and $t_u \log(t_q/t_u) = \Omega(\log n)$. For example, this proves that with update time $O(\log n)$, path minima queries would require time $\Omega(\log n)$ and vice versa.

3.4.1 Dynamic Edges in Weighted Forests

We prove cell probe lower bounds for the most general variant of the dynamic path minima problem, where all the update operations including link and cut are provided. Let t_u denote the maximum of the running time of link and cut. We show that if we want to support link and cut in a time within a constant factor of the query time, then $t_q = \Omega(\log n)$. Moreover, if we want a fast query time $t_q = o(\log n)$, then one of link or cut cannot be supported in $O(\log n)$ time, for example, if $t_q = O(\log n / \log \log n)$, then $t_u = \Omega(\log^{1+\varepsilon} n)$ for some $\varepsilon > 0$. We also show that $O(\log n / \log \log n)$ query time is the best achievable for polylogarithmic update time, for example, a faster query time $O(\log n / (\log \log n)^2)$ causes t_u to blow-up to $(\log n)^{\Omega(\log \log n)}$.

Reduction from fully dynamic connectivity. The fully dynamic connectivity problem on forests is defined as follows. We have to maintain a collection (forest) of undirected trees under three operations connect, link, and cut, where connect(x,y) returns true if there exists a path between the nodes x and y, and returns false otherwise. Let t_{con} be the running time of connect, and t_{update} be the maximum of the running times of link and cut. Pătraşcu and Demaine [PD06] proved the lower bound $t_{\text{con}} \log(2 + t_{\text{update}}/t_{\text{con}}) = \Omega(\log n)$ in the cell probe model.

This problem is reduced to the dynamic path minima problem as follows. We put a dummy root r on top of the forest, and connect r to an arbitrary node of each tree with an edge of weight $-\infty$. Thus the forest becomes a tree. For this tree, we construct a dynamic path minima data structure. The answer to connect(x,y) is false iff the answer to pathmin(x,y) is an edge of weight $-\infty$. To perform link(x,y), we first run pathmin(x,r) to find the edge e of weight $-\infty$ on the path from r to x. Then we remove e and insert the edge (x,y). To perform cut(x,y), we first run pathmin(x,r) to

find the edge e of weight $-\infty$. Then we change the weight of e to zero, and the weight of (x, y) to $-\infty$. Now, by performing pathmin(x,r), we figure out that x is connected to r through y, or y is connected to r through x. Without loss of generality, assume that x is connected to r through y. Therefore, we delete the edge (x, y), insert (x, r) with weight $-\infty$, and change the weight of e back to $-\infty$.

Thus, we obtain the trade-off $t_q \log((t_q + t_u)/t_q)) = \Omega(\log n)$. For example, we conclude that if $t_q = O(\log n/\log\log n)$, then $t_u = \Omega(\log^{1+\varepsilon} n)$, for some $\varepsilon > 0$. We can also show that if $t_u = O(t_q)$, then $t_q = \Omega(\log n)$.

Reduction from boolean union-find.   The boolean union-find problem is maintaining a collection of disjoint sets under the following operations: find(x,A): returns true if $x \in A$, and returns false otherwise; union(A,B): returns a new set containing the union of the disjoint sets A and B. Kaplan, Shafrir, and Tarjan [KST02] proved the trade-off $t_{\mathrm{find}} = \Omega(\frac{\log n}{\log t_{\mathrm{union}}})$ for this problem in the cell probe model, where t_{find} and t_{union} are the running time of find and union.

The incremental connectivity problem is the fully dynamic connectivity problem without the operation cut. The boolean union-find problem is trivially reduced to the incremental connectivity problem. The incremental connectivity problem is reduced to the dynamic path minima problem with the same reduction used above.

Therefore, we obtain $t_q = \Omega(\log n/\log(t_q + t_u))$.   We can conclude that when $t_q = O(\log n/(\log\log n)^2)$, slightly less than $O(\log n/\log\log n)$, then the running time of t_u blows-up to $(\log n)^{\Omega(\log\log n)}$.

3.5   Open Problems

There are several interesting questions related to the dynamic path minima problem that remain open. In this section, update time denotes the maximum of the running time of all the required update operations. In the most general setting, where all the operations are required, the only known solution is the dynamic trees (link-cut) trees of Sleator and Tarjan [ST83]. They can support all the operations in the semigroup model, using two basic operations root and evert in $O(\log n)$ amortized time. In the cell probe model, Pǎtraşcu and Demaine [PD06] proved, by reduction from the fully dynamic connectivity problem, that root is an expensive operation and link-cut trees of Sleator and Tarjan are optimal if we want to support root. It is not clear that if we can improve the link-cut trees by not using root. Pǎtraşcu and Thorup [PT06] conjectured that reducing the update time for dynamic data structures below the optimal query time is impossible, without a large blow-up in the query complexity. This conjecture and the lower bounds of Section 3.4 show that for logarithmic update time, link-cut trees are optimal in the RAM model. We list the following three open problems:

- Can we achieve $O(\log n/\log\log n)$ query time and $O(\log^{1+\varepsilon} n/\log\log n)$ update time? Notice that this upper bound touches the lower bound curve proved in Section 3.4.

- If we require the operations link and update, can we obtain $O(\log n/\log\log n)$ query time and $O(\log n)$ update time in the comparison model, or $O(\log n/\log\log n)$ for both the query time and update time in the RAM

model? The only structure that solves this problem is the link-cut trees that give logarithmic bounds. Note that our structures in Section 3.2 obtained these bounds by supporting insert, insert-leaf, and contract instead of link.

- If the only required update operation is link, can we achieve $O(\alpha(n))$ query time and $O(1)$ update time in the semigroup model and the comparison model? This is already known in the RAM model [AH00, KS08]. Recall that in the static case, with linear preprocessing time, the query time is $\Omega(\alpha(n))$ [AS87, Pet06].

Chapter 4

Range Diameter Queries

Abstract. The range diameter problem is considered in this chapter. The problem asks "which two points are farthest away in an orthogonal range within a point set?". The problem is studied for point sets in the plane. We step forward to address some questions about the space-efficiency of range diameter data structures such as: can we achieve sub-quadratic space to support queries in constant time? how fast can we answer a query using linear space? What are the special point sets, for which we can support queries in logarithmic time using linear space?

For an input point set of size n in the plane, we provide support for the hardness of the range diameter problem by showing a reduction from the set intersection problem. We slightly generalize an existing folklore conjecture on the hardness of the set intersection problem, thereby suggesting a lower bound of $\tilde{\Omega}(n^2)$ for the size of any data structure that supports range diameter queries in $O(1)$ time. We also prove a lower bound for a related problem which asks to find the furthest points in two given vertically-separated convex polygons for any arbitrary representation of them. Finally, we show that range diameter queries for convex polygons can be supported in logarithmic time using an $O(n + m\log m)$-space data structure, where m is the total modality of convex polygons.

4.1   Introduction

In this chapter, we consider the two dimensional (2D) range diameter problem. We preprocess a set of n points from $\mathbb{R}^2$ into a data structure such that given an orthogonal range query, we can find the two furthest points within the query efficiently.

As mentioned in Section 1.4, the standard divide and conquer technique to partition a query to subqueries, and combine (aggregate) the answer of the subqueries to obtain the final result, does not solve the range diameter problem. This property causes that the range diameter data structures need to store a lot of information about the relation of subqueries to each other in order to support the queries fast. In this chapter, we take a step forward to show this difficulty of the problem by relating it to a fundamental data structure problem that is popularly conjectured to be hard. We also prove a lower bound for another problem which is related to the range diameter problem. At the end, we consider the class of convex point sets for which we present a range diameter data structure that can overcome the difficulty of the problem depending of the total modality of input convex polygons.

4.1.1   Our Contributions

In Section 4.2, we give a reduction from the *set intersection* problem to the range diameter problem. The former problem is conjectured to need quadratic space for constant query time. This problem has fundamental applications in information retrieval [CP10a, PR10]. Assuming the conjecture in a stronger computational model, our reduction suggests a lower bound of $\tilde{\Omega}(n^2)$ space for $O(1)$ query time for general point sets. A general version of the conjecture implies $\tilde{\Omega}((n/k)^2)$ space for $\tilde{O}(k)$ query time, for a parameter k where $1 \leq k \leq n$.

In Section 4.3, we prove a lower bound for a related problem, where we have to find the two furthest points in two given convex polygons that are vertically separated. We prove that for any independent representation of the two polygons, any algorithm needs time nearly linear in the number of vertices of the smallest polygon. This lower bound not only addresses an open problem mentioned by Edelsbrunner [Ede85, Section 4], but might also give a direction towards proving the lower bound of Section 4.2 unconditionally.

In Section 4.4, we look at convex polygons by paying attention to their total modality, which is the total number of local maxima in the sequence of distances from each vertex to the other vertices. We present a data structure of size $O(n + m \log m)$ supporting range diameter queries in $O(\log n)$ time, where m is the total modality of inputs. This data structure beats the conditional lower bound proved in Section 4.2, when the total modality is $O(n^{2-\varepsilon})$.

4.2   Conditional Lower Bound: Relation to Set Intersection

In this section, we show that range diameter queries in the plane can verify the disjointness of two sets among a collection of sets. The latter problem is known as the *set intersection* problem. Our reduction implies conditional lower bounds for the range diameter problem. In the following, we define the set intersection problem, and then

we present the reduction algorithm. At the end, we explain the lower bounds obtained by this reduction.

