Efficient Data Structures for Partial Orders, Range Modes, and Graph Cuts

by

Bryce Sandlund

A thesis presented to the University of Waterloo in fulfillment of the thesis requirement for the degree of Doctor of Philosophy in Computer Science

Waterloo, Ontario, Canada, 2021

© Bryce Sandlund 2021

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Pat Morin Professor Carleton University
Supervisor(s):	J. Ian Munro University Professor and Canada Research Chair in Algorithm Design Cheriton School of Computer Science, University of Waterloo
Internal Member:	Lap Chi Lau Associate Professor Cheriton School of Computer Science, University of Waterloo
Internal-External Member:	Joseph Cheriyan Professor University of Waterloo
Other Member(s):	Trevor Brown Assistant Professor Cheriton School of Computer Science, University of Waterloo

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

This thesis considers the study of data structures from the perspective of the theoretician, with a focus on simplicity and practicality. We consider both the time complexity as well as space usage of proposed solutions. Topics discussed fall in three main categories: partial order representation, range modes, and graph cuts.

We consider two problems in partial order representation. The first is a data structure to represent a lattice. A lattice is a partial order where the set of elements larger than any two elements x and y are all larger than an element z, known as the join of x and y; a similar condition holds for elements smaller than any two elements. Our data structure is the first correct solution that can simultaneously compute joins and the inverse meet operation in sublinear time while also using subquadratic space. The second is a data structure to support queries on a dynamic set of one-dimensional ordered data; that is, essentially any operation computable on a binary search tree. We develop a data structure that is able to interpolate between binary search trees and efficient priority queues, offering more-efficient insertion times than the former when query distribution is non-uniform.

We also consider static and dynamic exact and approximate range mode. Given one dimensional data, the range mode problem is to compute the mode of a subinterval of the data. In the dynamic range mode problem, insertions and deletions are permitted. For the approximate problem, the element returned is to have frequency no less than a factor $(1+\epsilon)$ of the true mode, for some $\epsilon > 0$. Our results include a linear-space dynamic exact range mode data structure that simultaneously improves on best previous operation complexity and an exact dynamic range mode data structure that breaks the $\Theta(n^{2/3})$ time per operation barrier. For approximate range mode, we develop a static succinct data structure offering a logarithmic-factor space improvement and give the first dynamic approximate range mode data structure. We also consider approximate range selection.

The final category discussed is graph and dynamic graph algorithms. We develop an optimal *offline* data structure for dynamic 2− and 3− edge and vertex connectivity. Here, the data structure is given the entire sequence of operations in advance, and the dynamic operations are edge insertion and removal. Finally, we give a simplification of Karger's near-linear time minimum cut algorithm, utilizing heavy-light decomposition and iteration in place of dynamic programming in the subroutine to find a minimum cut of a graph G that cuts at most two edges of a spanning tree T of G.

Acknowledgments

This thesis marks the culmination of seven years of graduate school. I would not be where I am today without the help and support of so many. First and foremost, I'd like to thank my advisor, Ian Munro. Ian was always enthusiastic to talk algorithms and through his large group at Waterloo, facilitated numerous prolific collaborations. Thank you for showing me the area of data structure design, with all of its exciting puzzles and balancing acts. I would also like to thank Richard Peng. Despite no formal relationship, Richard was an instrumental mentor to me, including me on several research projects and further introducing me to some extremely bright young collaborators. A third mentor I wish to express my appreciation towards is Eric Bach. Eric showed me how to be a successful researcher, guiding me during my tenure at Wisconsin-Madison. He also supported me in my decision to switch my research focus to data structures, connecting me with Ian Munro at Waterloo.

I am very grateful for all of the coauthors I have had the pleasure of collaborating with on the topics of this thesis. Corey, thank you for welcoming me into the lattice research project. I thoroughly enjoyed our brainstorming sessions and the two eureka moments, when we figured out how to efficiently compute order-testing and meets/joins. Sebastian, thank you for collaborating with me on lazy search trees. It took countless tries, but we were able to come up with a meaningful solution that generalizes priority queues with binary search trees. Hicham, Meng, and Yakov, thank you for the many fruitful whiteboard sessions, distant email and hangout chats, and the two papers that came out of our range mode project. Antonio and Nalin, thank you for working with me on minimum cuts and Yinzhan, I am very grateful you included me on faster dynamic range mode. I am extremely fortunate to have met you three through the vibrant competitive programming community, two of you via Richard Peng. Lastly, Danny, thank you for being an inspirational example in the area of data structure design. I have thoroughly enjoyed all our conversations and hope the future holds more of them.

This thesis would not be possible if not for a committee that has volunteered their precious time to give their thoughts and academic critiques of my research. Thank you Lap Chi and Trevor Brown, for your volunteering on this thesis and for your support at Waterloo. Prof. Joseph Cheriyan, thank you for volunteering on this thesis committee despite residing in a different academic department. Finally, thank you Prof. Pat Morin; despite having no interactions with me personally, you have volunteered to share your expertise regarding this document.

My interest in pursuing a research path was kindled due to early encounters with several inspiring individuals. Kerrick, I recall meeting you on the bus to our first ACM International Collegiate Programming Contest; you were programming mergesort in assembly. I have been very impressed with your intellect and our interactions encouraged me to further pursue mathematics and contests, which has led to this thesis. Simanta Mitra, thank you for all your support in ICPC and for encouraging me to pursue a PhD. Steve Butler, I would like to thank you for allowing me to join your combinatorics class late in the term at Iowa State. You introduced me to a beautiful area of mathematics that shaped my research trajectory and helped me publish my first paper. Hridesh Rajan, Steve Kautz, and Jim Lathrop, thank you for the discussions that led me to pursue a PhD. Sariel Har-Peled, thank you for conversing with a curious undergraduate to whom you had no formal relation. Lastly, I'd like to thank Bernhard Haeupler. In our brief experience together, you were an inspirational theoretical computer scientist to a young graduate student, but also a powerful role model in the way you live your life outside of research.

There are many more individuals that helped me succeed in my graduate school experience. Chung Webster, thank you for supporting me in my decision to pursue a PhD. Dieter van Melkebeek and Shuchi Chawla, thank you for your support at Wisconsin-Madison. Andrew Morgan, Menghui Wang, Rex Fernando, Ben Miller, Dimitris Paparas, Jinman Zhao, Yunfeng Li, Krittisak Chaiyakul, Ingkarat Rak-amnouykit, and Songwong Tasneeyapant, thank you for the many conversations and experiences we shared at Wisconsin-Madison. Gautam Kamath and Jochen Koenemann, thank you for the insightful conversations and for your support at Waterloo. Kevin Wu, Lingyi Zhang, and Azin Nazari, thank you for the many research discussions and also for the much-needed dinner breaks at some of Waterloo's better restaurants. Joachim von zur Gathen, thank you for writing me a graduate admissions recommendation letter. Colin Zheng, thank you for the mentorship and advice during my summer at Google.

Lastly, I'd like to thank my friends and family who have made graduate school more tolerable. Brett, Mike, Ronak, Chris, Ross, Meaghan, Eric, Ainura, Cameron, Saraid, Dan, Erin, Danny, Pasha, Joey, Devon, Evan, Natasha, Jeremy, Ying, Saranya, Satyam, Ian, Ryan B, Paul, Graham, Sam, Jan, Akshay, Ryan H, Amit, Beatrix, Sierra, Nolan, Jaehyun, Andy, Jordyn, Adam, Matt, Nick, Mike, Chris, Steve, Tony, and Allen, thank you for all of the good times we shared together and in particular for keeping me sane with online poker and Age of Empires while under the many covid-19 lockdowns. Finally, thank you to my brothers Jacob and Caleb, my sister Jess, and my mom and dad, who have listened to my adventure into deeper mathematics for seven years, supporting me throughout my journey.

Dedication

This thesis is dedicated to immigrants. Wherever they may be from, wherever they may choose to go, may they be welcome.

Statement of Contributions

This thesis is based on the following publications.

Chapter 2: [183] J. Ian Munro, Bryce Sandlund, and Corwin Sinnamon. Space-efficient data structures for lattices. In *17th Scandinavian Symposium and Workshop on Algorithm Theory*, 2020

Chapter 3: [209] Bryce Sandlund and Sebastian Wild. Lazy search trees. In *Proceedings of the 61st Annual Symposium on Foundations of Computer Science*, 2020

Chapter 4: [80] Hicham El-Zein, Meng He, J. Ian Munro, and Bryce Sandlund. Improved Time and Space Bounds for Dynamic Range Mode. In *26th Annual European Symposium on Algorithms*, volume 112, pages 25:1–25:13, 2018

Chapter 5: [210] Bryce Sandlund and Yinzhan Xu. Faster dynamic range mode. In *47th International Colloquium on Automata, Languages and Programming*, 2020

Chapter 6: [79] Hicham El-Zein, Meng He, J. Ian Munro, Yakov Nekrich, and Bryce Sandlund. On approximate range mode and range selection. In *30th International Symposium on Algorithms and Computation*, 2019

Chapter 7: [201] Richard Peng, Bryce Sandlund, and Daniel D. Sleator. Optimal offline dynamic 2,3-edge/vertex connectivity. In *Algorithms and Data Structures Symposium*, 2019

Chapter 8: [29] Nalin Bhardwaj, Antonio Molina Lovett, and Bryce Sandlund. A simple algorithm for minimum cuts in near-linear time. In *17th Scandinavian Symposium and Workshop on Algorithm Theory*, 2020

Table of Contents

List of Figures

Chapter 1

Introduction

The design of efficient data structures is a classic topic within the field of theoretical computer science. Efficient data structures are fundamental to efficient algorithms, but also frequently find use in applications directly, such as in database systems. Most programming language standard libraries contain implementations of basic data structures, and frequently in production code, these libraries see extensive use. In performance-critical applications, even very minor improvements in data structure efficiency can lead to very important performance gains. For these reasons, the design of efficient data structures is uniquely important to the theory and practice of computing systems.

This thesis focuses on the study of data structures from the perspective of the theoretician. By studying data structures with worst-case guarantees, fundamental mathematical properties can be discovered. In many circumstances, these theoretically superior data structures can lead to improved practical implementations.

Data structures are evaluated on three metrics:

1. The time complexity of the supported operations.

2. The space complexity of the data structure.

3. The time it takes to construct the data structure.

This thesis will consider all three metrics. The greatest attention will be given to time complexity of supported operations. In chapters 2, 4, and 6, the space complexity will be equally important. Construction costs are analyzed in chapters 2, 3, and 4.

Algorithms relating to data structures are also considered. In Chapter 7, an *offline* data structure is given. This data structure assumes access to the entire operation sequence in advance. In Chapter 8, an algorithm is given based on data structure concepts, which further abstracts necessary computation onto another data structure.

The topics discussed fall into three main categories: partial order data structures (Part I), range mode data structures (Part II), and graph and dynamic graph algorithms (Part III). While the topics discussed have relatively minor overlap, the technical intuition as well as guiding principles are shared.

In general, the solutions developed value simplicity over theoretical efficiency. Most of the main ideas of this thesis can be explained at an undergraduate computer science level. Further attention is given to practicality. This affects both the algorithms developed as well as the problems considered. The problems considered are some of the most natural: representing unknown total orders, computing modes, and computing minimum cuts. The less constrained the problem, the more likely it is to be of relevance.

With regards to computation model, we use both the word RAM and the pointer machine, depending on the chapter. In the word RAM, we can compute bitwise operations on $\Theta(\log n)$-bit words in $O(1)$ time (in this thesis we will always assume a word to be $\Theta(\log n)$ bits) and have random access to memory in constant time. In the pointer machine, we lose array access, instead representing data as a node with a constant number of pointers that can be followed in constant time. In general we assume arithmetic and other reasonable functions are computable in $O(1)$ time. We use the full power of the word RAM typically only indirectly, in succinct data structures in Chapter 6 and van Emde Boas trees [230] in Chapter 4. In particular, we do not make use of bit tricks to shave off partial logarithmic or doubly logarithmic factors. We do this to maintain relative practicality. The black-box data structures that do use such tricks can be substituted, if desired, with slower comparison-based data structures on the pointer machine. When the word RAM is used, most of our results give polynomial $\Omega(n^c)$ for $0 < c < 1$ improvements which are not affected by the power of word RAM, which can give at most a $\Theta(\log n)$ speedup.

We discuss the specific contents of this thesis in the following three sections.

1.1 Partial Order Data Structures

The representation of partial orders is at the heart of data structure design. Binary search trees, perhaps the most fundamental data structure in computer science, represent an initially unknown total order. Directed acyclic graphs can be interpreted as representations of partial orders by their reachability relations. In algebra, we can create partial orders on common sets such as the integers, by considering divisibility relations, or subsets of a universe, ordering by inclusion. We consider data structures to represent unknown total orders in Chapter 3 and data structures to represent lattices, such as those just mentioned, in Chapter 2.

1.1.1 Lattice Data Structures

We first consider data structures to represent a specific type of partial order known as a lattice. A lattice is a partial order such that for every pair of elements x and y, the set of elements greater than both x and y are all greater than (or equal to) a unique element $x \vee y$, known as the *join* of x and y. Similarly, the set of elements less than both x and y are all less than (or equal to) a unique element $x \wedge y$, known as the *meet* of x and y. Lattices arise in many areas of mathematics, the social sciences, and programming languages [122, 235, 109, 180, 191, 7, 49, 50, 165]. The integers, ordered by divisibility, are a lattice, as are a collection of subsets closed under intersection and union, ordered by inclusion.

The lattice problem asks to create a data structure to represent lattices that can efficiently determine if an order relation exists between two elements x and y and to compute the meet and join of x and y. The goal is to use as little space as possible.

Initial work in lattice data structures was heuristic [7]. The problem was then studied theoretically by Talamo and Vocca in a series of papers in the 90's [221, 222, 223]. Unfortunately, the data structures proposed by Talamo and Vocca have a fundamental error that appears irreparable. We study the issue with their work and present a data structure that takes $O(n\sqrt{n})$ words of space ($O(n\sqrt{n}\log n)$ bits), computes order relations in $O(1)$ time, and computes meets and joins in $O(n^{3/4})$ time. Here we assume a word RAM, where our use of this model is only to get $O(1)$ time dictionary lookup. We also give time-space tradeoffs, construction costs, and a data structure with complexity characterized by a concept of degree on a minimal graph representation of the partial order. This is discussed in Chapter 2.

1.1.2 Lazy Search Trees

Binary search trees are perhaps the most fundamental data structure in computer science. Research on binary search trees (BSTs) has primarily focused on developing BSTs with efficient *access* times. Specifically, dynamic optimality research looks for a BST that performs any access sequence in time within a constant factor of the offline-optimal BST. Although $\Omega(\log n)$ is the worst-case time complexity of a single access, on an operation sequence such as repeatedly accessing a single element, binary search trees can perform this sequence in about $O(1)$ time per access by storing said element at or near the root. This subarea has received vast attention [8, 67, 68, 143, 35, 218, 234, 164, 52, 142, 16].

Instead of focusing on efficient access times when possible, lazy search trees consider the problem of achieving efficient *insertion* times when the operation sequence permits such efficiency. The main insight is that priority queues have been developed with $O(1)$ time insertion and decrease-key operations [99, 101, 53, 46, 82, 128, 41, 129]; thus, on certain operation sequences (like when we only query for the minimum element), it is possible to support insertion in $o(\log n)$ time. Despite the plethora of research on BSTs with efficient access times, our work is the first to ask whether a data structure supporting the operations of a BST can achieve efficient insertion time. Our data structure is not a binary search tree, however. In order to avoid $\Omega(\log n)$ insert time, we delay sorting of inserted elements, employing structures similar to those used in priority queue literature, operating over a pointer machine. We don't see this as a disadvantage; the goal of the algorithm designer is to solve *problems* efficiently; in this case we observe that constraining to a model of binary search trees is an arbitrary limitation to the most efficient solutions.

We analyze the performance of our data structure based on a partition of current elements into a set of *gaps* $\{\Delta_i\}$ based on rank. A query falls into a particular gap and *splits* the gap into two new gaps at a rank r associated with the query operation (BST queries such as rank, select, membership, predecessor, successor, minimum, and maximum are supported). If we define $B = \sum_i |\Delta_i| \log_2(n/|\Delta_i|)$, our performance over a sequence of n insertions and q distinct queries is $O(B + \min(n \log\log n, n \log q))$. We show B is a lower bound.

The bound B satisfies $B = O(n \log q)$; when queries are non-uniform, better bounds are possible. In the priority queue case, our data structure achieves $O(\log\log n)$ time insertion and decrease-key operations, thus interpolating between binary search trees and priority queues. We show that we can achieve this while simultaneously achieving the desirable efficient access properties that have been studied in dynamic optimality literature. We discuss precise performance guarantees and other useful properties of lazy search trees in Chapter 3.

1.2 Range Mode

In the range mode problem, we are given a sequence of elements and must answer queries that request the most frequent element in a contiguous subsequence. If the problem is dynamic, we must also support insertion and deletion of elements in the sequence.

The mode is one of the three fundamental data statistics, along with median and mean, and range mode is one of few basic range problems that have not been solved optimally. Indeed, upper bounds remain quite large: linear space static range mode requires time about $O(\sqrt{n})$ per query and dynamic range mode about $O(n^{2/3})$ per operation [55], compared to range problems such as sum, minimum, median, or majority, which permit logarithmic time static solutions and polylogarithmic time or better dynamic solutions, even when space is restricted to $O(n)$ words [102, 44, 83, 200, 57, 131, 104, 83].

We consider a dynamic range mode data structure that slightly improves operation time while simultaneously improving space usage in Chapter 4. In Chapter 5, we break the $O(n^{2/3})$ time per operation barrier for dynamic range mode. In Chapter 6, we consider succinct data structures for approximate static range mode, approximate dynamic range mode, and approximate static range selection. All data structures in this part of the thesis operate on the word RAM. In Chapter 4, we use only standard operations with constant time array access as well as more powerful blackbox word RAM data structures non-central to our results. In Chapter 5 we measure performance ignoring logarithmic factors, so this distinction is not important. Chapter 6 requires the word RAM model more centrally.

1.2.1 Space-Efficient Dynamic Range Mode

The first dynamic range mode data structure was proposed by Chan et al. [55] and operated on a time-space continuum, achieving $O(n^{3/4} \log n / \log \log n)$ worst-case time query and $O(n^{3/4} \log \log n)$ amortized expected time update at $O(n)$ words of space and $O(n^{2/3} \log n / \log \log n)$ worst-case time query and amortized expected time update at $O(n^{4/3})$ words of space.

We improve both operation time and space usage by providing a data structure with $O(n^{2/3})$ worst-case time query and update while occupying $O(n)$ words of space. Our data structure can also be generalized to find the least frequent element in a range or an element of a particular frequency, with some restrictions. We discuss these data structures in Chapter 4.

1.2.2 Faster Dynamic Range Mode

Several different approaches can achieve a dynamic range mode data structure with $\tilde{O}(n^{2/3})$ time operations, where the $\tilde{O}(\cdot)$ notation hides logarithmic factors. A conditional lower bound from the online matrix-vector multiplication problem [132] states that a dynamic range mode data structure taking $O(n^{1/2-\epsilon})$ time per operation for $\epsilon > 0$ is not possible with current knowledge [55]. This leaves a gap of about $\Theta(n^{1/6})$ between the lower and upper bound.

We tighten this gap and break the $O(n^{2/3})$ time per operation barrier by presenting a data structure with per-operation complexity of about $\tilde{O}(n^{.656})$. Our result reduces to a Min-Plus product, which in turn reduces to fast matrix multiplication. We discuss this work in Chapter 5.

1.2.3 Approximate Succinct Range Mode and Range Selection

For any $\varepsilon \in (0,1)$, fixed at construction time, a $(1+\varepsilon)$-approximate range mode query asks for the position of an element whose frequency in the query range is at most a factor $(1+\varepsilon)$ smaller than the true mode. The best static approximate range mode data structure prior to our work was a result by Greve et al. [123] which achieves $O(\log(1/\epsilon))$ query time while using $O(n/\varepsilon)$ words of space. We improve this result by moving to a succinct encoding model, where we build the data structure off the input, then require only the data structure to answer queries. Our data structure achieves $O(\log(1/\epsilon))$ query time while using only $O(n/\varepsilon)$ *bits* of space, which is a $\Theta(\log n)$ factor reduction in space usage.

We also consider dynamic approximate range mode. Our data structure is the first dynamic approximate range mode data structure. For any fixed constant ε, it achieves $O(\log n/\log\log n)$ query time, $O(\log n)$ update time, and occupies $O(n)$ words of space. Finally, we consider approximate range selection. We discuss these results in Chapter 6.

1.3 Graph and Dynamic Graph Algorithms

The final part of this thesis is dedicated to graph and dynamic graph algorithms. There has been a large amount of recent work on data structures to compute queries in dynamic graphs, particularly connectivity queries [84, 85, 86, 148, 137, 138, 141, 149, 172, 226, 241]. Connectivity has been studied both because it is fundamental and simple, with the latter property permitting efficient dynamic solutions. Connectivity can be extended to c-edge connectivity, which queries the existence of c edge-disjoint paths between two vertices of a graph, or c-vertex connectivity, which queries the existence of c vertex-disjoint paths. Algorithms and data structures for these higher versions of connectivity are related to the notion of a *graph cut*, which is a set of edges or vertices whose removal disconnects a graph. Graph cuts are fundamental to the minimum cut problem, to which an algorithm we originally devised for a dynamic graph data structure provides a simpler near-linear time weighted minimum cut algorithm. We study an offline dynamic data structure for $c \leq 3$ edge and vertex connectivity in Chapter 7 and a simpler algorithm for the minimum cut problem in Chapter 8.

1.3.1 Offline Dynamic Higher Connectivity

Consider the problem of answering 2- and 3-edge and vertex connectivity queries in a graph undergoing edge insertion and deletion. In the typical data structure model, a query must be answered before the next operation is given. In this setting, known as the *online* model, data structures for 3-edge and 3-vertex connectivity require $O(n^{2/3})$ and $O(n)$ time per update, respectively [85]. Consider, however, a situation where the complete sequence of updates and queries is known in advance. This may happen directly, when data about a dynamic network is collected but not analyzed until a later point in time, or indirectly, if the algorithm is used as a subroutine of a different algorithm. In this setting, known as the *offline* setting, simpler and more efficient algorithms are possible. We give such an offline data structure for $2, 3$-edge/vertex connectivity that takes $O(\log n)$ time per operation, which is optimal due to the dynamic connectivity lower bound of Patrascu and Demaine [197]. We discuss this in Chapter 7.

1.3.2 Minimum Cuts

The weighted minimum cut problem on an undirected graph G asks to partition the vertices of G into two sets such that the total weight of edges crossing the partition is minimized. This problem is fundamental in graph optimization and has seen vast attention [153, 103, 205, 150, 155, 121, 228, 145, 59, 133, 51, 152, 113, 158, 134, 64, 112, 187, 220, 114].

At the time of our first study of the problem, the state-of-the-art weighted minimum cut algorithm was a result of Karger which finds a minimum cut on a graph with n vertices and m edges in $O(m \log^3 n)$ time [153]. Karger's algorithm leverages random sampling and tree packing to reduce the problem to finding a minimum cut that cuts at most two edges of a spanning tree T of a graph G. Karger then employs a complicated dynamic program which solves the reduced problem in $O(m \log^2 n)$ time.

We observed that a routine we originally devised for dynamic graph data structures simplified Karger's dynamic program to a straightforward edge iteration utilizing heavy-light decomposition [218]. With the use of a top tree [10], we achieve $O(m \log^2 n)$ complexity on the subroutine, matching Karger's approach. If we want to avoid the use of advanced data structures, heavy-light decomposition can be used a second time with a standard augmented binary search tree to achieve $O(m \log^3 n)$ time complexity. We further condense Karger's algorithm to a self-contained version and implement it. We discuss this in Chapter 8.