Problem 4.1 *The set intersection problem with parameters (m, N, U) asks to preprocess m sets $S_1, S_2, \ldots, S_m \subseteq [U]$, where $\sum_{i=1}^{m} |S_i| = N$ such that given two query indexes i and j, we can efficiently verify if the sets S_i and S_j are disjoint or not.*

A naive solution would be tabulating the answer of disjointness for every pair of sets. The size of this table is $O(m^2)$ and the query time will be $O(1)$. Notice that we can assume $m \leq N$. Cohen and Porat [CP10a, CP10b] gave a data structure of size $O((N/k)^2)$ which supports queries in $O(k)$ time. They tabulate the answer of the queries for pairs of sets where both sets have at least k elements. To answer a query, if any of S_i or S_j, say S_i, has less than k elements, they search for each element of S_i in S_j, each in constant time using the linear perfect hashing of [Pag00].

In the following theorem, we give a reduction from the set intersection problem to the 2D range diameter problem.

Theorem 4.1 *Given a data structure that solves the 2D range diameter problem for any point set of size n using $s(n)$ space and $t(n)$ query time, we can solve the set intersection problem with parameters (m, N, U) using $s(2N)$ space and $t(2N)$ query time.*

Proof. For the set intersection problem, we are given m sets $S_1, S_2, \ldots, S_m \subseteq [U]$, where $\sum_{i=1}^{m} |S_i| = N$. We construct a point set of size $2N$, and we show that the answer to each of the m^2 set intersection queries can be found using the answer to an orthogonal range diameter query.

We map each element $e \in S_i$ to two points positioned on the first and third quadrant of the circle c_i with radius r_i centred on $(0,0)$. The positions are determined by the two intersection points of the line $y = ex$ with c_i (see Figure 4.1). It is clear that the distance between the two points corresponding to e is $2 \cdot r_i$. Let $r_i = 2^{i-1}$ for $i = 1, \ldots, m$. Notice that for $e \in S_i$ and $e \in S_j$, the distance between the corresponding points on the first quadrant of c_i and the third quadrant of c_j is $r_i + r_j$. By the triangle inequality, for $e \in S_i$ and $e' \in S_j$, where $e \neq e'$, the distance between the corresponding points of e and e' on the first quadrant of c_i and the third quadrant of c_j is less than $r_i + r_j$. Therefore, to verify the disjointness of S_i and S_j, we make a rectangular range query with bottom-left point $(-r_i, -r_i)$ and top-right point (r_j, r_j). If the diameter within this rectangle is $r_i + r_j$, then there is a common element in S_i and S_j, and if the diameter is smaller than $r_i + r_j$, then S_i and S_j are disjoint. □

Using the above reduction, we prove a conditional lower bound based on a conjecture for the set intersection problem. Pătraşcu and Roditty state the following folklore conjecture [PR10].

Conjecture 4.1 ([PR10, Conjecture 3]) *Any data structure that solves the set intersection problem with parameters $(m, N, \log^c m)$ for a large enough constant c, in $O(1)$ query time, requires $\tilde{\Omega}(m^2)$ space.*

We generalize this conjecture to the following, which then implies a time-space trade-off for the range diameter problem in terms of conditional lower bounds (see also [PR10]).
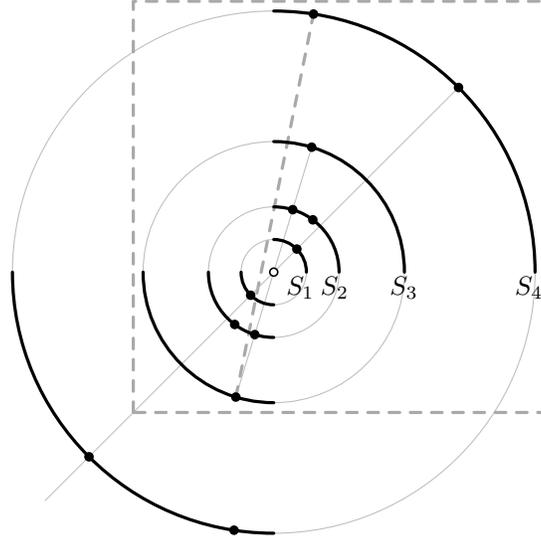
Figure 4.1: Example reduction for Theorem 4.1. The dashed point pair forms the range diameter for the dashed query range, implying that $S_3 \cap S_4 = \emptyset$. Note that for example $S_2 \cap S_3 \neq \emptyset$.

Conjecture 4.2 *Consider a computational model in which algorithms can work with unbounded real numbers using standard arithmetic and comparisons operations plus square root.*

Any data structure that solves the set intersection problem with parameters $(m, N, \log^c m)$ for a large enough constant c, in $\tilde{O}(k)$ query time, requires $\tilde{\Omega}((N/k)^2)$ space, for a parameter k where $1 \leq k \leq N$.

The following corollary states our conditional lower bound for the 2D range diameter problem.

Corollary 4.1 *Assuming Conjecture 4.2, any data structure that solves the 2D range diameter problem for a point set of size n with $\tilde{O}(k)$ query time, requires $\tilde{\Omega}((n/k)^2)$ space, for a parameter k where $1 \leq k \leq n$.*

Remarks.   We stress the importance of suggesting a stronger model for the set intersection problem in Conjecture 4.2. In the reduction algorithm, the N elements of the sets are transformed into a point set of size $2N$. The points of this point set have exponentially large coordinates. In particular the elements of the last set S_m are mapped to points on a circle with radius $r_m = 2^{m-1}$. This is the reason that we need to consider the set intersection problem in a model that supports unbounded real numbers. Also the reduction algorithm uses the operation square root to determine the coordinates of the points on the circles. Therefore, we have to provide this operation to the set intersection data structures as well. This model for the set intersection problem becomes more reasonable when we see that a related problem, the set disjointness problem, has been previously considered in a similar model, as we will discuss in the next section.

4.3   Relation to Set Disjointness

We consider a problem related to the 2D range diameter problem defined as follows. We separately preprocess two convex polygons P and Q in the plane that are vertically separated, that is, the preprocessing of each polygon is oblivious to the other polygon. Using the representations of P and Q, we have to compute the two furthest points of $P \cup Q$. We denote this problem as the *polygons diameter* problem. This problem usually arises as a subproblem in the 2D range diameter problem, in case we want to combine the answer of subqueries. We prove a lower bound for the polygons diameter problem by a reduction from the the *set disjointness* problem in communication complexity. This lower bound may be a step forward to prove the lower bound of Corollary 4.1 unconditionally. It also solves an open problem mentioned by Edelsbrunner [Ede85, Section 4].

In the set disjointness problem, we are given two sets A and B, each containing n real non-negative numbers. We have to determine whether A and B are disjoint or not, that is, $A \cap B = \emptyset$. There is a reduction from this problem to the *set diameter* problem defined as follows: Given a 2D point set of size n, compute the two furthest points. The reduction gives an $\Omega(n \log n)$ lower bound for the set diameter problem in the algebraic decision tree model [BO83]. The reduction is obtained by transforming the sets to a point set of size $|A| + |B|$ in the plane [PS91, Section 4.2.3]. Using the idea of that transformation, we reduce the *asymmetric* version of the set disjointness problem ($|A| \neq |B|$) in communication complexity to the polygons diameter problem. The asymmetric version of the set disjointness problem is also denoted as the lopsided set disjointness problem [Păt10].

In the asymmetric set disjointness problem in communication complexity, Alice and Bob receive sets A and B respectively, where $A, B \subseteq [n]$, and $|A| < |B| < n/2$, and they want to determine whether A and B are disjoint or not. Miltersen, Nisan, Safra, and Wigderson [MNSW98] showed that the number of bits that Alice and Bob need to communicate to solve the problem is $\Omega(|A|)$ (indeed they gave a stronger lower bound, but this is enough for our application). Their result immediately gives an $\Omega(\frac{|A|}{\log n})$ lower bound for the following problem.

Problem 4.2 *Given two sets A and B, where $A, B \subseteq [n]$, and $|A| < |B| < n/2$. Preprocess both of the sets separately (the preprocessing of each set is oblivious to the other set) such that after we are done with preprocessing, we can efficiently check whether A and B are disjoint or not.*

Now, we prove the lower bound for the polygons diameter problem by giving a reduction from Problem 4.2 in the following theorem.

Theorem 4.2 *For any representation of two convex polygons P and Q that are vertically separated, and $|P| < |Q|$, finding the two furthest points in $P \cup Q$ requires $\Omega(\frac{|P|}{\log n})$ time.*

Proof. Let A and B be the given sets in Problem 4.2. We construct two point sets P and Q corresponding to A and B respectively that are vertically separated, and we show that the disjointness of A and B can be verified by finding the diameter of $P \cup Q$.

We map each element $e \in A$ to a point positioned on the intersection point of the line $y = ex$ with the first quadrant of the unit circle. Similarly, we map each element $e \in B$ to a point positioned on the intersection point of the line $y = ex$ with the third quadrant of the unit circle. Hence, P has $|A|$ points and Q has $|B|$ points. It is clear that there exists an element e belonging to both A and B if and only if there exist a point $p \in P$ and a point $q \in Q$ such that the distance between p and q is 2. We compute the diameter of $P \cup Q$. If the diameter is 2 then there is a common element in A and B, and otherwise (the diameter is less than 2) A and B are disjoint.   □

4.4   Convex point sets

As it appears unlikely that we can get polylogarithmic query time when using $O(n^{2-\varepsilon})$ space for range diameter queries on sets of n points, we consider in this section the case of convex point sets: sets of points formed by the vertices of a convex polygon. For this case we give a data structure with logarithmic query time that uses space depending on the *total modality* of the polygon. The total modality of a polygon $P = (p_1, p_2, \ldots, p_n)$ (in clockwise order) is defined as the sum over all vertices p_i of the number of local maxima in the sequence of distances from p_i to the other vertices. More formally, take $p_0 := p_n$, $p_{n+1} := p_1$, and let $M_i = \{1 \le j \le n : d(p_i, p_{j-1}) < d(p_i, p_j) > d(p_i, p_{j+1})\}$ be the set of local maxima for vertex p_i where $d(p,q)$ is the Euclidean distance between points p and q, then $m = \sum_{i=1}^{n} |M_i|$. Note that $m = O(n^2)$. In the special case where $|M_i| \le 1$ for all i (and hence $m = O(n)$), P is called *unimodal*. Avis, Toussaint, and Bhattacharya [ATB82] show that not all convex polygons are unimodal, and that there exist polygons for which $m = \Theta(n^2)$. However, Abrahamson [Abr90] proves that under two definitions of random convex polygons, the expected maximum modality $\max_i\{|M_i|\}$ is small: for the convex hull of points drawn uniformly from a disc it is $O(\log n / \log \log n)$ and for the convex hull of points drawn from a two-dimensional normal distribution it is $O(\log \log n / \log \log \log n)$. Hence, for both cases $m = o(n \log n)$.

In this section we show that it is possible to answer range diameter queries by making a constant number of predecessor queries (in coordinate space) and two-dimensional range maximum queries on a set of m points (in rank space), yielding an $O(n + m \log m)$-space solution supporting queries in $O(\log n)$ time using the range maximum data structure of [GBT84] (in the word RAM, one can use the 2D range maximum data structure of [CLP11], which uses $O(m \log^\varepsilon m)$ space and answers queries in $O(\log \log m)$ time). Hence, we beat the conjectured lower bound of Corollary 4.1 in case the total modality of a given convex polygon is $O(n^{2-\varepsilon})$. We now discuss our solution, first describing how to find the sections of P intersecting a query, and then describing how to find the furthest point pair among those sections.

Finding sections.   The boundary of any query $q = [x_1 : x_2] \times [y_1 : y_2]$ can intersect P at most twice for each side as P is convex, so there can be at most four such sections. Assume for simplicity of exposition that no two vertices of P have the same x or y coordinate. To determine the index of the first and last vertex in each section, we construct predecessor/successor-search structures on the sets of x- and y-coordinates of
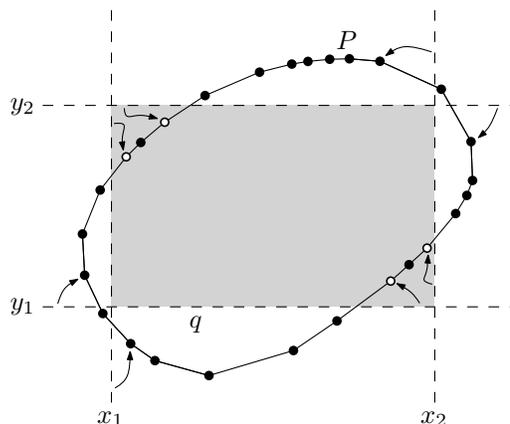
Figure 4.2: Predecessor and successor queries, indicated by arrows. White vertices are within the query range and determine the sections of $q \cap P$.

the vertices on the upper, lower, left, and right hulls of P (defined as usual). Then, we query the upper and lower hulls for the successors of x_1 and the predecessors of x_2, and similarly we query the left and right hulls for the successors of y_1 and predecessors of y_2. We now determine which of the eight vertices found in this way are contained in q and thereby find the sections of $q \cap P$ (see Figure 4.2). Let S_i be the sequence of vertices in the ith section of $q \cap P$ and let f_i be the index of the first vertex in S_i and l_i the index of the last vertex in S_i. The diameter d of the vertices in $q \cap P$ can be found by taking the maximum of all vertex pair distances: $\max_{i,j}\{\max_{p \in S_i, q \in S_j}\{d(p,q)\}\}$. We will now focus on determining the maximum vertex pair distance between two (possibly equal) sequences S_i and S_j. The final answer can then be obtained by simply taking the maximum distance over all sequence pairs.

Maximum distance between sequence pairs. Let Q be a set of points (i, j) for each local maximum p_j in the distance sequence of p_i, that is, $Q = \{(i, j) \mid j \in M_i\}$. To each point (i, j) we assign a weight of $d(p_i, p_j)$. Then we create a data structure D_Q over Q that can efficiently find the point with maximum weight within a given orthogonal query range (or $-\infty$ if no such point exists).

Lemma 4.1 *A query for the furthest point pair in sequences S_i and S_j can be answered using a constant number of 2D range maximum queries in D_Q.*

Proof. We need to determine the maximum distance within the range of S_j of the distance function for every point p in S_i. This maximum will either be one of the local maxima of the distance sequence of p, or be one of the end-vertices p_{f_j} or p_{l_j} of S_j (see Figure 4.3). Therefore, we can split the problem into two parts: 1) finding the maximum of the local maxima that lie within the range of S_j for all points in S_i, and 2) finding for the two end-vertices p_{f_j} and p_{l_j} the furthest vertex in S_i. The second part can again be split into two in the same way: 2.1) finding the maximum local maximum in the distance function of p_{f_j} (p_{l_j}) within S_i, and 2.2) finding the maximum distance between any of p_{f_j}, p_{f_j}, p_{f_i}, and p_{l_i}. Problem 2.2 can be solved in constant time by trying all combinations and problem 2.1 can be solved in the same way as
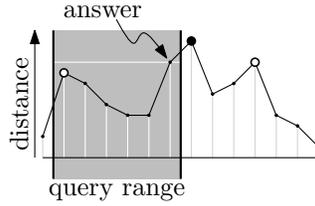
Figure 4.3: Example of a distance function for a point. The local maxima are drawn as white circles and the global maximum (the furthest point) is drawn as a black circle. A query range and the corresponding answer are also shown.

problem 1, so we only need to consider problem 1. Note that the points in Q inside a range $[f_i : l_i] \times [f_j : l_j]$ (for $f_i \leq l_i$ and $f_j \leq l_j$) represent the local maxima of points in S_i that are contained in S_j. Hence, a range maximum query over this range in D_Q returns the distance to the furthest local maximum. In case $f_i > l_i$ or $f_j > l_j$, we need two or four queries to cover the query range. This way, we solve problem 1, thereby proving the claim.   □

Theorem 4.3 *A convex point set can be preprocessed into an $O(n + m \log m)$-space data structure allowing range diameter queries in $O(\log n)$ time.*

Proof. We can use balanced binary search trees as predecessor/successor-search structures, requiring $O(n)$ space. For range maximum queries we use the data structure of [GBT84]. As $|Q| = m$, this results in the claimed space bound. Queries are answered by a constant number of predecessor/successor queries and 2D range maximum queries, all taking $O(\log n)$ time.   □

Remark.   For unimodal point sets (where $M_i \leq 1$ for all i) the solution can be simplified by replacing the 2D range maximum structure by a 1D range maximum structure, yielding $O(n)$ space and $O(\log n)$ query time.

Chapter 5

Succinct Dynamic Representation of Low-Arity Cardinal Trees

Abstract. In this chapter, we consider succinct representations of dynamic k-ary cardinal trees. We address the question "how fast can we traverse a k-ary cardinal tree, when k is not too large, while supporting enhanced queries and insertions/deletions of leaves efficiently at the current node of the traversal?"

For an input k-ary cardinal tree containing n nodes, where $k = O(\text{polylog}(n))$, we present a data structure of size $2n + n \log k + o(n \log k)$ bits, which is close to the information-theoretic lower bound. Our data structure supports the operations parent, i-th child, label-child, degree, subtree-size, preorder, is-ancestor in $O(1)$ time, and performs the update operations insert-label-leaf, and delete-label-leaf in $O(1)$ amortized time. The operations are performed at the current node of a traversal of the tree starting and ending at the root.

5.1   Introduction

In this chapter, we consider succinct representations for dynamic k-ary cardinal trees, which are rooted trees in which each node has at most k children and each edge is labeled by a symbol from a totally ordered set of fixed size k such as $\{1,\ldots,k\}$. In this problem, we have to design a data structure of size close to the information-theoretic lower bound to maintain an input k-ary cardinal tree, that supports the navigational operations and enhanced queries on the tree under inserting and deleting nodes. The following theorem states our result.

Theorem 5.1 *There exists a data structure of size $2n + n\log k + o(n\log k)$ bits to maintain an input k-ary cardinal tree containing n nodes, for $k = (\log n)^{O(1)}$. The data structure supports the navigational operations* parent, i-th child, *and* label-child *in $O(1)$ time, the enhanced queries* degree, subtree-size, preorder, *and* is-ancestor *in $O(1)$ time, and the update operations* insert-label-leaf *and* delete-label-leaf *in $O(1)$ amortized time.*

Achieving this result was posed as an open problem by Arroyuelo [Arr08]. Our data structure works in the unit-cost RAM model with word size $w = \Theta(\log n)$ bits. In Section 5.2, we present some preliminaries which form the building blocks of our data structure. In Section 5.3, we demonstrate our data structure, and in Section 5.4, we explain how to support the navigation, queries, and updates.

5.2   Preliminaries

5.2.1   Dynamic Arrays

A dynamic array [RR03] is a structure that supports maintaining a sequence of elements under accessing, inserting, and deleting elements in the sequence efficiently with a small memory overhead.

Lemma 5.1 (Dynamic arrays [RR03, RR08]) *There exists a data structure to represent an array of $\ell = w^{O(1)}$ elements, each of size $r = O(w)$ bits, using $\ell r + O(k\log\ell)$ bits, for any parameter $k \leq \ell$. This data structure supports accessing the element of the array in a given index in $O(1)$ time, and inserting/deleting an element in/from a given index in $O(1 + \ell r/kw)$ amortized time. The data structure requires a precomputed table of size $O(2^{\varepsilon w})$ bits for any fixed $\varepsilon > 0$.*

5.2.2   Dynamic Searchable Partial Sums

In the searchable partial sums problem, we have to maintain an array A of m numbers from the range $[0,\ldots,k-1]$ under the following operations:

sum(i): return the value $\sum_{j=1}^{i} A[j]$,

update(i,δ): set $A[i] = A[i] + \delta$, assuming that $A[i] + \delta < k$,

search(i): return the smallest j such that sum(j)$\geq i$.