Part I

Partial Order Data Structures

Chapter 2

Space-Efficient Data Structures for Lattices

2.1 Introduction

A lattice is a partially-ordered set with the property that for any pair of elements x and y, the set of all elements greater than or equal to both x and y must contain a unique minimal element less than all others in the set. This element is called the *join* (or *least upper bound*) of x and y. A similar condition holds for the set of all elements less than both x and y: It must contain a maximum element called the *meet* (or *greatest lower bound*) of x and y.

We consider lattices from the perspective of succinct data structures. This area of study is concerned with representing a combinatorial object in essentially the minimum number of bits while supporting the "natural" operations in constant time. The minimum number of bits required is the logarithm (base 2) of the number of such objects of size n, e.g. about 2n bits for a binary tree on n nodes. Succinct data structures have been very successful in dealing with trees, planar graphs, and arbitrary graphs. Our goal in this chapter is to broaden the horizon for succinct and space-efficient data structures and to move to more algebraic structures. There has indeed been progress in this direction with abelian groups [91] and distributive lattices [184]. We take another step here in studying space-efficient data structures for arbitrary finite lattices.

Lattices have a long and rich history spanning many disciplines. Existing at the intersection of order theory and abstract algebra, lattices arise naturally in virtually every area of mathematics [122]. The area of formal concept analysis is based on the notion of a *concept lattice*. These lattices have been studied since the 1980s [235] and have applications in linguistics, data mining, and knowledge management, among many others [109]. Lattices have also found numerous applications in the social sciences [180].

Within computer science, lattices are also important, particularly for programming languages. Lattice theory is the basis for many techniques in static analysis of programs, and thus has applications to compiler design. Dataflow analysis and abstract interpretation, two major areas of static analysis, rely on fixed-point computations on lattices to draw conclusions about the behaviour of a program [191].

Lattice operations appear in the problem of hierarchical encoding, which is relevant to implementing type inclusion for programming languages with multiple inheritance (among other applications) [7, 49, 50, 165]. Here the problem is to represent a partially-ordered set by assigning a short binary string to each element so that lattice-like operations can be implemented using bitwise operations on these strings. The goal is to minimize the length of the strings for the sake of time and space efficiency.

In short, lattices are pervasive and worthy of study. From a data structures perspective, the natural question follows: How do we represent a lattice so that not too much space is required and basic operations like partial order testing, meet, and join can be performed quickly?

It was proven by Klotz and Lucht [161] that the number of different lattices on n elements is at least $2^{\Omega(n^{3/2})}$, and an upper bound of $2^{O(n^{3/2})}$ was shown by Kleitman and Winston [160]. Thus, any representation for lattices must use $\Omega(n^{3/2})$ bits in the worst case, and this lower bound is tight within a constant factor. We should then expect a data structure for lattices to use comparably little space.

Two naive solutions suggest themselves immediately. First, we could simply build a table containing the meet and join of every pair of elements in the given lattice. Any simple lattice operation could be performed in constant time. However, the space usage would be quadratic — a good deal larger than the lower bound. Alternatively, we could store only the transitive reduction graph of the lattice. This method turns out to be quite space-efficient: Since the transitive reduction graph of a lattice can only have $O(n^{3/2})$ edges [161, 240], the graph can be stored in $O(n^{3/2} \log n)$ bits of space; thus, the space complexity lies within a $\Theta(\log n)$ factor of the lower bound. However, the lattice operations become extremely slow as they require exhaustively searching through the graph. Indeed, it is not easy to come up with a data structure for lattices that uses less than quadratic space while answering meet, join, and partial order queries in less than linear time in the worst case.

The construction of a lattice data structure with good worst-case behaviour also has attractive connections to the more general problem of reachability in directed acyclic graphs (DAGs). Through its transitive reduction graph, a lattice can be viewed as a special type of DAG. Among other things, this chapter shows that we can support reachability queries in constant time for this class of graphs while using subquadratic space. Most classes of

DAGs for which this has been achieved, such as planar DAGs [227], permit a strong bound on the order dimension of the DAGs within that class. Order dimension of a partial order P is the minimum number of total orders $T_1, \ldots, T_\ell$ such that $x < y$ in P if and only if $x < y$ in each of $T_1, \ldots, T_\ell$. Lattices do not permit a strong bound on order dimension; they can have order dimension linear in the size of the lattice. A long-standing difficult problem in this line of research is to show a similar nontrivial result for the case of arbitrary sparse DAGs [199].

There has been significant progress in representation of *distributive* lattices, an especially common and important class of lattices. Space-efficient data structures for distributive lattices have been established since the 1990s [126, 127] and have been studied most recently by Munro and Sinnamon [184]. Munro and Sinnamon show that it is possible to represent a distributive lattice on n elements using $O(n \log n)$ bits of space while supporting meet and join operations (and thus partial order testing) in $O(\log n)$ time. This comes within a $\Theta(\log n)$ factor of the space lower bound by enumeration: As the number of distributive lattices on n elements is $2^{\Theta(n)}$ [87], at least $\Theta(n)$ bits of space are required for any representation.

The problem of developing a space-efficient data structure for arbitrary lattices was first studied by Talamo and Vocca in 1994, 1997, and 1999 [221, 222, 223]. They claimed to have an $O(n^{3/2} \log n)$-bit data structure that supports partial order queries in constant time and meet and join operations in $O(\sqrt{n})$ time. However, there is a nontrivial error in the details of their structure. Although much of the data structure is correct, we believe that this mistake is a critical flaw that is not easily repaired.

To our knowledge, no other data structures have been proposed that can perform lattice operations efficiently while using less than quadratic space. Our primary motivation is to fill this gap.

2.2 Contributions

Drawing on ideas from [223], we present new data structures for lattices that are simple, efficient for the natural lattice operations, and nearly optimal in space complexity. Our data structures support three queries:

- **Test Order:** Given two elements x and y, determine whether $x \leq y$ in the lattice order.
- **Find Meet:** Find the meet of two elements.
- **Find Join:** Find the join of two elements.

Our first data structure (Theorem 9) is based on a two-level decomposition of a lattice into many smaller lattices. It tests the order between any two elements in $O(1)$ time and answers meet and join queries in $O(n^{3/4})$ time in the worst case. It uses $O(n^{3/2})$ words of space[1], which is a $\Theta(\log n)$ factor from the known lower bound of $\Omega(n^{3/2})$ bits. The preprocessing time is $O(n^2)$.

We generalize this structure (Corollary 10) to allow for a tradeoff between the time and space requirements. For any $c \in [\frac{1}{2}, 1]$, we give a data structure that supports meet and join operations in $O(n^{1-c/2})$ time, occupies $O(n^{1+c})$ space, and can be constructed in $O(n^2 + n^{1+3c/2})$ time. At $c = 1/2$, it coincides with the first data structure.

Taking a different approach to computing meets and joins, we present another data structure (Theorem 12) based on a recursive decomposition of the lattice. Here the operational complexity is parameterized by the maximum degree d of any element in the lattice, where the degree is defined in reference to the transitive reduction graph of the lattice. This structure answers meet and join queries in $O(d\frac{\log n}{\log d})$ time, which improves significantly on the first data structure when applied to lattices with low degree elements (as is the case for distributive lattices, for example). It uses $O(n^{3/2})$ space.

This chapter is organized as follows. In Section 2.3, we give the necessary definitions and notation used throughout the chapter. In Section 2.4, we give the main tool we use to decompose a lattice, which we call a block decomposition. Section 2.5 describes the order-testing data structure and Section 2.6 extends this data structure to compute meets and joins. Some details of the preprocessing are left to Section 2.7. Section 2.8 contains our recursive degree-bounded data structure. In Section 2.9, we discuss the error in the papers [222, 223] and give some evidence of why it may be irreparable.

2.3 Preliminaries

Given a partially-ordered set (poset) $(P, \leq)$, we define the *downset* of an element $x \in P$ by $\downarrow x = \{z \in P \mid z \leq x\}$ and the *upset* of x by $\uparrow x = \{z \in P \mid z \geq x\}$.

Definition 1. *A* lattice *is a partially-ordered set* $(L, \leq)$ *in which every pair of elements has a* meet *and a* join*.*

The meet *of x and y, denoted $x \wedge y$, is the unique maximal element of $\downarrow x \cap \downarrow y$ with respect to $\leq$. Similarly, the* join *of x and y, denoted $x \vee y$, is the unique minimal element of $\uparrow x \cap \uparrow y$.*

[1]We assume a word RAM model with $\Theta(\log n)$-bit words. Henceforth, unless bits are specified, "$f(n)$ space" means $f(n)$ words of size $\Theta(\log n)$.

Meet ($\wedge$) and join ($\vee$) are also called greatest lower bound (GLB) and least upper bound (LUB), respectively. Lattices have the following elementary properties [66]. Let $x, y, z \in L$.

- The meet and join operations are idempotent, associative, and commutative:

$$\begin{array}{lll} x \vee x = x & x \vee (y \vee z) = (x \vee y) \vee z & x \vee y = y \vee x \\ x \wedge x = x & x \wedge (y \wedge z) = (x \wedge y) \wedge z & x \wedge y = y \wedge x \end{array}$$

- If $x \leq y$, then $x \wedge y = x$ and $x \vee y = y$.
- If $z \leq x$ and $z \leq y$, then $z \leq x \wedge y$. If $z \geq x$ and $z \geq y$, then $z \geq x \vee y$.
- A lattice must have a unique *top* element above all others and unique *bottom* element below all others in the lattice order.

Moreover, meet and join are dual operations. If the lattice is flipped upside-down, then meet become join and vice versa.

In this chapter, we prefer to work with *partial lattices*. A partial lattice is the same as a lattice except that it does not necessarily have top or bottom elements. Thus, the meet or join of two elements may not exist in a partial lattice; we use the symbol `null` to indicate this. We write $x \wedge y =$ `null` if $\downarrow x \cap \downarrow y = \varnothing$ and $x \vee y =$ `null` if $\uparrow x \cap \uparrow y = \varnothing$. Note that in a partial lattice the meet or join of x and y may not exist, but when they do exist they must be unique.

Equivalently, a partial lattice is a partially-ordered set satisfying the *lattice property*: If there are four elements x_1, x_2, y_1, and y_2 such that $x_1, x_2 < y_1, y_2$, then there must exists an intermediate element z with $x_1, x_2 \leq z \leq y_1, y_2$. See Figure 2.1. This statement trivially follows from the definition of a lattice; it only says that there cannot be multiple maximal elements in $\downarrow y_1 \cap \downarrow y_2$ or multiple minimal elements in $\uparrow x_1 \cap \uparrow x_2$.

Henceforth, we use the term "lattice" to mean "partial lattice". The difference is trivial in a practical sense, and our results are easier to express when we only consider partial lattices.

We assume that any lattice we wish to represent is given initially its *transitive reduction graph* (TRG). This is a directed acyclic graph (DAG) having a node for each lattice element and an edge (u, v) whenever $u < v$ and there is no intermediate node w such that $u < w < v$. The edge relation of this graph is called the *covering* relation: Whenever (u, v) is an edge of the TRG we say that v *covers* u.

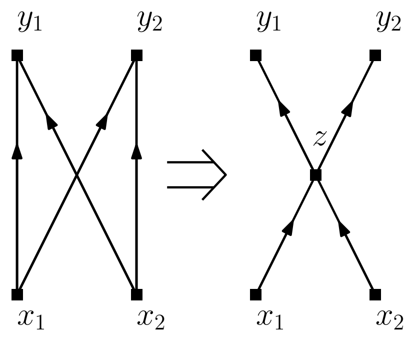

Figure 2.1: The configuration on the left cannot exist in a lattice for any nodes x_1, x_2, y_1, and y_2. There must be a node z between them as shown. We refer to this as the *lattice property.*

2.4 Block Decompositions

The main tool used in our data structure is called a block decomposition of a lattice. It is closely based on techniques used by Talamo and Vocca in [222, 223].