This problem has been considered for different ranges of m and k [RRR01,HSS03]. Raman and Rao [RRR01] gave a data structure that solves the problem for $m = w^\varepsilon$, for any fixed $0 \le \varepsilon < 1$. Their data structure achieves $O(1)$ time for all the operations and uses $O(mw)$ bits (they do not bound k, that is, $k \le 2^w$). In the following, we present a data structure of size $m \log k + o(m \log k)$ bits, that achieves $O(1)$ time for all the operations, when $m = O(w^c)$, for a constant $c > 0$.

Lemma 5.2 *For any integer $n < 2^w$, there exists a searchable partial sums structure to represent an array of m elements from the range $[0, \ldots, k-1]$, using $m \log k + o(m \log k)$ bits and a precomputed table of size $o(n)$ bits, where $m = (\log n)^{O(1)}$. This data structure supports the operations* sum, update, *and* search *in $O(1)$ time.*

Proof. We pack every $w/\log k$ elements of the array into a word. Within each word, every b numbers denote a chunk, where $b = \log^{1/4} n$. Within each chunk, the operations can be supported in $O(1)$ time using a precomputed table of size $o(n)$ bits. The space usage to store all the chunks is $m \log k + o(m \log k)$ bits.

Now, we make a B-tree with branching factor at most b. Each leaf of the B-tree stores a pointer to one of the chunks such that scanning the chunks of the leaves from the left of the B-tree to the right gives the original array. The number of leaves is m/b and the depth of the B-tree is $O(1)$. At each internal node u, we maintain two arrays of length b. The i-th element of the first array maintains the sum of all the elements in the chunks that are descendants of the i-th child of u. The i-th element of the second array maintains the number of all the elements in the chunks that are descendants of the i-th child of u. The operations on these two arrays can be supported in $O(1)$ time, using a precomputed table of size $o(n)$ bits. Since the number of internal nodes is $O(m/b^2)$, the space usage for the B-tree is $O((m/b^2) \cdot (b(\log k + \log m))) = o(m \log k)$ bits.

The operations on the input array, can be performed by traversing the tree top-down and computing the operations at the internal nodes in $O(1)$ time. $\square$

5.2.3 Dynamic Data Structure for Balanced Parentheses

We maintain a *short* sequence of balanced parentheses under the operations rank, select, updates and the following operations performed in constant time:

findclose(i): find the position of the close parenthesis matching the open parenthesis in position i,

findopen(i): find the position of the open parenthesis that matches the closing parenthesis in position i,

enclose(i): given a parenthesis pair whose open parenthesis is in position i, return the position of the open parenthesis corresponding to the closest matching parenthesis pair enclosing i.

Lemma 5.3 *There exists a dynamic data structure of size $2m + o(m)$ bits to maintain a sequence of m pairs of balanced parenthesis using precomputed tables of size $o(n)$ bits, where $m = (\log n)^{O(1)}$. This data structure supports the operations:* findclose, findopen, *and* enclose *in $O(1)$ time, and supports inserting and deleting of the pair of parentheses "()" in $O(1)$ amortized time.*

Proof. This representation is similar to the one of [CHLS07]. We divide the sequence into chunks of size $w\ell$ bits, where $\ell = O(\sqrt{\log n})$. Each chunk is represented by a dynamic array of size $w\ell + O(\sqrt{\log n} \log \ell)$ bits (see Lemma 5.1), which allows us to access, insert, or delete a parenthesis at a given index in $O(1)$ time (amortized for updates) using a precomputed table of size $o(n)$ bits. Therefore, the total space used for the chunks is $2m + o(m)$ bits.

Now, we make a B-tree with branching factor b, where $b = O(\log^{1/4} n)$. Each leaf of the tree stores a pointer to a sub-chunk of size ℓ such that scanning the sub-chunks of the leaves from the left of the tree to the right gives the original sequence. The number of leaves is $2m/\ell$, and the depth of the tree is $O(1)$. At each internal node u, we maintain an array of length b such that its i-th element stores the number of open parenthesis in the chunks that are descendants of the i-th child of u. Since the array is small (i.e., $O(\log^{\frac{1}{4}} n \cdot \log \log n)$ bits), we can represent it by a searchable partial sums structure using a precomputed table of size $o(n)$ bits. This array is used to perform the operations rank and select in $O(1)$ time by traversing the tree from its root to the appropriate leaf. In addition to this array, similar to [CHLS07], we store seven arrays containing different information about the parentheses stored in the subtrees of u. These arrays are used to perform the parenthesis operations. Update operations are also straightforward. See [CHLS07] for more details. Since the number of internal nodes is $O(2m/(b\ell))$, the space usage for the B-tree is $O(2m/(bl) \cdot b \log m) = o(m)$ bits. $\qquad\square$

5.2.4 Dynamic Rank-Select Structure

The operations rank and select on bit vectors are defined in Section 1.1.1. They can also be defined for strings of characters similarly as follows. The operation $\mathsf{rank}_c(i)$ returns the number of occurrences of the character c before the position i in a string. The operation $\mathsf{select}_c(i)$ returns the position of the i-th c in a string. We present a data structure to maintain a *short* sequence over an alphabet of *small size* to support the operations rank and select under insertions and deletions.

Lemma 5.4 *There exists a dynamic representation of size $m \log k + o(m \log k)$ bits for a sequence of m symbols from an alphabet of size k using precomputed tables of size $o(n)$ bits, where m and k are $(\log n)^{O(1)}$. This data structure supports the operations* rank *and* select *in $O(1)$ time, and supports the update operations* insert *and* delete *in $O(1)$ amortized time.*

Proof. There exists a static data structure that supports the operations rank and select in $O(1)$ time for an alphabet of size k, using a multi-ary wavelet tree with $O(1)$ height (Theorem 3.2 of [FMMN07]). We dynamize their structure in the following way. We set the branching factor of their wavelet tree to be k', where $k' = O(\sqrt{\log n})$. At each internal node we use a dynamic rank/select structure for an alphabet of size k'. In the following, we explain this data structure. Note that the update operations do not change the structure of the wavelet tree, and thus only the internal node structures should be dynamized.

We pack every ℓ symbols of the sequence into a chunk of size $\ell \log k'$ bits, for $\ell = (w/\log k') \log^{1/4} n$. Each chunk is represented by a dynamic array of

size $\ell \log k' + O(\log^{1/4} n \log \ell)$ bits, which allows us to access, insert, or delete a symbol at a given index in $O(1)$ time (amortized for updates) using a precomputed table of size $o(n)$ bits (see Lemma 5.1). Therefore, the total space used for the chunks is $m \log k' + o(m \log k')$ bits.

Now, we make a B-tree with branching factor at most $\log^{\frac{1}{4}} n$. Each leaf of the B-tree stores a pointer to a sub-chunk of size w bits in one of the chunks such that scanning the sub-chunks of the leaves from the left of the B-tree to the right gives the original sequence. Therefore, each chunk corresponds to $\log^{1/4} n$ leaves. The number of leaves is $m/(\ell \log^{1/4} n)$ and the depth of the B-tree is $O(1)$. At each internal node u, we maintain $k + 1$ arrays, each of length $\log^{1/4} n$. One of the arrays is denoted by Size. The i-th element of the array Size maintains the number of symbols in the sub-chunks that are descendants of the i-th child of u. Each of the other k' arrays is for a symbol in the alphabet, and its i-th element maintains the number of the corresponding symbol in the leaves that are descendants of the i-th child of u. We represent each of these arrays by a searchable partial sums structure with $O(1)$ time for the partial sums operations, using a precomputed table of size $o(n)$ bits, since the arrays are small (that is, $O(\log^{\frac{1}{4}} n \cdot \log \log n)$ bits).

To perform the operation $\mathsf{rank}_\alpha(i)$, we traverse the B-tree top-down starting from the root. Let h be the sub-chunk containing the i-th symbol of the original sequence. At each internal node u, we count the number of α in the sub-chunks that are to the left of h, and are descendants of u. This counting can be performed in $O(1)$ time, using the partial sums structures that are constructed for the array Size and the array corresponding to α. At the leaf level, where we should perform rank in a sub-chunk of size w bits, we read the sub-chunk in $O(1)$ time and perform the rank using word-level computation. The operation $\mathsf{select}_\alpha(i)$ can be performed similarly in $O(1)$ time (the array Size is not required for select).

For the operations insert and delete, we perform them on the appropriate chunks in $O(1)$ amortized time (with the support of the dynamic arrays), and then we update the nodes of the B-tree along the appropriate path in a straightforward manner. Therefore, the total update time is $O(1)$ amortized. $\qquad \square$

5.2.5 Dynamic Predecessor Search Structure

Given a set of elements from a totally ordered set, the predecessor of a query element is the maximum element in the set which is smaller than or equal to the query element. We present a data structure to maintain a *small* set of elements over a *small* range, that supports predecessor search under insertions and deletions.

Lemma 5.5 *There exists a dynamic predecessor data structure of size $m \log k + o(m \log k)$ bits for a set containing m elements, where $m = (\log n)^{O(1)}$ and each element is from the range $[0 \cdots k - 1]$, using a precomputed table of size $o(n)$ bits. This data structure supports the operation* predecessor *in $O(1)$ time, and supports the update operations* insert *and* delete *in $O(1)$ amortized time.*

Proof. We maintain the set with an array that contains the elements in sorted order. For this structure, we use the same packing strategy and dynamic arrays as

we used in the proof of Lemma 5.4. We make a B-tree with branching factor b, where $b = \sqrt{\log n}$. Each leaf maintains b elements from the array, such that concatenating the leaves from left to right, gives the original array. The height of the tree is $O(1)$. At each internal nodes, we maintain b guiding indexes. Every node (including leaves) has $b \log k = o(w)$ bits which can be handled using a precomputed table of size $o(n)$ bits. To perform the operations, we traverse the tree top-down in $O(1)$ time. For the update operations, we also update the internal nodes in a bottom-up traversal. The rebalancing is applied as needed. This structure is similar to the one of [FW94] for atomic heaps. □

5.3   Data Structure

We present a succinct data structure of size $2n + n \log k + o(n \log k)$ bits to maintain an input k-ary cardinal tree containing n nodes under the navigational operations, enhanced queries, and insertions and deletions of leaves. In this section, we demonstrate the data structure, and in Section 5.4, we explain how it supports the required operations. This data structure supports the operations in the traversal model, where each operation is performed at the current node of a traversal starting and ending at the root of the tree (see Section 1.1.2 for more explanation of this model).