Let L be a lattice with n elements. A block decomposition of L is a partition of the elements of L into subsets called *blocks*. The blocks are chosen algorithmically using the following method. We first specify a positive integer k to be the *block size* of the decomposition (our application will use the block size $\sqrt{n}$). Then we label the elements of L as "fat" or "thin" according to the sizes of their downsets. A fat node is "minimal" if all elements in its downset, except itself, are thin. Formally:

Definition 2. *A node $x \in L$ is called* fat *if $|{\downarrow}x| \geq k$, and x is called* thin *if $|{\downarrow}x| < k$. We say x is a* minimal fat node *if x is fat and every other node in ${\downarrow}x$ is thin.*

Minimal fat nodes are the basis for choosing blocks, which is done as follows. While there exists a minimal fat node h in the lattice, create a new *principal* block B containing the elements of ${\downarrow}h$, and then delete those nodes from the lattice. The node h is called the *block header* of B.

Deleting the elements of B may cause some fat nodes to become thin by removing elements from their downsets; this should be accounted for before choosing the next block. When there are no fat nodes in the lattice, put the remaining elements into a single block B_{res} called the *residual block.*

This method creates a set of principal blocks $\{B_1, B_2, \ldots, B_m\}$ and a residual block B_{res}. Each principal block B_i has a block header h_i, which was the minimal fat node used to create B_i. A block header is always the top element within its block. The residual

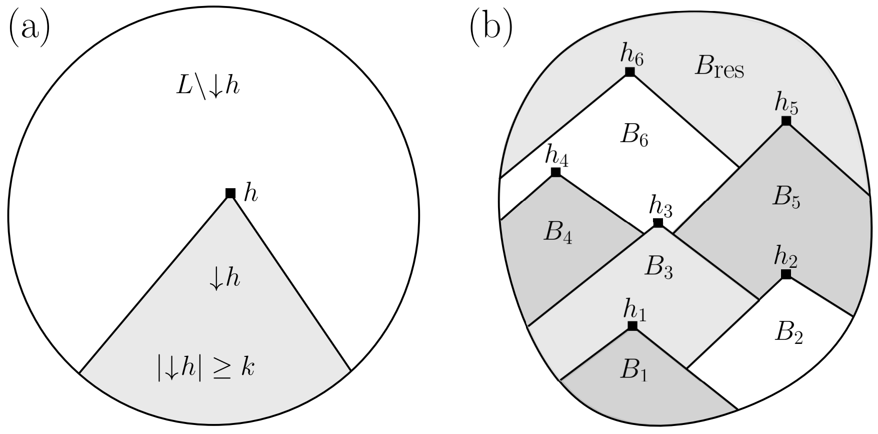

Figure 2.2: (a) A minimal fat node h is used as a block header during the decomposition. The downset of h is removed and the process repeats on $L \setminus \downarrow h$. (b) A block decomposition yields a set of disjoint principal blocks, each having a block header. The residual block consists of the lattice elements that are not below any block header.

block may or may not have a top element, but it is not considered to have a block header regardless. Figure 2.2 shows a full block decomposition.

The block decomposition algorithm is summarized in Algorithm 1; it will be shown later that this algorithm can be implemented to run in $O(n^{7/4})$ time, where n is the number of elements in the lattice.

Algorithm 1 Block Decomposition (Intuitive Version)

Require: A partial lattice L on n elements and a positive integer k.
Ensure: A block decomposition of L with block size k.

```
1: $i = 1$
2: while there exists a minimal fat node $h$ do
3:     $B_i = \downarrow h \cap L$
4:     $L = L \setminus B_i$
5:     $i = i + 1$
6: $B_{\text{res}} = L$
```

Properties of Block Decompositions

Let us note some elementary properties of block decompositions. Let L be a lattice with n elements.

- Every element of the lattice lies in exactly one block.
- There can be at most n/k principal blocks as each one has size between k and n. Consequently, there are at most n/k block headers.
- Since the block headers are chosen to be *minimal* fat nodes, every other element is thin relative to the block it lies in. That is, if x lies in a block B and x is not the block header of B, then $|\downarrow x \cap B| < k$.

The last fact motivates the following term, which we will use frequently.

Definition 3. *The* local downset *of an element x is the set $\downarrow x \cap B$, where B is the block containing x.*

Restated, the last property listed above says that the local downset of any element that is not a block header has size less than k. We also note that if h is the block header of a principal block B, then the local downset of h is B.

Somewhat less obvious is the following lemma.

Lemma 4. *Every block is a partial lattice.*[2]

Proof. The lemma follows from two facts.

1. The downset of any element in a partial lattice is also a partial lattice.
2. If the downset of an element is removed from a partial lattice, then the remaining elements still form a partial lattice.

We prove the first fact. Let h be an element of a partial lattice L. We prove that the poset $\downarrow h$ satisfies the lattice property (see Figure 2.1). Suppose there are four elements $x_1, x_2, y_1, y_2 \in \downarrow h$ such that $x_1, x_2 < y_1, y_2$. These elements also lie in L, and since L is a lattice there must be an element $z \in L$ such that $x_1, x_2 \leq z \leq y_1, y_2$. As $z \leq y_1 \leq h$, z must lie in $\downarrow h$. Thus $\downarrow h$ is a partial lattice because it satisfies the lattice property.

The second fact is similar. Suppose $\downarrow h$ is removed from a partial lattice L. If there are four elements $x_1, x_2, y_1, y_2 \in L \setminus \downarrow h$ with $x_1, x_2 < y_1, y_2$, then there must be an element $z \in L$ with $x_1, x_2 \leq z \leq y_1, y_2$. This element z cannot lie in $\downarrow h$ because $x_1 \leq z$ and $x_1 \notin \downarrow h$. Therefore $z \in L \setminus \downarrow h$. $\square$

[2]Here the partial order on a block is inherited from the order on L.

Remark 5. To avoid confusion in our notation, all lattice relations and operators are assumed to be with respect to L. In particular, $\wedge$, $\vee$, $\uparrow$, and $\downarrow$ always reference the full lattice and are not restricted to a single block.

Intuition for Block Decompositions

We can now explain intuitively why a block decomposition is a good idea and how it leads to an effective data structure. Lemma 4 means that the blocks can be treated as independent partial lattices. Moreover, the elements within each block are all thin, with the noteworthy exception of the block headers. For any single block, this thinness condition makes it possible to create a fast, simple, space-efficient data structure that facilitates computations within that block. However, such a data structure only contains local information about its block; it cannot handle operations that span multiple blocks.

For those operations, we rely upon the block headers to bridge the gaps. The block headers are significant because they induce a *unique representative property* on the blocks: If h is the block header of some principal block B and x is some element of the lattice, then we think of $x \wedge h$ as the representative of x in block B. For all of the operations that we care about, the representative of x in B faithfully serves the role of x during computations within B.

Combining the power of the unique representative property with our ability to quickly perform block-local operations gives us an effective data structure for lattices, which we are now prepared to describe.

2.5 A Data Structure for Order Testing

First, we describe a simple data structure that performs order-testing queries (answers "Is $x \le y$?") in constant time. We later extend it to handle meet and join queries as well.

Given a partial lattice L with n elements, we perform a block decomposition on L using the block size $k = \sqrt{n}$. Let $B_1, B_2, \ldots, B_m$, and B_{res} be the blocks of this decomposition and $h_1, \ldots, h_m$ be the block headers. Note that $m \le \sqrt{n}$.

Information Stored

We represent each element of L by a node with two fields.[3] One field contains a unique *identifier* for the lattice element, a number between 0 and $n-1$, for indexing purposes. The other field indicates the block that the element belongs to.

[3]We often use the term "node" to refer to the element of L that the node represents.

Our data structure consists of ($\mathcal{A}$), a collection of arrays, and ($\mathcal{B}$), a collection of dictionaries.

($\mathcal{A}$) For each block header h_i, we store an array containing a pointer to the node $h_i \wedge x$ for each $x \in L$. The meet of any node with any block header can be found with one access to the appropriate array.

($\mathcal{B}$) For each $x \in L$ we store a dictionary $\text{DOWN}(x)$ containing the identifiers of all the nodes in the local downset of x. By using a space-efficient static dictionary (e.g. [47]), membership queries can be performed in constant time. With this, we can test the order between any two nodes in the same block in constant time.

Testing Whether $x \leq y$

Given nodes x and y in L, we can test whether $x \leq y$ in three cases.

Case 1: If x is in a principal block B_i, then find $y_i = h_i \wedge y$ using ($\mathcal{A}$). If $y_i \in B_i$, then $x \leq y$ if and only if x is a member of $\text{DOWN}(y_i)$; this can be tested using ($\mathcal{B}$). If $y_i \notin B_i$, then $x \nleq y$.

Case 2: If $x \in B_{\text{res}}$ and $y \in B_{\text{res}}$, then $x \leq y$ if and only if x is a member of $\text{DOWN}(y)$.

Case 3: If $x \in B_{\text{res}}$ and $y \notin B_{\text{res}}$, then $x \nleq y$.

The three cases can be tested in constant time using ($\mathcal{A}$) and ($\mathcal{B}$).

Proposition 6. *The above method correctly answers order queries.*

Proof. Clearly the three cases cover all possibilities for x and y.

In Case 1, $y_i = h_i \wedge y$ has the property that $x \leq y$ if and only if $x \leq y_i$. This property holds because $x \leq h_i$ by assumption, and by the definition of meet,

$$x \leq h_i \wedge y \text{ if and only if } x \leq h_i \text{ and } x \leq y.$$

If $y_i \in B_i$, then the order can be tested directly using $\text{DOWN}(y_i)$. If $y_i \notin B_i$, then y_i cannot be above x in the lattice because $y_i \leq h_i$ and every element between x and h_i must lie in B_i.

Case 2 is checked directly using ($\mathcal{B}$).

Case 3 is correct because B_{res} consists of all elements that are not below any block header. As y is in some principal block, it must lie below some block header. Hence, x cannot be below y. $\square$

Space Complexity

Storing the n nodes of the lattice requires $\Theta(n)$ space. Each array of $(\mathcal{A})$ requires $\Theta(n)$ space and there are at most $\sqrt{n}$ block headers, yielding $O(n^{3/2})$ space in total.

Assuming $(\mathcal{B})$ uses a succinct static dictionary (see [47]), the space usage for $(\mathcal{B})$ will be proportional to the sum of $|{\downarrow} x \cap B_x|$ over all $x \in L$, where B_x is the block containing x. If x is not a block header, then $|{\downarrow} x \cap B_x| < \sqrt{n}$ because the local downsets must be smaller than the block size of the decomposition. There are $n - m$ such elements, as m denotes the number of principal blocks. If x is a block header, then ${\downarrow} x \cap B_x = B_x$. Thus

$$\sum_{x \in L} |{\downarrow} x \cap B_x| \le (n-m)\sqrt{n} + \sum_{i=1}^{m} |B_i| \le (n-m)\sqrt{n} + n \le 2n^{3/2}.$$

The total space for the data structure is therefore $O(n^{3/2})$.

2.6 Finding Meets and Joins

We now extend the order-testing data structure of the last section to answer meet queries: Given two elements x and y in L, we wish to find $x \wedge y$. Our data structure can answer these queries in $O(n^{3/4})$ time.

Subblock Decompositions

Let B_i be a principal block with block header h_i. A *subblock decomposition* of B_i is simply a block decomposition of $B_i \setminus \{h_i\}$.

To state it explicitly, the subblock decomposition is a partition of $B_i \setminus h_i$ into a set of principal subblocks $\{S_{i,1}, S_{i,2}, \ldots, S_{i,\ell_i}\}$, each having a subblock header $g_{i,j}$, and one residual subblock $S_{i,\text{res}}$. The decomposition strategy is identical to that of a block decomposition, and it still depends on a *subblock size* r that we specify.

We exclude h_i from the subblock decomposition as a convenience. We want to use the property that the local downsets of the elements in B_i have size less than $\sqrt{n}$, and this holds for every element of B_i except for h_i.

Obviously, the subblocks have the same properties as blocks.

- Each principal subblock $S_{i,j}$ is a subset of B_i with $|S_{i,j}| \ge r$. Hence, $\ell_i \le \frac{|B_i|}{r}$.
- If $x \in S_{i,j} \setminus \{g_{i,j}\}$ then $|{\downarrow} x \cap S_{i,j}| < r$.
- If $x \in S_{i,\text{res}}$ then $|{\downarrow} x \cap S_{i,\text{res}}| < r$.
- Each subblock is a partial lattice.

Extending the Data Structure

As before, let $B_1, B_2, \ldots, B_m$, and B_{res} be the blocks of the decomposition of L, each having size at least $\sqrt{n}$. Within each principal block B_i, we perform a subblock decomposition with subblock size $r = \sqrt{|B_i|}$, yielding subblocks $S_{i,1}, S_{i,2}, \ldots, S_{i,\ell_i}$, and $S_{i,\text{res}}$. We have $\ell_i \le \sqrt{|B_i|}$ for $1 \le i \le m$. There is a subblock header $g_{i,j}$ for each principal subblock $S_{i,j}$, $1 \le i \le m$ and $1 \le j \le \ell_i$.

Information Stored

We add a new field to each node that indicates which subblock contains it. We store $(\mathcal{A})$ and $(\mathcal{B})$ as in the order-testing structure, and additionally:

- $(\mathcal{C})$ For each subblock header $g_{i,j}$, we store an array containing a pointer to $g_{i,j} \wedge x$ for all $x \in B_i$. These arrays allow us to determine the meet of any subblock header and any node in the same block with a single access.
- $(\mathcal{D})$ For each principal subblock $S_{i,j}$, we store a table that contains the meet of each pair of elements from $S_{i,j}$, unless the meet lies outside $S_{i,j}$. That is, the table has $|S_{i,j}|^2$ entries indexed by pairs of elements in $S_{i,j}$. The entry for (x, y) contains a pointer to $x \wedge y$ if it lies in $S_{i,j}$, or `null` otherwise. We can compute meets within any principal subblock in constant time using these tables.
- $(\mathcal{E})$ For every element x in a residual subblock $S_{i,\text{res}}$, we store $\downarrow x \cap S_{i,\text{res}}$ as a linked list of pointers. This allows us to iterate through the local downset of each element in the residual subblock.

Finding the Meet

This data structure allows us to find the meet of two elements $x, y \in L$ in $O(n^{3/4})$ time. The meet-finding operation works by finding representative elements for x and y in each principal block and computing the meet of each pair of representatives. We call these *candidate meets* for x and y. Once the set of candidate meets is compiled, the algorithm finds the largest element among them (with respect to the lattice order) and returns it.

We refer to the algorithm as MEET. This algorithm uses a subroutine called MEET-IN-BLOCK that finds the meet of two elements from the same principal block, or else determines that the meet does not lie within that block. The subroutine is similar to the main procedure except that it works on the subblock level instead of the block level.

MEET: Given $x, y \in L$, find $x \wedge y$.

(a) Initialize an empty set Z to store candidate meets for x and y.

(b) *Check principal blocks:* For each principal block B_i, find the representative elements $x_i = x \wedge h_i$ and $y_i = y \wedge h_i$ using $(\mathcal{A})$. If $x_i \in B_i$ and $y_i \in B_i$, then use the subroutine MEET-IN-BLOCK to either find $x_i \wedge y_i$ or determine that B_i does not contain it. If $x_i \wedge y_i$ is found, then add it to Z.

(c) *Check residual block:* If x and y are both in the residual block B_{res}, then use DOWN(x) to iterate through every element $z \in {\downarrow} x \cap B_{\text{res}}$. Add z to Z whenever $z \leq y$.

(d) Using the order-testing operation, determine the maximum element in Z and return it. If Z is empty, then conclude that the meet of x and y does not exist and return `null`.

MEET-IN-BLOCK: Given $x_i, y_i \in B_i$, either find $x_i \wedge y_i \in B_i$ or determine that $x_i \wedge y_i \notin B_i$.

(a) If $x_i = h_i$ or $y_i = h_i$, then return the smaller of x_i and y_i. Otherwise, initialize an empty set Z_i to store candidate meets for x_i and y_i in B_i.

(b) *Check principal subblocks:* For each principal subblock $S_{i,j}$, find the representative elements $x_{i,j} = x_i \wedge g_{i,j}$ and $y_{i,j} = y_i \wedge g_{i,j}$ using $(\mathcal{C})$. If $x_{i,j}$ and $y_{i,j}$ are both in $S_{i,j}$, then look up

$$z_{i,j} = \begin{cases} x_{i,j} \wedge y_{i,j} & \text{if } x_{i,j} \wedge y_{i,j} \in S_{i,j} \\ \texttt{null} & \text{otherwise} \end{cases}$$

using the appropriate table in $(\mathcal{D})$. If $z_{i,j} \neq$ `null` then add it to Z_i.

(c) *Check residual subblock:* If x_i and y_i are both in the residual subblock $S_{i,\text{res}}$, then use $(\mathcal{E})$ to iterate through every element $z \in {\downarrow} x_i \cap S_{i,\text{res}}$. Add z to Z_i whenever $z \leq y_i$.

(d) Using the order-testing operation, determine the largest node in Z_i and return it. If Z_i is empty, then conclude that $x_i \wedge y_i \notin B_i$ and return `null`.

Correctness

We now prove that this algorithm is correct, beginning with the correctness of MEET-IN-BLOCK.

Lemma 7. MEET-IN-BLOCK *returns $x_i \wedge y_i$ if it lies in B_i and* `null` *otherwise.*

Proof. If $x_i = h_i$ or $y_i = h_i$, then $x_i \wedge y_i$ is returned in step (i). Otherwise, the correctness of the algorithm relies on two facts.

Fact 1. Every element $z \in Z_i$ satisfies $z \leq x_i \wedge y_i$.

Fact 2. If $x_i \wedge y_i$ exists and lies in B_i, then it is added to Z.

Assuming these hold, step (iv) must correctly answer the query: In the case that $x_i \wedge y_i \in B_i$, the meet must be added to Z_i and it must the maximum element among all elements in Z_i. If $x_i \wedge y_i \notin B_i$, then Z_i will be empty by the first fact.

Fact 1 is straightforward. Every candidate meet z added to Z_i in step (ii) is $x_{i,j} \wedge y_{i,j}$ for some $j \in \{1, \ldots, \ell_i\}$, as reported by $(\mathcal{D})$. Since $x_{i,j} \leq x_i$ and $y_{i,j} \leq y_i$ we have $z \leq x_i \wedge y_i$. When a candidate meet z is added to Z in step (iii) it is because $z \in \downarrow x_i \cap B_{\text{res}}$ and $z \leq y_i$; hence $z \leq x_i \wedge y_i$.

To prove Fact 2, first suppose that $x_i \wedge y_i$ lies in a principal subblock $S_{i,j}$. Then $x_i \wedge y_i \leq g_{i,j}$. By the elementary properties of the meet operation,

$$x_i \wedge y_i = x_i \wedge y_i \wedge g_{i,j} = (x_i \wedge g_{i,j}) \wedge (y_i \wedge g_{i,j}) = x_{i,j} \wedge y_{i,j}.$$

Thus, $x_i \wedge y_i$ is added to Z during step (ii) when the subblock $S_{i,j}$ is considered.

Now suppose that $x_i \wedge y_i$ lies in the residual subblock $S_{i,\text{res}}$. In this case, x_i and y_i must themselves lie in $S_{i,\text{res}}$, for if either one is below any subblock header of B_i then their meet would also be below that same block header. Thus, $x_i \wedge y_i$ will be added to Z_i in step (3) during which every element of $\downarrow x_i \cap \downarrow y_i \cap S_{i,\text{res}}$ is added to Z_i. This proves Fact 2. □

Lemma 8. MEET *finds $x \wedge y$ or correctly concludes that it does not exist.*

Proof. This proof is similar to that of Lemma 7. It relies on the same two facts.

Fact 1. Every element $z \in Z$ satisfies $z \leq x \wedge y$.

Fact 2. If $x \wedge y$ exists, then it is added to Z.

Assuming these hold, step (4) must correctly answer the query. The only significant difference between Meet and Meet-In-Block is the method of finding candidate meets in step (2). Meet calls Meet-In-Block to find $x_i \wedge y_i$ if it lies in B_i whereas Meet-In-Block uses $(\mathcal{D})$ to find $x_{i,j} \wedge y_{i,j}$ if it lies in $S_{i,j}$. By Lemma 7, Meet-In-Block accurately returns $x_i \wedge y_i$ if $x_i \wedge y_i \in B_i$ and `null` otherwise. Now Facts 1 and 2 may be proved by the same arguments. $\square$

Time Analysis

The meet procedure takes $O(n^{3/4})$ time in the worst case. We first analyze the time for Meet-In-Block applied to a principal block B_i. Step (i) takes constant time. Step (ii) takes constant time per principal subblock of B_i using $(\mathcal{C})$ and $(\mathcal{D})$. Since each principal subblock has size at least $\sqrt{|B_i|}$, there are at most $|B_i|/\sqrt{|B_i|} = \sqrt{|B_i|}$ principal subblocks; hence the time for step (ii) is $O(\sqrt{|B_i|})$. Step (iii) performs constant-time order testing on all the elements below x_i in the residual subblock. By the subblock decomposition method, there are at most $\sqrt{|B_i|}$ such elements.

When step (iv) is reached, Z_i has been populated with at most one element per principal subblock ($\sqrt{|B_i|}$ in total) and at most $\sqrt{|B_i|}$ elements from the residual sublock. The maximum element in Z_i is found in linear time during this step. Thus, Meet-In-Block runs in $O(\sqrt{|B_i|})$ time when applied to block B_i.

Now the main procedure can be analyzed in a similar fashion. Step (1) takes constant time. Step (2) calls Meet-In-Block on every principal block, and hence the total time for step (2) is proportional to $\sum_{i=1}^{m} \sqrt{|B_i|}$. By Jensen's inequality, $\sum_{i=1}^{m} \sqrt{|B_i|}$ is maximized when all the blocks have size $\sqrt{n}$, since each principal block has size at least $\sqrt{n}$ and $\sum_{i=1}^{m} |B_i| \leq n$. Thus

$$\sum_{i=1}^{m} \sqrt{|B_i|} \leq \sum_{i=1}^{\sqrt{n}} n^{1/4} \leq n^{3/4}.$$

As in the analysis of steps (iii) and (iv), steps (3) and (4) take $O(\sqrt{n})$ time. Therefore the time complexity of Meet is $O(n^{3/4})$.

Space Complexity

The space required to store the nodes, $(\mathcal{A})$, and $(\mathcal{B})$ is $O(n^{3/2})$ as in Section 2.5.

Fix $i \in \{1, \ldots, m\}$. We show that the parts of $(\mathcal{C})$, $(\mathcal{D})$, and $(\mathcal{E})$ relating to B_i occupy $O(|B_i|\sqrt{n})$ space. Since $\sum_{i=1}^{m} |B_i| \leq n$, it follows that the entire data structure takes $O(n^{3/2})$ space.

Each array in $(\mathcal{C})$ requires $O(|B_i|)$ space. There are at most $\sqrt{|B_i|}$ subblock headers for a total of $O(|B_i|^{3/2})$ space.

The lookup table in $(\mathcal{D})$ for subblock $S_{i,j}$ takes $O(|S_{i,j}|^2)$ space. Since $\sqrt{|B_i|} \leq |S_{i,j}| \leq \sqrt{n}$, we have $\sum_{j=1}^{\ell_i} |S_{i,j}|^2 \leq \sqrt{n} \sum_{j=1}^{\ell_i} |S_{i,j}|$. Notice $\sum_{j=1}^{\ell_i} |S_{i,j}| \leq |B_i|$ as the subblocks are disjoint subsets of B_i. Therefore the total space occupied by $(\mathcal{D})$ is $O(|B_i|\sqrt{n})$.