The input tree is decomposed into disjoint subtrees called micro trees. The data structure consists of the representation of every micro tree. The representation of each micro tree τ is a data structure that is constructed for τ to support the navigation, queries and updates within τ. This data structure also encodes some information about the micro trees that are neighbors to τ. This information is collected about pointers to the neighboring micro trees, and also about the subtree sizes of the neighboring micro trees. Our data structure is similar to the one presented in [Arr08].

When the traversal starts from the root, we are at the root of the micro tree τ that contains the root of the tree. The desired operations on the tree are performed on τ until the navigation take the current node of the traversal to a neighboring micro tree. Then, the representation of τ is used to find a pointer to move to the appropriate micro tree. When an update occurs at the current node within a micro tree τ, we have to update the representation of τ, and also we have to update the representation of its neighboring micro trees that store some information about τ, such as its subtree size. But the traversal model allows us to postpone updating the neighboring micro trees until the traversal arrives at those micro trees. Therefore, we can amortize updates over the navigation through the tree. In the following, we first describe the decomposition algorithm of the input tree, and then we demonstrate the data structure constructed for each micro tree.

Definitions.   Here, we define some of the terms that are used in the following sections. Let τ be a micro tree. The number of nodes (size) of τ is denoted by $|\tau|$. A *frontier* node of τ is a node that is adjacent to nodes in other micro trees. But the root of a micro tree is not a frontier node of the micro tree. If the root of τ is adjacent to a frontier node of another micro tree τ', then τ' is the *parent micro tree* of τ, and τ is a *child micro tree* of τ'. The number of frontier nodes of τ is denoted by $n_f(\tau)$.

Decomposition algorithm. We decompose the input tree into micro trees of size in the range $[\log^2 n \ldots k^2 \log^2 n]$ using the decomposition algorithm in [GRR06]. The micro tree containing the root might be smaller than $\log^2 n$. In our decomposition, we maintain the following. For each micro tree τ, we duplicate its frontier nodes such that every frontier is also the root of a child micro tree of τ. Therefore, all the children of a frontier node are in the same micro tree, each frontier node is a leaf, and is also adjacent to only one child micro tree, that is, $n_f(\tau)$ equals the number of child micro trees of τ.

5.3.1 Representation of Micro Trees

We preprocess each micro tree into a data structure that stores information about the topology of the micro tree, (that is, τ seen as an ordinal tree), its edge-labels, its frontier nodes, pointers to its child micro trees and its parent micro tree, and the subtree size of the root of each of its child micro trees. As previously mentioned, this data structure supports the operations to be performed within the micro tree, and allows us to navigate to a neighboring micro tree. It also provides information about the subtree size of the child micro trees, such that we can compute the subtree size of the current node at any time. The representation of a micro tree τ consists of the following parts:

- The topology of τ, represented using the DFUDS representation.

- The edge-labels of τ, represented using the sequence of the edge-labels stored in the DFUDS order. Recall that in the DFUDS order, all the edge-labels of the children of each internal node appear consecutively once the internal node is visited in the depth-first traversal (Section 1.1.1).

- Frontiers of τ stored in the order of the preorder traversal of τ.

- Pointers to the child micro trees of τ.

- A pointer to the parent micro tree of τ.

- The subtree size of all the child micro trees of τ.

In the following, we describe each of the representations in detail.

Tree topology of micro trees. We use a dynamic DFUDS representation to represent each of the micro trees. The DFUDS representation consists of a sequence of parentheses that along with a data structure supporting rank, select and balanced parentheses operations: findclose, findopen, and enclose on the sequence, can be used to perform navigation and query operations on an ordinal tree (Section 1.1.1 and [BDM$^+$05]).

Chan, Hon, Lam, and Sadakane [CHLS07] presented a data structure that maintains a sequence of balanced parentheses under findclose, findopen, enclose, and updates which can be performed in logarithmic time. Their data structure combined with a dynamic rank-select structure [MN08] gives a dynamic DFUDS representation that can be used to represent the topology of a cardinal tree under the update operations [Arr08]. Here, we improve this combined data structure to a dynamic DFUDS representation for micro trees that supports the operations in constant time. Our data structure, in the following lemma, is an immediate application of Lemma 5.3.

Lemma 5.6 *There exists a dynamic DFUDS representation of size $2|\tau| + o(|\tau|)$ bits for a micro tree τ containing $(\log n)^{O(1)}$ nodes using precomputed tables of size $o(n)$ bits. This data structure supports the operations* parent, *i*-th child, degree, subtree-size, is-ancestor, *and* preorder *all in $O(1)$ time, and supports the update operations* insert-leaf *and* delete-leaf *in $O(1)$ amortized time.*

Edge labels of micro trees.   We represent the edge-labels of a micro tree τ by the sequence L_τ of the edge-labels that appear in the DFUDS order of τ. To support the operations label-child, insert-label-leaf, and delete-label-leaf within τ, we construct two types of data structures. The first one is a dynamic rank-select data structure constructed for L_τ. The second one is a collection of dynamic data structures, in which each one supports the *predecessor* search operation over the edge-labels of an internal node under the update operations. The combination of these data structures along with the structure of Lemma 5.6 allows us to support the required operations within τ. As previously mentioned in Section 1.1.1, to perform the operations on the edge-labels, the rank of an edge-label among the edge-labels of an internal node determines the appropriate position of the edge-label which then can be used by the ordinal tree operations.

To construct the dynamic rank-select data structure for L_τ, we use the structure of Lemma 5.4. To make the dynamic predecessor structure for the edge-labels of each internal node, we use the structure of Lemma 5.5. The following lemma presents our data structure to represent the edge-labels of a micro tree, which is a combination of the data structures in Lemmas 5.4, 5.5, and 5.6.

Lemma 5.7 *For a k-ary cardinal tree τ of at most $k^2 \log^2 n$ nodes where $k = (\log n)^{O(1)}$, there exists a dynamic representation of size $2|\tau| + |\tau| \log k + o(|\tau| \log k)$ bits that supports the operations* parent, *i*-th child, label-child, degree, subtree-size, is-ancestor, *and* preorder *in $O(1)$ time, and supports the update operations* insert-leaf *and* delete-leaf *in $O(1)$ amortized time. The structure uses precomputed tables of size $o(n)$ bits.*

Frontiers of micro trees.   We make a data structure to represent the frontier nodes of a micro tree τ, such that we can check whether a node is a frontier or not. Let the i-th frontier of τ be the i-th frontier that appears in the order determined by the preorder traversal of τ. We build an array F_τ containing $n_f(\tau)$ elements (the number of frontiers), where $F_\tau[i]$ maintains the difference between the preorder number of the i-th frontier and the preorder number of the $i+1$-st frontier. Then, we make a *searchable partial sums* structure for F_τ.

Since F_τ has $(\log n)^{O(1)}$ elements, each of size $O(\log F_\tau)$ bits, we use the searchable partial sums structure of Lemma 5.2 that supports the operations sum, update, and search in $O(1)$ time, using $n_f(\tau) \log |\tau| + o(n_f(\tau) \log |\tau|)$ bits. Thus the overall space for all the micro trees is $o(n)$ bits.

Pointers to the neighboring micro trees.   For each micro tree τ, we store the pointers to the neighboring micro trees of τ to facilitate traversing through the tree. For the parent micro tree of τ, we store a pointer to a location where the root of τ is maintained

as a frontier in the parent micro tree. In other words, if τ' is the parent micro tree and its i-th frontier points to the root of τ, we store i.

Now, we consider the pointers to the child micro trees of τ. To provide fast access to the pointers, we store the pointers in a particular order in an array such that we can find a pointer once we find its corresponding frontier node. We make an array P_τ of size $n_f(\tau)$, such that $P_\tau[i]$ maintains the pointer to the child micro tree that is rooted at the i-th frontier of τ. The space usage to store P_τ for all the micro trees is $o(n)$ bits.

Subtree sizes. We make a data structure that allows us to compute the subtree size of the current node in $O(1)$ time. Let τ be the micro tree containing the current node. Lemma 5.6 allows us to compute the local subtree size of the current node, that is, its subtree size within τ. The subtree size of the current node is its local subtree size plus the sum of the subtree sizes of all the child micro trees that are descendants of the current node, where the subtree size of a child micro tree denotes the subtree size of its root in the whole tree (not its local subtree size). For the subtree sizes of the child micro trees, we make the following data structure.

For each micro tree τ, we make an array such that its i-th entry maintains the subtree size of the i-th child micro tree of τ. Notice that the length of the array is equal to the number of frontier nodes in τ, and each entry of the array is of size $O(\log n)$ bits. We build a searchable partial sums for this array using Lemma 5.2. Therefore, the size of the data structure is $n_f(\tau)\log n + o(n_f(\tau)\log n)$ bits, where $n_f(\tau)$ denotes the number of frontiers in τ. The overall space usage of this data structure for all the micro trees is $o(n)$ bits.

5.4 Supporting Operations

In this section, we explain how our data structure supports the required operations. Recall that the operations are performed in the traversal model, that is each operation is performed at the current node of a traversal starting and ending at the root. In the following, we assume that the current node is in a micro tree τ.

5.4.1 Navigation

We first determine whether the current node is the root of τ or is a frontier of τ (or is neither of them). To check whether the current node is a frontier of τ or not, we use the data structure that represents the frontier nodes of τ. Recall that the data structure maintains a searchable partial sums for an array containing the difference between the local preorder numbers of the successive frontiers. Using this data structure, we search for the preorder number of the current node. The preorder number of the current node is stored in the array iff the current node is a frontier. Then, we do the following.

(1) If the current node is neither the root nor a frontier node, then any navigation at the current node is supported in $O(1)$ time using the data structures constructed for the DFUDS sequence and the edge-labels of τ. (2) If the current node is the root of τ and the operation at hand is parent, then we move to the parent micro tree τ', and we find a frontier node of τ' that is a copy of the root of τ. Now that the current node is at that frontier node of τ', we perform parent within τ'. (3) If the current node is a frontier

node u in τ, and an operation asks to go to a child of u, then we follow the pointer to the appropriate child micro tree.

To perform the operation label-child(α) within τ, we find the rank i of α among all the edge-labels of the current node. Then, we perform the operation i-th child at the current node. To find i, we find the number of occurrences of α in the sequence of the edge-labels of τ before the position of the current node, and then find the position of the next α using the rank-select data structure that maintains the edge-labels of τ.

5.4.2 Enhanced Queries

Subtree size.   To compute the subtree size of the current node, we first compute the sum of the subtree sizes of the child micro trees that are descendants of the current node. For this, among all the frontiers of τ that are descendants of the current node, we find the left most one and the right most one. Then, we sum the subtree sizes using the partial sums structure that we constructed for τ to support subtree-size. To find the left most child micro tree that is a descendant of the current node, we search for the local preorder number of the current node in the array F_τ that maintains the difference between the local preorder numbers of the successive frontiers. The search is performed using the searchable partial sums structure constructed for F_τ in $O(1)$ time. The right most frontier that is a descendant of the current node is found as follows. Let v be the right most leaf (not necessarily a frontier) of τ that is also a descendant of the current node. We first find the local preorder number of v within τ by adding the local preorder number of the current node and its local subtree size within τ. Then we search for the local preorder number of v in the array F_τ. This determines the right most frontier node that is a descendant of the current node.

Preorder.   We always maintain the preorder number of the current node in the tree, whereas its local preorder number within τ can be computed using the DFUDS representation of τ. Once, the current node moves in τ using the navigational operations, we need to update the preorder number of the current node that we maintain.

When we move to the i-th child, we compute the preorder number of the i-th child by adding the following numbers together: (1) the preorder number of the current node; (2) the local subtree sizes of all the left siblings of the i-th child, that is, the j-th children for $j = 1 \ldots i - 1$; (3) the global subtree sizes of all the frontiers of τ that are descendants of the left siblings of the i-th child. The subtree sizes in item (2) can be computed by subtracting the local preorder number of the current node from the local preorder number of the i-th child. The subtree size in item (3) can be computed using the similar approach used to compute the global subtree size of the current node. Updating the preorder number of the current node, when we move to the parent, is performed using similar way.

Is-ancestor.   The operation is-ancestor(u) can be performed using subtree-size as follows. Notice that the input to is-ancestor(u) is the preorder number of the node u. This operation is straightforward due to the following fact. The current node is an ancestor of u iff the preorder number of the current node is smaller than the preorder number of u, and the preorder number of the current node plus its subtree size is greater than the preorder number of u.

Degree. Since our decomposition algorithm puts all siblings in the same micro tree, the operation degree is always performed locally within the micro tree that contains the current node. The operation degree is supported by the DFUDS representation. Thus, we can support it locally within each micro tree at the current node.

5.4.3 Updates

Insertion. To perform insert-label-leaf(α) in a micro tree τ, we update the representation of τ in the following way. The new leaf is inserted into the DFUDS sequence of τ, by inserting "()" at the appropriate position. This position is determined by performing a predecessor search for α among the edge-labels of the current node. The new label α is inserted into the sequence of the edge-labels of τ in the same position. If the preorder number of the new leaf is between the preorder numbers of two frontiers, then inserting the new leaf changes the difference between the preorder numbers of those two frontiers. Since the difference between the successive frontiers in τ is used to represent the frontiers of τ with a searchable partial sums structure, we need to fix this information by updating the data structure representing the frontiers. Therefore, we increment the appropriate entry in the array containing the difference between the preorder numbers of successive frontiers. All the above operations can be performed in $O(1)$ time.

If $|\tau|$ exceeds the value of $k^2 \log^2 n$, we split τ into micro trees of size in the range $[2 \log^2 n \cdots 2k \log^2 n]$ using the decomposition algorithm that we used in Section 5.3. Then we reconstruct the representation of each new micro tree. This can be performed by inserting leaves one by one into the new micro trees. The split and the construction of the representations of micro trees can be performed in $O(|\tau|) = O(k^2 \log^2 n)$ time. Since, this procedure makes micro trees of small enough size (at most $2k \log^2 n$), therefore, $O(k^2 \log^2 n)$ number of insert-leaf operations are required to make any of them full. It can be shown that the insertion time is $O(1)$ amortized.

Deletion. To perform delete-label-leaf(α) in a micro tree τ, we update the representation of τ similarly as insert-label-leaf(α). If $|\tau|$ becomes smaller than $\log^2 n$, then we combine τ with its parent micro tree. This can be performed by inserting the nodes of τ into the parent micro tree, in the preorder traversal of τ using insert-leaf. This procedure takes $O(|\tau|) = O(\log^2 n)$ time. The new micro trees that we construct in the split procedure of Section 5.4.3 are large enough (at least $2 \log^2 n$ size). It can be shown that the deletion time is $O(1)$ amortized.

5.4.4 Memory Management

An extendible array [BCD$^+$99] is a data structure which maintains a sequence of m equal-sized records, each assigned a unique index between 0 and $m - 1$, under accessing, creating, and discarding records. If each record in the sequence is of size r bits, the nominal size of the extendible array is nr bits. In our data structure, each micro tree can be viewed as a sequence of records. Thus, we maintain each micro tree using an extendible array.

Now, the problem is how to maintain all the micro trees in our data structure under accessing and modifying each micro tree, and also creating and destroying micro trees. Notice that the number of micro trees is at most $n/\log^2 n$. We maintain each micro tree in a separate location of the memory using an extendible array. The problem of maintaining a collection of extendible arrays is studied in [BCD$^+$99, RR08]. The nominal size of a collection of extendible arrays is the sum of the nominal sizes of the extendible arrays in the collection. The nominal size of all the micro trees is $s = 2n + n\log k + o(n\log k)$ bits. Using the ideas in [RR08], we can maintain the whole collection of micro trees in $s + O(nw/\log^2 n + \sqrt{snw/\log^2 n}) = 2n + n\log k + o(n\log k)$ bits

Bibliography

[Abr90]   Karl Abrahamson. On the modality of convex polygons. *Discrete and Computational Geometry*, 5:409–419, 1990.

[AE99]   Pankaj K. Agarwal and Jeff Erickson. Geometric range searching and its relatives. In *Proc. 1996 AMS-IMS-SIAM Advances in Discrete and Computational Geometry*, pages 1–56. American Mathematical Society, 1999.

[AFL07]   Amihood Amir, Johannes Fischer, and Moshe Lewenstein. Two-dimensional range minimum queries. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching*, volume 4580 of *LNCS*, pages 286–294. Springer-Verlag, 2007.

[Afs08a]   Peyman Afshani. On dominance reporting in 3d. In *Proc. 16th Annual European Symposium on Algorithms*, pages 41–51. Springer-Verlag, 2008.

[Afs08b]   Peyman Afshani. *On Geometric Range Searching, Approximate Counting and Depth Problems*. PhD thesis, University of Waterloo, 2008.

[AGKR04]   Stephen Alstrup, Cyril Gavoille, Haim Kaplan, and Theis Rauhe. Nearest common ancestors: a survey and a new distributed algorithm. *Theory of Computing Systems*, 37(3):441–456, 2004.

[AH00]   Stephen Alstrup and Jacob Holm. Improved algorithms for finding level ancestors in dynamic trees. In *Proc. 27th International Colloquium on Automata, Languages and Programming*, pages 73–84. Springer-Verlag, 2000.

[AHR98]   Stephen Alstrup, Thore Husfeldt, and Theis Rauhe. Marked ancestor problems. In *Proc. 39th Annual Symposium on Foundations of Computer Science*, page 534, Washington, DC, USA, 1998. IEEE Computer Society.

[AHU73]   A.V. Aho, J.E. Hopcroft, and J.D. Ullman. On finding lowest common ancestors in trees. In *Proc. 5th Annual ACM Symposium on Theory of Computing*, pages 253–265. ACM Press, 1973.

[AHU87]   A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data structures and algorithms*. Addison-Wesley series in computer science and information processing. Addison-Wesley, 1987.

[AKO04]   Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Algorithms of Discrete Algorithms*, 2(1):53–86, 2004.

[Arr08]   Diego Arroyuelo. An improved succinct representation for dynamic k-ary trees. In *Proc. 19th Annual Symposium on Combinatorial Pattern Matching*, pages 277–289. Springer-Verlag, 2008.

[AS87]   Noga Alon and Baruch Schieber. Optimal preprocessing for answering on-line product queries. Technical report, Department of Computer Science, School of Mathematical Sciences, Tel Aviv University, 1987.

[ASS97]   Stephen Alstrup, Jens P. Secher, and Maz Spork. Optimal on-line decremental connectivity in trees. *Information Processing Letters*, 64(4):161–164, 1997.

[ATB82]   David Avis, Godfried T. Toussaint, and Binay K. Bhattacharya. On the multimodality of distances in convex polygons. *Computers & Mathematics with Applications*, 8(2):153–156, 1982.

[AY10]   Mikhail J. Atallah and Hao Yuan. Data structures for range minimum queries in multidimensional arrays. In *Proc. 20th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 150–160. SIAM, 2010.