The lists stored by $(\mathcal{E})$ occupy $O(\sqrt{|B_i|})$ space each for a total of $O(|B_i|^{3/2})$ space. The space charged to block B_i is therefore $O(|B_i|^{3/2} + |B_i|\sqrt{n} + |B_i|^{3/2}) = O(|B_i|\sqrt{n})$.

Preprocessing

It remains to discuss how to efficiently decompose the lattice and initialize the structures $(\mathcal{A})$ — $(\mathcal{E})$. Recall that we the lattice is presented initially by its transitive reduction graph. It is known that the number of edges in the TRG of a lattice is $O(n^{3/2})$ [161, 240]. We assume that the TRG is stored as a set of n nodes, each with a list of its out-neighbours (nodes that cover it) and a list of in-neighbours (nodes that it covers). The total space needed for this representation is $O(n^{3/2})$. The preprocessing takes $O(n^2)$ time and the space usage never exceeds $O(n^{3/2})$.

The first step in preprocessing is to determine the block decomposition. The same technique will apply to subblock decompositions. We begin by computing a *linear extension* of the lattice. A linear extension of a partially-ordered set is an order of the elements $x_1, x_2, \ldots, x_n$ such that if $i \leq j$ then $x_j \nleq x_i$. A linear extension may be found by performing a topological sort on the TRG, which can be done in $O(n^{3/2})$ time [146].

We now visit each element of L in the order of this linear extension and determine the size of its downset. The size of the downset can be computed by a depth-first search beginning with the element and following edges descending the lattice. This search takes time proportional to the number of edges between elements in the downset. As soon as this process discovers a fat node h (a node with at least $\sqrt{n}$ elements in its downset), it can be used as a block header. Then h and every element of its downset can be deleted from L. The process of computing the sizes of the downsets can continue from the node following h in the linear extension, and the only difference is that the graph searches used to compute the size of each downset must now be restricted to $L \setminus {\downarrow} h$. There is no need to recompute the downset size of any node before h in the linear extension because the size of its downset was less than $\sqrt{n}$ previously and deleting ${\downarrow} h$ can only reduce this value. The fat nodes encountered in this way form the block headers of the decomposition. After every node has been visited, the remaining elements can be put into the residual block.

The time needed for the decomposition depends on the number of edges in each downset. By Lemma 4, every downset is a partial lattice, and thus a downset with k nodes can have only $O(k^{3/2})$ edges. For every thin node encountered, the number of edges in the downset is at most $O((\sqrt{n})^{3/2}) = O(n^{3/4})$ because it contains less than $\sqrt{n}$ elements. Thus, the time needed to visit all the thin nodes is $O(n^{7/4})$.

Whenever a fat node is discovered its downset is removed immediately, and so the edges visited during the DFS are never visited again. Hence, the time needed to examine all of the block headers is proportional to the number of edges in the whole TRG. Therefore a block decomposition can be computed in $O(n^{7/4})$ time.

By the same procedure, the subblocks can be computed in $O(\sum_{i=1}^{m} |B_i|^{7/4})$ time. Since $\sum_{i=1}^{m} |B_i| \leq n$, this is at most $O(n^{7/4})$.

With the block and subblock decompositions in hand, data structures $(\mathcal{A})$ — $(\mathcal{E})$ can be initialized. We explore this in the following section.

We have now proven the main theorem of this chapter.

Theorem 9. *There is a data structure for lattices that requires $O(n^{3/2})$ space, answers order-testing queries in $O(1)$ time, and computes the meet or join of two elements in $O(n^{3/4})$ time. The preprocessing time starting from the transitive reduction graph of the lattice is $O(n^2)$.*

A straightforward generalization of the data structure allows for a space-time tradeoff.

Corollary 10. *For any $c \in [\frac{1}{2}, 1]$, there is a data structure for lattices that requires $O(n^{1+c})$ space and computes the meet or join of two elements in $O(n^{1-c/2})$ time. The preprocessing time, starting from the transitive reduction graph of the lattice, is $O(n^2 + n^{1+3c/2})$.*

Proof. The modification is obtained by adjusting the block size of the initial decomposition from $\sqrt{n}$ to n^c. Otherwise, the data structure and methods are identical. The time, space, and preprocessing analyses are similar. □

Note that for $c = \frac{1}{2}$, this data structure is precisely that of Theorem 9.

2.7 Initializing the Data Structure

We now show how $(\mathcal{A})$, $(\mathcal{B})$, $(\mathcal{C})$, $(\mathcal{D})$, and $(\mathcal{E})$ can be constructed in $O(n^2)$ time. We assume that we have access to the TRG of the lattice and that the block and subblock decompositions have already been computed.

$(\mathcal{A})$ Some care is required to construct $(\mathcal{A})$ efficiently. Let $x_1, \ldots, x_n$ be a linear extension of L. Consider a principal block B_i with block header h_i. To find $z \wedge h_i$ for each $z \in L$, we do the following.

1. Initialize an array of length n to store the meet of h_i with each element and populate the array with `null` in every entry.
2. Perform a DFS to find $\downarrow h_i$ in L. Note that $\downarrow h_i$ may be considerably larger than B_i. Put the elements of $\downarrow h_i$ (note that this includes h_i) into a linear extension $y_1, \ldots, y_k$ by restricting the linear extension of L to these elements.
3. Traverse the nodes in reverse order of this extension (beginning with y_k and ending at y_1). For each node y_j, perform a DFS on the *upset* of that node in the full lattice. For every node z visited during the DFS for node y_j, record that $z \wedge h_i = y_j$ in the array, and then mark z so that it will not be visited by later graph searches. After all the nodes in $\downarrow h_i$ have been processed, restore the lattice by unmarking all nodes.

By this method, the entry for $z \wedge h_i$ in the array is recorded to be the last element in the linear extension of $\downarrow h_i$ that is below z. This must be the correct node because it is below both z and h_i, and every other element below z and h_i occurs earlier in the linear extension. Whenever $z \wedge h_i$ does not exist in the lattice, the array entry for $z \wedge h_i$ is the default value `null`.

The time for this procedure is bounded by the number of edges in the TRG for L because no node is visited more than once over all of the graph searches. Recall that the number of edges in the TRG is $O(n^{3/2})$. Summing over all block headers, the total time to create $(\mathcal{A})$ is at most $O(n^{3/2}\sqrt{n}) = O(n^2)$.

$(\mathcal{B})$ This can be computed by performing a DFS on the local downset of each node and adding the elements visited to a dictionary for that node.

Initializing and populating the space-efficient dictionary of [47] takes time linear in the number of dictionary entries. Excluding the block headers, the local downsets have at most $\sqrt{n}$ nodes and $O(n^{3/4})$ edges; hence the time spent on all non-block headers is at most $O(n^{7/4})$. The local downsets of the block headers are all disjoint, so the total time required is $O(n^{7/4})$.

$(\mathcal{C})$ Use the same method for $(\mathcal{A})$ restricted to each block to compute $(\mathcal{C})$. The total time is $O(\sum_{i=1}^{m} |B_i|^2)$, which is no larger than $O(n^2)$.

$(\mathcal{D})$ The method of $(\mathcal{A})$ can also be used to compute $(\mathcal{D})$. For each element z in a principal subblock $S_{i,j}$, find the meet of z with every other element in the subblock in $O(|S_{i,j}|^{3/2})$ time, where z plays the role of h_i in the method for $(\mathcal{A})$. It takes $O(|S_{i,j}|^{5/2})$ time to do this for every element in a single subblock and the total time is proportional to

$$\sum_{i=1}^{m}\sum_{j=1}^{\ell_i}|S_{i,j}|^{5/2} \le \sum_{i=1}^{m}\sum_{j=1}^{\ell_i}|S_{i,j}|(\sqrt{n})^{3/2} \le n^{7/4}.$$

The first inequality uses the fact that each subblock has size at most $\sqrt{n}$. The second inequality holds because the subblocks are disjoint.

($\mathcal{E}$) Each linked list can be constructed by performing a DFS on the downset of each element in a residual subblock. This takes $O(n^{7/4})$ time as in the analysis for ($\mathcal{B}$).

2.8 Degree-Bounded Extensions

Recall that we assume that the lattice is initially represented by its transitive reduction graph (TRG). Let the *degree* of a lattice node be the number of in-neighbours in the TRG, or equivalently, the number of nodes it covers. Interestingly, developing methods that handle high-degree nodes efficiently has been the primary obstacle to improving on our data structure. Indeed, the "dummy node" technique of Talamo and Vocca, explained in Section 2.9, is effectively used to get around high-degree lattice elements. We have found that meets and joins can be computed more efficiently as long as the maximum degree of any node in the lattice is not too large. This is the case for distributive lattices, for example, as $\log_2 n$ is the maximum degree of a node in a distributive lattice[4]. In this section, we explore new data structures for meet and join operations that perform well under this assumption.

Let d be the maximum degree of any node in a partial lattice L. As a convenience, we assume in this section that L has a top element. The purpose of this assumption is to avoid a lattice with more than d maximal elements; otherwise we would need to define d as the larger of the maximum degree and the number of maximal elements in the lattice.

This assumption has the effect that the residual block in any block decomposition of L has a top element (unless it is empty). The only practical difference between the residual block and a principal block is that the residual block may be smaller than the block size of the decomposition. The results of this section are easier to relate if we assume henceforth that all blocks are principal blocks and each has a block header. Thus, a block decomposition with block size k creates m blocks $B_1, \ldots, B_m$ with block headers $h_1, \ldots, h_m$, where $|B_i| \ge k$ for $1 \le i \le m-1$. The number of blocks is at most $\frac{n}{k}+1$.

We begin with a simple data structure that computes joins between elements using a new strategy. It is more efficient than our earlier method when $d \le n^{3/4}$. We then generalize

[4]We leave this as an exercise using Birkhoff's Representation Theorem [31].

the idea to create a more sophisticated recursive data structure. It improves on the simple structure for all values of d and works especially well when $d \le \sqrt{n}$. The space usage is $O(n^{3/2})$ for both data structures. Either one can be used to compute meets as well by inverting the lattice order and rebuilding the data structure, although the value of d may be different in the flipped lattice.

Theorem 11. *There is a data structure for lattices that requires $O(n^{3/2})$ space and computes the join of two elements in $O(\sqrt{n} + d)$ time.*

Proof. This data structure uses a block decomposition with block size $k = \sqrt{n}$ and stores $(\mathcal{A})$ and $(\mathcal{B})$ just as in Section 2.5. This is everything we need to perform order-testing in constant time. However, we now use this information to compute *joins* instead of meets.

Let $B_1, \ldots, B_m$ be the blocks of the decomposition with block headers $h_1, \ldots, h_m$. Further assume that the order $B_1, B_2, \ldots, B_m$ reflects the order that the blocks were extracted from L during the decomposition.

Given $x, y \in L$, $x \vee y$ may be found as follows.

(1) Use order-testing to compare x and y to every block header. Let $i^* \in \{1, \ldots, m\}$ be the smallest value for which $x \le h_{i^*}$ and $y \le h_{i^*}$.

(2) It must be that $x \vee y$ lies in B_{i^*}. Let $c_1, c_2, \ldots, c_t \in B_{i^*}$ be the elements covered by h_{i^*} in B_{i^*}[5]. Compare x and y to each of these elements using order-testing queries. If $x, y \le c_j$ for some $j \in \{1, \ldots, t\}$, then proceed to step (3). Otherwise, conclude that $x \vee y = h_{i^*}$.

(3) The join of x and y must lie in the local downset of c_j. Find $x \vee y$ by comparing x and y to every element in $\downarrow c_j \cap B_{i^*}$ and choosing the smallest node z with $x, y \le z$.

This procedure always finds $x \vee y$. The purpose of step (1) is to identify the block containing $x \vee y$. With i^* defined as in the algorithm, observe that $x \vee y$ must have been added to B_{i^*} during the decomposition because $x \vee y \in \downarrow h_{i^*}$ and $x \vee y \notin \downarrow h_i$ for any $i < i^*$. This step takes $O(\sqrt{n})$ time as $m \le \sqrt{n} + 1$.

Once B_{i^*} has been identified, the difficulty lies in finding the join. The algorithm checks all of the children $c_1, \ldots, c_t$ of h_{i^*} to find an element c_j above $x \vee y$. This step requires $O(d)$ time as $t \le d$. If the algorithm succeeds in finding c_j then it compares x and y with all of the elements in the local downset of c_j to determine the join. By the thinness property, this step takes only $O(\sqrt{n})$ time. If no such c_j exists, then h_{i^*} must be the only element in B_{i^*} above both x and y. Thus, this data structure finds $x \vee y$ in $O(\sqrt{n} + d)$ time. □

[5] It is possible that h_{i^*} covers other elements belonging to earlier blocks. These are not included.

Theorem 12. *There is a data structure for lattices that requires $O(n^{3/2})$ space and computes the join of two elements in $O(d\frac{\log n}{\log d})$ time.*

Proof. We extend the ideas of Theorem 11 using a recursive decomposition of a lattice.

The recursive decomposition works in two stages. First, we perform a block decomposition of L using the block size n/d. This produces up to $d+1$ blocks $B_1, \dots, B_m$.

We decompose each B_i further using a *cover decomposition*. If B_i has a block header h_i and $c_1, c_2, \dots, c_t \in B_i$ are the elements covered by h_i, then a cover decomposition of B_i is a partition of B_i into the sets

$$C_{i,j} = (\downarrow c_j \cap B_i) \smallsetminus (\bigcup_{\ell=1}^{j-1} \downarrow c_j) \text{ for } 1 \leq j \leq t.$$

We call these sets *chunks* to avoid overloading "block" and we call c_j the *chunk header* of $C_{i,j}$. Unlike a block decomposition, a cover decomposition does not depend on a block size. It is unique up to the ordering of $c_1, \dots, c_t$.

So far, our decomposition produces blocks $\{B_1, \dots, B_m\}$ and chunks $\{C_{i,j} \mid 1 \leq i \leq m, 1 \leq j \leq \deg(h_i)\}$. We recursively decompose every chunk $C_{i,j}$ in the same two stages, first by a block decomposition with block size $\frac{|C_{i,j}|}{d}$ and then by a cover decomposition of each of the resulting blocks. The recursive decomposition continues in this fashion on any chunk with size at least $2d$.

The recursion induces a tree structure on the set of block headers and chunk headers in the lattice. The children of each block header are the chunk headers chosen during its decomposition and vice versa. The order of the children of a node corresponds to the order that the blocks or chunks are taken during the decomposition. Finally, we create one special node to act as the root of the tree. The children of the root are the block headers of the initial decomposition. We call this the *decomposition tree*.

It is easy to see that every lattice element occurs at most once in the tree and that the maximum degree of any tree node is at most $d+1$. Less obvious is the fact that the depth of the tree is $O(\frac{\log n}{\log d})$.

To see this, let c be a chunk header, let h be one of its children in the tree, and let c' be a child of h. Assume c is the header of a chunk C, h is the header of a block B contained in C, and c' is the header of a chunk C' contained in B; see Figure 2.3. Block B was formed during a block decomposition of C with size $|C|/d$. Since h covers c' in B, c' must have been a thin node during that decomposition. The chunk C' was then created from the local downset of c' in B. Thus $|C'| \leq |\downarrow c' \cap B| \leq |C|/d$.

This implies that the size of chunks decreases by a factor of d between every chunk header and its grandchildren in the decomposition tree. After $2\lceil\frac{\log n}{\log d}\rceil$ generations in the decomposition tree, every chunk must have size less than $2d$. This proves the claim.

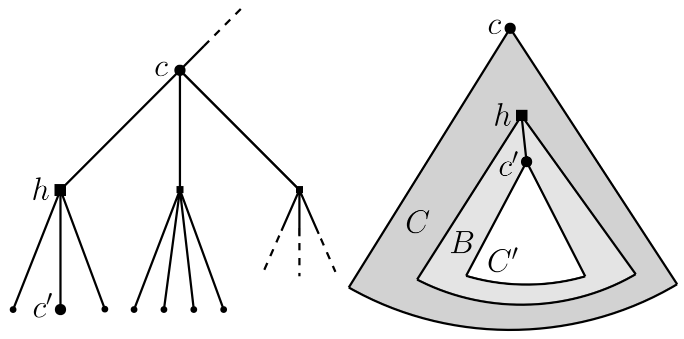

Figure 2.3: Three nodes in the decomposition tree and the corresponding chunks and block of the recursive decomposition. The size of C' can be no larger than $|C|/d$.

The data structure is now simple to describe. We store the decomposition tree and, for each leaf, we store a list of the elements in the chunk of that chunk header. Since the chunks represented by leaves are pairwise disjoint, only $O(n)$ space is needed for this structure. Additionally, we create and store the order-testing structure of Section 2.5, bringing the total space to $O(n^{3/2})$.

The join of two elements can be found using a recursive version of the algorithm from Theorem 11. Suppose we are given $x, y \in L$ and must determine $x \vee y$. Through a variable u that represents the node being considered, we recursively traverse the decomposition tree. Initially set u equal to the root and proceed as follows.

Base Case:

If u is a leaf, then consider the stored list of elements for u. Find $x \vee y$ by comparing x and y to every element in the list and returning the smallest node z with $x, y \leq z$.

Recursive Case:

If u is not a leaf, let $v_1, v_2, \ldots, v_k$ be the children of u in the decomposition tree, listed in order. Use order-testing to compare x and y to each v_i. If there is no v_i such that $x \leq v_i$ and $y \leq v_i$, then conclude that $x \vee y = u$. Otherwise, let $i^* \in \{1, \ldots, k\}$ be the smallest value for which $x \leq v_{i^*}$ and $y \leq v_{i^*}$. Recurse on v_{i^*}.

This procedure spends $O(d)$ time on each node. In the base case, the list stored for u has length $O(d)$ and the join can be found in this list in linear time. The recursive case takes $O(d)$ time as well since the maximum degree of the decomposition tree is at most $d+1$. As the depth of the tree is $O(\frac{\log n}{\log d})$, the total time of this procedure is $O(d\frac{\log n}{\log d})$.

Correctness is a consequence of the fact that $x \vee y$ lies in the first block of each block decomposition whose header is above both x and y. The same fact holds for the chunks in a cover decomposition. Thus, each time i^* is chosen in the recursive case, it must be that $x \vee y$ lies in the block or chunk for v_{i^*}. □

2.9 Correcting Earlier Work

As stated in the introduction, this chapter relies on ideas from the lattice data structure of [221, 222, 223]. These papers contain a mistake that we believe is not easily repaired. The purpose of this section is to summarize their techniques, explain where the error occurs, and argue that it cannot be fixed by a minor modification. We urge the interested reader to consult [223] to confirm this analysis.

We restate their algorithm in the language of this chapter. In the interest of a clear and concise explanation, we do not rebuild all the machinery of their work. In particular, we ignore their *double-tree* structure and we only consider blocks made from downsets (in their papers, blocks may be built from upsets or downsets). In our observation, the double-tree structure is necessary only as a null/non-null value check for order testing and meet/join queries (thus a simple dictionary suffices); further, while we have concerns about using both upsets and downsets for blocks, using downsets alone avoids such issues and still satisfies the requirements in their papers (Lemma 4.1 in [223]). We take these liberties for the purpose of quickly coming to the relevant issue. Readers will need to confirm for themselves that our explanation is fundamentally accurate.

Their method relies on a lattice decomposition to build the data structure, and our block decomposition is similar to the basic version of the decomposition described in their papers. Note that what we call "blocks" are called "ideals" in [222] and "clusters" in [223]. They do not decompose the lattice at a second level like our subblock decompositions. The error is introduced in the extended version of their lattice decomposition, which we now describe.

The intuition behind their data structure is that everything would be easier if every block had size $\Theta(\sqrt{n})$, say between $\sqrt{n}$ and $2\sqrt{n}$. If this were the case, then we could afford to explicitly store the meet/join and reachability property between every pair of elements from the same block, as this would use roughly $\sum_{i=1}^{\sqrt{n}} (\sqrt{n})^2 = O(n^{3/2})$ space. This would allow the meet of two elements from the same block to be found in constant time by a simple table lookup. In terms of our meet-finding algorithm from Section 2.6, this would reduce the time for Meet-In-Block to a constant and the time for Meet to $O(\sqrt{n})$.

Dummy Nodes

A block decomposition by itself cannot guarantee anything about the sizes of the blocks except that each is at least $\sqrt{n}$. They attempt to simulate blocks of size $\sqrt{n}$ by modifying the transitive reduction graph (TRG) of the lattice, creating "dummy nodes" with downsets of size $\Theta(\sqrt{n})$ to act as block headers when none exist naturally.

Dummy nodes are introduced as follows. Suppose a block B is created that has more than $2\sqrt{n}$ elements. Assume that the block header has children $c_1, c_2, \ldots, c_t$ in the TRG. Consider the sequence

$$|{\downarrow} c_1 \cap B|, |({\downarrow} c_1 \cup {\downarrow} c_2) \cap B|, \ldots, |({\downarrow} c_1 \cup \cdots \cup {\downarrow} c_t) \cap B|.$$

As each of the children is a thin element (its local downset has size less than $\sqrt{n}$), the difference between adjacent numbers in this sequence is less than $\sqrt{n}$. Thus, there is some $i \in \{1, \ldots, k\}$ such that

$$\sqrt{n} \le |({\downarrow} c_1 \cup \cdots \cup {\downarrow} c_i) \cap B| \le 2\sqrt{n}.$$

The children $c_1, \ldots, c_i$ may be grouped together and the set $({\downarrow} c_1 \cup \cdots \cup {\downarrow} c_i) \cap B$ may be considered as an *artificial* block having size $\Theta(\sqrt{n})$. By removing this artificial block and iterating on the remaining children, B is partitioned into a collection of artificial blocks with sizes between $\sqrt{n}$ and $2\sqrt{n}$ (except that there may be one smaller block at the end). The only difference between these artificial blocks and ordinary principal blocks is that they lack a block header.

To remedy this, a dummy node is introduced at the top of each artificial block. That is, a new element d is created and inserted into the TRG with $c_1, \ldots, c_i$ as its in-neighbours and the block header of B as its only out-neighbour.

Talamo and Vocca rely on the fact that the graph still represents a partial lattice after adding dummy nodes in this way. They state on page 1794 of [223]:

> "By construction, the dag obtained by adding dummy vertices still satisfies the lattice property."

Unfortunately, this claim is not true in many cases. Consider the stripped-down example in Figure 2.5. The lattice on the left is changed to the graph on the right by introducing a dummy node as described. However, the graph on the right fails the lattice property because the join of x and y is not well-defined: Both c_3 and d are minimal among elements in ${\uparrow} x \cap {\uparrow} y$. Symmetrically, the meet of c_3 and d is not well-defined either. In this case, adding d broke the lattice property.

Although the example is on a very small lattice, it scales easily to any size. Any number of nodes could be added to the original lattice so that $|{\downarrow} c_1 \cup {\downarrow} c_2| \in [\sqrt{n}, 2\sqrt{n}]$. The dummy node added in this case would still violate the lattice property.

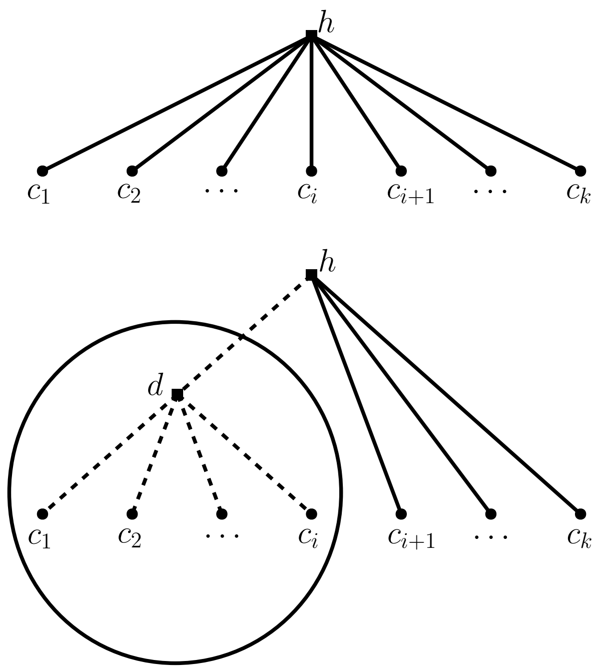

Figure 2.4: Dummy nodes are inserted between the block header and its children to simulate blocks of size $\Theta(\sqrt{n})$.