[BBG06]   Iwona Bialynicka-Birula and Roberto Grossi. Amortized rigidness in dynamic cartesian trees. In *Proc. 23rd Annual Symposium on Theoretical Aspects of Computer Science*, pages 80–91. Springer Verlag, 2006.

[BCD+99]   Andrej Brodnik, Svante Carlsson, Erik D. Demaine, J. Ian Munro, and Robert Sedgewick. Resizable arrays in optimal time and space. In *Proc. 6th Workshop on Algorithms and Data Structures*, pages 37–48. Springer-Verlag, 1999.

[BCR96]   Gerth Stølting Brodal, Shiva Chaudhuri, and Jaikumar Radhakrishnan. The randomized complexity of maintaining the minimum. *Nordic Journal of Computing*, 3(4):337–351, 1996.

[BDM+05]   David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[BDMR99]   David Benoit, Erik D. Demaine, J. Ian Munro, and Venkatesh Raman. Representing trees of higer degree. In *Proc. 6th International Workshop on Algorithms and Data Structures*, pages 169–180. Springer Verlag, 1999.

[BDR10]   Gerth Stølting Brodal, Pooya Davoodi, and S. Srinivasa Rao. On space efficient two dimensional range minimum data structures. In *Proc. 18th Annual European Symposium on Algorithms*, volume 6347 of *LNCS*, pages 171–182. Springer-Verlag, 2010.

[BDR11a]   Gerth Stølting Brodal, Pooya Davoodi, and S. Srinivasa Rao. On space efficient two dimensional range minimum data structures. *Algorithmica, Special issue on ESA 2010*, 2011. In press.

[BDR11b]   Gerth Stølting Brodal, Pooya Davoodi, and S. Srinivasa Rao. Path minima queries in dynamic weighted trees. In *Proc. 12th Algorithms and Data Structures Symposium*, LNCS. Springer-Verlag, 2011. To appear.

[BF79]   Jon Louis Bentley and Jerome H. Friedman. Data structures for range searching. *ACM Computing Surveys*, 11(4):397–409, 1979.

[BFCP⁺05]   Michael A. Bender, Martín Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005.

[BGSV89]   O. Berkman, Z. Galil, B. Schieber, and U. Vishkin. Highly parallelizable problems. In *Proc. 21st Annual ACM Symposium on Theory of Computing*, pages 309–319. ACM, 1989.

[BM99]   Andrej Brodnik and J. Ian Munro. Membership in constant time and almost-minimum space. *SIAM Journal on Computing*, 28(5):1627–1640, 1999.

[BMN⁺04]   Prosenjit Bose, Anil Maheshwari, Giri Narasimhan, Michiel Smid, and Norbert Zeh. Approximating geometric bottleneck shortest paths. *Computational Geometry*, 29(3):233–249, 2004.

[BO83]   Michael Ben-Or. Lower bounds for algebraic computation trees (preliminary report). In *Proc. 15th Annual ACM Symposium on Theory of Computing*, pages 80–86. ACM, 1983.

[CE87]   Bernard Chazelle and Herbert Edelsbrunner. Linear space data structures for two types of range search. *Discrete & Computational Geometry*, 2:113–126, 1987.

[CH05]   Richard Cole and Ramesh Hariharan. Dynamic lca queries on trees. *SIAM Journal on Computing*, 34(4):894–923, 2005.

[Cha82]   Bernard Chazelle. A theorem on polygon cutting with applications. In *Proc. 23rd Annual Symposium on Foundations of Computer Science*, pages 339–349, 1982.

[Cha87]   Bernard Chazelle. Computing on a free tree via complexity-preserving mappings. *Algorithmica*, 2:337–361, 1987.

[Cha88]   Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.

[Cha10]   Timothy M. Chan. Optimal partition trees. In *Proc. 26th Symposium on Computational Geometry*, pages 1–10. ACM, 2010.

[CHLS07]   Ho-Leung Chan, Wing-Kai Hon, Tak Wah Lam, and Kunihiko Sadakane. Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms*, 3(2), 2007.

[Cla96]   David Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, 1996.

[CLP11]   Timothy M. Chan, Kasper Green Larsen, and Mihai Pătraşcu. Orthogonal range searching on the RAM, revisited. In *Proc. 27th ACM Symposium on Computational Geometry*, 2011.

[CM96]   David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage (extended abstract). In *Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 383–391. SIAM, 1996.

[CP10a]   Hagai Cohen and Ely Porat. Fast set intersection and two-patterns matching. *Theoretical Computer Science*, 411(40-42):3795–3800, 2010.

[CP10b]   Hagai Cohen and Ely Porat. On the hardness of distance oracle for sparse graph. *The Computing Research Repository (arXiv)*, abs/1006.1117, 2010.

[CPS08]   Gang Chen, Simon J. Puglisi, and W. F. Smyth. Lempel-Ziv factorization using less time and space. *Mathematics in Computer Science*, 1:605–623, 2008.

[CR89]   Bernard Chazelle and Burton Rosenberg. Computing partial sums in multidimensional arrays. In *Proc. 5th Annual Symposium on Computational Geometry*, pages 131–139. ACM, 1989.

[CR91]   Bernard Chazelle and Burton Rosenberg. The complexity of computing partial sums off-line. *International Journal of Computational Geometry and Applications*, 1(1):33–45, 1991.

[dBCvK08] Mark de Berg, Otfried Cheong, and Marc van Kreveld. *Computational geometry: algorithms and applications*. Springer Verlag, 2008.

[DCW93]   John J. Darragh, John G. Cleary, and Ian H. Witten. Bonsai: a compact representation of trees. *Software - Practice and Experience*, 23(3):277–291, 1993.

[DLW09a]   Erik D. Demaine, Gad M. Landau, and Oren Weimann. On cartesian trees and range minimum queries. In *Proc. 36th International Colloquium on Automata, Languages and Programming*, volume 5555 of *LNCS*, pages 341–353. Springer-Verlag, 2009.

[DLW09b]   Erik D. Demaine, Gad M. Landau, and Oren Weimann. On cartesian trees and range minimum queries. `http://cs.haifa.ac.il/ ~landau/gadi/icalp09oren.pdf`, 2009. Manuscript.

[DPM05]   Sartaj Sahni Dinesh P. Mehta. *Handbook Of Data Structures And Applications (Chapman & Hall/Crc Computer and Information Science Series.)*. Chapman & Hall/CRC, 2005.

[DR11]   Pooya Davoodi and S. Srinivasa Rao. Succinct dynamic cardinal trees with constant time operations for small alphabet. In *Proc. 8th Annual Conference on Theory and Applications of Models of Computation*, LNCS. Springer-Verlag, 2011. To appear.

[DRT92]   Brandon Dixon, Monika Rauch, and Robert Endre Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM Journal on Computing*, 21(6):1184–1192, 1992.

[Ede85]   Herbert Edelsbrunner. Computing the extreme distances between two convex polygons. *Journal of Algorithms*, 6(2):213–224, 1985.

[FH06]   Johannes Fischer and Volker Heun. Theoretical and practical improvements on the rmq-problem, with applications to lca and lce. In *Proc. 17th Annual Symposium on Combinatorial Pattern Matching*, volume 4009 of *LNCS*, pages 36–48. Springer-Verlag, 2006.

[FH07]   Johannes Fischer and Volker Heun. A new succinct representation of rmq-information and improvements in the enhanced suffix array. In *Proc. 1st International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, volume 4614 of *LNCS*, pages 459–470. Springer-Verlag, 2007.

[FHS08]   Johannes Fischer, Volker Heun, and Horst Martin Stühler. Practical entropy-bounded schemes for o(1)-range minimum queries. In *Proc. 18th Data Compression Conference*, pages 272–281. IEEE Computer Society, 2008.

[Fis10]   Johannes Fischer. Optimal succinctness for range minimum queries. In *Proc. 9th Latin American Theoretical Informatics Symposium*, volume 6034 of *LNCS*, pages 158–169. Springer-Verlag, 2010.

[FM10]   Arash Farzan and J. Ian Munro. Succinct representation of dynamic trees. *Theoretical Computer Science*, In Press, Corrected Proof, 2010.

[FMMN07]   Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2), 2007.

[FMN08]   Johannes Fischer, Veli Mäkinen, and Gonzalo Navarro. An(other) entropy-bounded compressed suffix tree. In *Proc. 19th Annual Symposium on Combinatorial Pattern Matching*, volume 5029 of *LNCS*, pages 152–165. Springer-Verlag, 2008.

[Fre81]   Michael L. Fredman. A lower bound on the complexity of orthogonal range queries. *Journal of the ACM*, 28(4):696–705, 1981.

[Fre85]   Greg N. Frederickson.   Data structures for on-line updating of mini-
            mum spanning trees, with applications. *SIAM Journal on Computing*,
            14(4):781–798, 1985.

[FRR09]   Arash Farzan, Rajeev Raman, and S. Srinivasa Rao. Universal succinct
            representations of trees? In *Proc. 36th International Colloquium on Au-
            tomata, Languages and Programming*, pages 451–462. Springer, 2009.

[FS89]   Michael L. Fredman and Michael E. Saks. The cell probe complexity
            of dynamic data structures. In *Proc. 21st Annual ACM Symposium on
            Theory of Computing*, pages 345–354. ACM Press, 1989.

[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.

[GBT84]   Harold N. Gabow, Jon Louis Bentley, and Robert E. Tarjan. Scaling and
            related techniques for geometry problems. In *Proc. 16th annual ACM
            symposium on Theory of computing*, pages 135–143. ACM Press, 1984.

[GHSV07]   Ankur Gupta, Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. A
            framework for dynamizing succinct data structures. In *Proc. 34th Inter-
            national Colloquium on Automata, Languages and Programming*, pages
            521–532. Springer Verlag, 2007.

[GKP88]   Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete
            Math*.   Addison-Wesley Longman Publishing Co., Inc., Boston, MA,
            USA, 1st edition, 1988.

[GM07]   Anna Gál and Peter Bro Miltersen. The cell probe complexity of succinct
            data structures. *Theoretical Computer Science*, 379(3):405–417, 2007.

[Gol07]   Alexander Golynski. Optimal lower bounds for rank and select indexes.
            Theoretical Computer Science, 387(3):348–359, 2007.

[GRR06]   Richard F. Geary, Rajeev Raman, and Venkatesh Raman. Succinct ordi-
            nal trees with level-ancestor queries. *ACM Transactions on Algorithms*,
            2(4):510–534, 2006.

[GT04]   Loukas Georgiadis and Robert E. Tarjan. Finding dominators revisited:
            extended abstract. In *Proc. 15th Annual ACM-SIAM symposium on Dis-
            crete algorithms*, pages 869–878. SIAM, 2004.

[Gup05]   Prosenjit Gupta. Algorithms for range-aggregate query problems involv-
            ing geometric aggregation operations. In *Proc. 16th International Sympo-
            sium on Algorithms and Computation*, pages 892–901. Springer Verlag,
            2005.

[Hag98]   Torben Hagerup. Sorting and searching on the word ram. In *Proc. 15th
            Annual Symposium on Theoretical Aspects of Computer Science*, pages
            366–398. Springer-Verlag, 1998.

[Har85]   Dov Harel. A linear time algorithm for finding dominators in flow graphs and related problems. In *Proc. 17th Annual ACM Symposium on Theory of Computing*, pages 185–194. ACM Press, 1985.

[HSS03]   Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Succinct data structures for searchable partial sums. In *Proc. 14th International Symposium on Algorithms and Computation*, pages 505–516. Springer, 2003.

[HT84]   Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

[HW87]   David Haussler and Emo Welzl. epsilon-nets and simplex range queries. *Discrete & Computational Geometry*, 2:127–151, 1987.

[ICK+08]   Costas S. Iliopoulos, Maxime Crochemore, Marcin Kubica, M. Sohel Rahman, and Tomasz Walen. Improved algorithms for the range next value problem and applications. In *Proc. 25th International Symposium on Theoretical Aspects of Computer Science*, volume 1 of *Leibniz International Proceedings in Informatics*, pages 205–216. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008.

[Jac89]   Guy Jacobson. *Succinct Static Data Structures*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1989.

[JGKS08]   Ravi Janardan, Prosenjit Gupta, Yokesh Kumar, and Michiel H. M. Smid. Data structures for range-aggregate extent queries. In *Proc. 20th Annual Canadian Conference on Computational Geometry*, 2008.

[Kin97]   Valerie King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18(2):263–270, 1997.

[Kir83]   David G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12(1):28–35, 1983.

[Knu98]   Donald E. Knuth. *The art of computer programming, volume 3: (2nd edition) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[Kom85]   János Komlós. Linear verification for spanning trees. *Combinatorica*, 5(1):57–65, 1985.

[KS08]   Haim Kaplan and Nira Shafrir. Path minima in incremental unrooted trees. In *Proc. 16th Annual European Symposium on Algorithms*, volume 5193 of *LNCS*, pages 565–576. Springer-Verlag, 2008.

[KST02]   Haim Kaplan, Nira Shafrir, and Robert Endre Tarjan. Meldable heaps and boolean union-find. In *Proc. 34th annual ACM symposium on Theory of computing*, pages 573–582. ACM Press, 2002.

[Lue78]   George S. Lueker. A data structure for orthogonal range queries. In *Proc. 19th Annual Symposium on Foundations of Computer Science*, pages 28–34. IEEE Computer Society, 1978.

[Mat92]   Jirí Matousek. Efficient partition trees. *Discrete & Computational Geometry*, 8:315–334, 1992.

[Mat93]   Jirí Matousek. Range searching with efficient hiearchical cutting. *Discrete & Computational Geometry*, 10:157–182, 1993.

[Meh84]   Kurt Mehlhorn. *Multi-dimensional Searching and Computational Geometry*. Springer-Verlag, Heidelberg, West Germany, 1st edition, 1984.

[Mil]   Peter Bro Miltersen. Cell probe complexity - a survey. Advances in Data Structures Workshop (Pre-workshop of FSTTCS), 1999. `http://www.daimi.au.dk/~bromille/Papers/survey3.ps`.

[MN08]   Veli Mäkinen and Gonzalo Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms*, 4(3), 2008.

[MNSW98]   Peter Bro Miltersen, Noam Nisan, Shmuel Safra, and Avi Wigderson. On data structures and asymmetric communication complexity. *Journal of Computer and System Sciences*, 57(1):37–49, 1998.

[MR01]   J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[MRS01]   J. Ian Munro, Venkatesh Raman, and Adam J. Storm. Representing dynamic binary trees succinctly. In *Proc. 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 529–536. SIAM, 2001.

[Mun96]   J. Ian Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science*, LNCS, pages 37–42. Springer-Verlag, 1996.

[Mut02]   S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. 13th Annual ACM-SIAM symposium on Discrete algorithms*, pages 657–666. SIAM, 2002.

[MW94]   Udi Manber and Sun Wu. Glimpse: A tool to search through entire file systems. In *USENIX Winter Technical Conference*, pages 23–32. USENIX Association, 1994.

[Net99]   David Michael Neto. *Efficient cluster compensation for lin-kernighan heuristics*. PhD thesis, University of Toronto, Toronto, Ontario, Canada, Canada, 1999.

[Niv09]   Gabriel Nivasch. Inverse ackermann without pain. `http://www.yucs.org/~gnivasch/alpha/`, 2009.

[Pag00]   Rasmus Pagh. Faster deterministic dictionaries. In *Proc. 11th Symposium on Discrete Algorithms*, pages 487–493, 2000.

[Păt10]   Mihai Pătraşcu. Unifying the landscape of cell-probe lower bounds. *The Computing Research Repository (arXiv)*, abs/1010.3783, 2010.

[PD06]   Mihai Pătraşcu and Erik D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM Journal on Computing*, 35(4):932–963, 2006. See also STOC 2004.

[Pet06]   Seth Pettie. An inverse-ackermann type lower bound for online minimum spanning tree verification. *Combinatorica*, 26(2):207–230, 2006.

[Poo03]   Chung Keung Poon. Dynamic orthogonal range queries in OLAP. *Theory of Computing Systems*, 296(3):487–510, 2003.

[PR10]   Mihai Pătraşcu and Liam Roditty. Distance oracles beyond the Thorup-Zwick bound. In *Proc. 51th Annual IEEE Symposium on Foundations of Computer Science*, pages 815–823. IEEE Computer Society, 2010.

[PS85]   Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, USA, 1st edition, 1985.

[PS91]   F.P. Preparata and M.I. Shamos. *Computational geometry: an introduction*. Texts and monographs in computer science. Springer, 1991.

[PT06]   Mihai Pătraşcu and Mikkel Thorup. Time-space trade-offs for predecessor search. In *Proc. 38th annual ACM symposium on Theory of computing*, pages 232–240. ACM Press, 2006.

[RR99]   Venkatesh Raman and S. Srinivasa Rao. Static dictionaries supporting rank. In *Proc. 10th International Symposium on Algorithms and Computation*, pages 18–26. Springer Verlag, 1999.

[RR03]   Rajeev Raman and S. Srinivasa Rao. Succinct dynamic dictionaries and trees. In *Proc. 30th International Colloquium on Automata, Languages and Programming*, pages 357–368. Springer-Verlag, 2003.

[RR08]   Rajeev Raman and S. Srinivasa Rao. Succinct dynamic dictionaries and trees. Manuscript, 2008.

[RRR01]   Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct dynamic data structures. In *Proc. 7th Workshop on Algorithms And Data Structures*, pages 426–437. Springer-Verlag, 2001.

[RRS07]   Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding -ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4), 2007.

[Sad07a]   Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.

[Sad07b]   Kunihiko Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5(1):12–22, 2007.

[Sax09]   Sanjeev Saxena. Dominance made simple. *Information Processing Letters*, 109(9):419–421, 2009.

[Sei06]   Raimund Seidel. Understanding the inverse ackermann function. PDF presentation, March 2006. 22nd European Workshop on Computational Geometry, Delphi, Greece.

[SG06]   R. Sharathkumar and Prosenjit Gupta. Range-aggregate proximity detection for design rule checking in vlsi layouts. In *Proc. 18th Annual Canadian Conference on Computational Geometry*, pages 151–154, 2006.

[SN10]   Kunihiko Sadakane and Gonzalo Navarro. Fully-functional succinct trees. In *Proc. 21st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 134–149. SIAM, 2010.

[ST83]   Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of computer and system sciences*, 26(3):362–391, 1983.

[SV88]   Baruch Schieber and Uzi Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM Journal on Computing*, 17(6):1253–1262, 1988.

[Tar78]   Robert Endre Tarjan. Complexity of monotone networks for computing conjunctions. *Algorithmic aspects of combinatorics*, page 121, 1978.

[Tar79a]   Robert Endre Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, 1979.

[Tar79b]   Robert Endre Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18(2):110–127, 1979.

[TP04]   Yufei Tao and Dimitris Papadias. Range aggregate processing in spatial databases. *IEEE Transactions on Knowledge and Data Engineering*, 16(12):1555–1570, 2004.

[VM07]   Niko Välimäki and Veli Mäkinen. Space-efficient algorithms for document retrieval. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching*, volume 4580 of *LNCS*, pages 205–215. Springer-Verlag, 2007.

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

[Wil82]   Dan E. Willard. Polygon retrieval. *SIAM Journal on Computing*, 11(1):149–165, 1982.

[Yao82]   Andrew Chi-Chih Yao. Space-time tradeoff for answering range queries (extended abstract). In *Proc. 14th annual ACM symposium on Theory of computing*, pages 128–136. ACM Press, 1982.