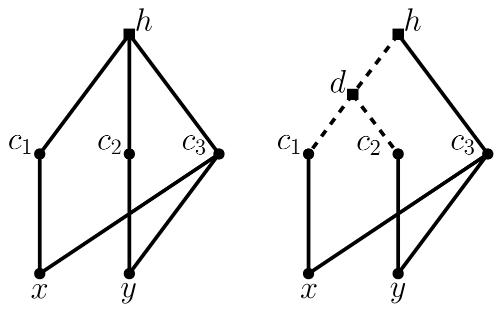

Figure 2.5: Inserting a dummy node breaks the lattice property.

This detail is easy to overlook, especially since $\downarrow d \cap B$ is necessarily a partial lattice. However, the lattice property may fail in the larger structure when dummy nodes are added. This fundamentally impacts the correctness of their approach.

Impact Of The Error

With dummy nodes, it is no longer true that every element has a unique representative in each block. Talamo and Vocca use the following text on page 1789 of [223], "given an external vertex v, the pair $(v, Clus(c))$ univocally identifies a vertex $u \in Clus(c)$ representing either the $\texttt{LUB}(Clus^+(c) \cap Clus^+(v))$ or the $\texttt{GLB}(Clus^-(c) \cap Clus^-(v))$." In the language of our exposition, the claim is that for every block header h, any external element v must have a unique representative $x \wedge h$. Consider again the example in Figure 2.5 with d as the block header of its downset. The external element c_3 does not have a unique representative in the block headed by d, since the meet of d and c_3 is now undefined.

In [223], this breaks Lemma 3.1 when c is a dummy node, which in turn breaks Lemma 3.3 and implies their data structure C on page 1792 of [223] would need to keep multiple entries for an element-cluster pair in order to guarantee correctness of the reachability algorithm described below it. We see no reason why the number of such representatives stored per element should be small, nor that the total number of representatives stored should be small, which undermines both the proposed query and space complexities.

The same issue arises in Talamo and Vocca's meet and join algorithms. The algorithm given relies on the unique representative of an element with a block, and without it, neither the $O(n\sqrt{n})$ space bound nor the $O(\sqrt{n})$ time bound on meet or join operations follow in Proposition 6.4 of [223].

Further, if dummy nodes are avoided altogether, the space bound can be $\Omega(n^2)$, as explained on page 1793 of [223].

It has been suggested to us that the issues may be avoided if the dummy nodes are not considered as actual nodes of the lattice itself, but instead as a construct to group small clusters together for a counting reason. That is, the claim is that an $O(n\sqrt{n})$-space $O(1)$-time order-testing structure can be made without tangibly introducing dummy nodes. As we have shown in this chapter, this is indeed true. However, let us emphasize that the work of Talamo and Vocca does not achieve this. It describes a very different technique that crucially relies on the unique representative property remaining true after grouping clusters using dummy nodes, which does not hold in general regardless of whether dummy nodes are actually inserted into the graph or just used as a conceptual tool. With their techniques, we see no way to achieve their claimed $O(\sqrt{n})$ time meet/join algorithm without their erroneous dummy nodes. We give evidence in the following section as to why this might be infeasible.

Can It Be Fixed?

It is natural to search for a small change to the dummy node method that will fix this issue, allowing us to effectively perform a block decomposition where every principal block has size $\Theta(\sqrt{n})$. It is especially tempting to do so because it could reduce the time for meet and join operations from $O(n^{3/4})$ to $O(\sqrt{n})$, as is claimed in [223]. The dummy node technique also seems like a reasonable approach to handling high-degree lattice nodes, which have often been an obstacle to the approaches we have considered.

There is good reason to expect that this is not possible, relying on some small assumptions. Suppose that there were a correct method of creating artificial principal blocks and that the method still works when we increase the block size from $\sqrt{n}$ to $n^{2/3}$. That is, suppose that we can reliably decompose any lattice into $\Theta(n^{1/3})$ blocks of size $\Theta(n^{2/3})$ (and perhaps some $O(n^{1/3})$ smaller blocks). Note that the structure of the lattice has no impact on the ability to apply this method, thus we assume it applicable to all lattices.

There is the remaining issue of the residual block, however this is not a major difficulty. By adding a top element to the partial lattice (as in a complete lattice), we can treat the residual block in the same fashion as a principal block using the new top element as its block header.

Since the number of lattices on k elements is $2^{\Theta(k^{3/2})}$, it is possible to uniquely identify any such lattice using only $\Theta(k^{3/2})$ bits. Thus, each block of size $\Theta(n^{2/3})$ can be encoded in $\Theta(n)$ bits, and all of the blocks in the decomposition can be encoded in $\Theta(n^{4/3})$ bits. The order between any pair of elements in the same block can be tested, however inefficiently, using the encoding for that block. As well, since there are only $\Theta(n^{1/3})$ block headers, all of the meets between a lattice element and a block header can be stored in $\Theta(n^{4/3})$ space. In other words, we can simulate both $(\mathcal{A})$ and $(\mathcal{B})$ in only $O(n^{4/3})$ space.

This information is sufficient to perform order-testing between any pair of elements, and thus it uniquely determines the lattice. Lattices do not permit such a small representation; this would violate the $\Theta(n^{3/2})$-bit lower bound. This strongly suggests that artificial blocks cannot be simulated without sacrificing the unique representative property, which is essential to the data structure.

2.10 Conclusions

We have presented a data structure to represent lattices in $O(n^{3/2})$ words of space, which is within a $\Theta(\log n)$ factor of optimal. It answers order queries in constant time and meet or join queries in $O(n^{3/4})$ time. This work is intended to replace the earlier solution to this problem which was incorrect; see Section 2.9 for a discussion of the error. Our degree-bounded data structure uses $O(n^{3/2})$ space and answers meet or join queries in $O(d^{\frac{\log n}{\log d}})$ time. For some low-degree lattices, this structure improves dramatically on our subblock-based approach. Ours are the only data structures known to us that uses less than the trivial $O(n^2)$ space.

We wonder what can be done to improve on our results. The time to answer meet and join queries may yet be reduced, perhaps to the $O(\sqrt{n})$ bound claimed by [223]. Another natural question is whether the space of the representation can be reduced to the theoretical minimum of $\Theta(n^{3/2})$ bits.

Chapter 3

Lazy Search Trees

3.1 Introduction

We consider data structures supporting order-based operations such as rank, select, membership, predecessor, successor, minimum, and maximum while providing dynamic operations insert, delete, change-key, split, and merge. The classic solution is the binary search tree (BST), perhaps the most fundamental data structure in computer science. The original unbalanced structure is credited to papers by Booth and Colin [33], Douglas [73], Windley [239], and Hibbard [135] in the early 1960's. Since then, a plethora of *balanced* binary search tree data structures have been proposed [5, 22, 23, 11, 108, 217, 214, 192], notable examples including AVL trees [5], red-black trees [22], and splay trees [217]. A balanced binary search tree is a staple data structure included in nearly all major programming language's standard libraries and nearly every undergraduate computer science curriculum. The data structure is the dynamic equivalent of binary search in an array, allowing searches to be performed on a changing set of keys at nearly the same cost. Extending to multiple dimensions, the binary search tree is the base data structure on which range trees [28], segment trees [26], interval trees [78, 176], kd-trees [25], and priority search trees [177] are all built.

The theory community has long focused on developing binary search trees with efficient *query* times. Although $\Omega(\log n)$ is the worst-case time complexity of a query, on non-uniform access sequences binary search trees can perform better than logarithmic time per query by, for example, storing recently accessed elements closer to the root. The splay tree was devised as a particularly powerful data structure for this purpose [217], achieving desirable access theorems such as static optimality, working set, scanning theorem, static finger, and dynamic finger [217, 62, 61]. The most famous performance statement about the splay tree, however, is the unproven dynamic optimality conjecture, which claims that the performance of the splay tree is within a constant factor of any binary search tree on any sufficiently long access sequence, subsuming all other access theorems. Proving this conjecture is widely considered one of the most important open problems in theoretical computer science, receiving vast attention by data structure researchers [8, 67, 68, 143, 35, 218, 234, 164, 52, 142, 16]. Despite ultimately remaining unsolved for nearly four decades, this topic continues to receive extensive treatment [143, 35, 52, 169, 16].

Although widely considered for the task in literature, the binary search tree is not the most efficient data structure for the standard dictionary abstract data type: in practice, dictionaries are almost always implemented by hash tables, which support $O(1)$ time insert, delete, and look-up in expectation [100, 195]. The advantage of binary search trees, over hash tables, is that they support *order-based* operations. We call dictionaries of this type *sorted dictionaries*, to differentiate them from the simpler data structures supporting only membership queries.

If we limit the order-based operations required of our sorted dictionary to queries for the minimum or maximum element (or both), a number of alternative solutions to the binary search tree have been developed, known as priority queues. The first of which was the binary heap, invented in 1964 for the heapsort algorithm [236]. The binary heap achieves asymptotic complexity equivalent to a binary search tree, though due to the storage of data in an array and fast average-case complexity, it is typically the most efficient priority queue in practice. Later, the invention of the binomial heap showed that the merging of two arbitrary priority queues could be supported efficiently [232, 48], thus proving that the smaller operation set of a priority queue allows more efficient runtimes. The extent to which priority queues can outperform binary search trees was fully realized with the invention of Fibonacci heaps, which showed insertion, merge, and an additional decrease-key operation can all be supported in $O(1)$ amortized time [99]. Since then, a number of priority queues with running times close to or matching Fibonacci heaps have been developed [101, 53, 46, 82, 128, 41, 129]. We refer to such priority queues with $o(\log n)$ insertion and decrease-key costs as *efficient* priority queues, to distinguish them from their predecessors and typically simpler counterparts with $O(\log n)$ insertion and/or decrease-key cost.

The history of efficient priority queues contrasts that of binary search trees. Efficient priority queues have been developed for the case when the number of queries is significantly less than the number of insertions or updates. On the other hand, research on binary search trees has focused on long sequences of element *access*. Indeed, the dynamic optimality conjecture starts with the assumption that n elements are already present in the binary search tree, placing any performance improvements by considering insertion cost entirely outside of the model. However, the theory of efficient priority queues shows that on some operation sequences, the efficiency gains due to considering insertion cost can be as much as a $\Theta(\log n)$ factor, showing an as-of-yet untapped area of potential optimization for data structures supporting the operations of a binary search tree. Further, aside from the theoretically-appealing possibility of the unification of the theories of efficient priority queues and binary search trees, the practicality of improved insertion performance is arguably greater than that of improved access times. For the purpose of maintaining keys in a database, for example, an insert-efficient data structure can provide superior runtimes when the number of insertions dominates the number of queries, a scenario that is certainly the case for some applications [193, 43] and is, perhaps, more likely in general. Yet in spite of these observations, almost no research has been conducted that seriously explores this frontier [36].

We attempt to bridge this gap. We seek a general theory of comparison-based sorted dictionaries that encompasses efficient priority queues and binary search trees, providing the operational flexibility of the latter with the efficiency of the former, when possible. We do not restrict ourselves to any particular BST or heap model; while these models with their stronger lower bounds are theoretically informative, for the algorithm designer these lower bounds in artificially constrained models are merely indications of what *not* to try. If we believe in the long-term goal of improving algorithms and data structures in practice – an objective we think will be shared by the theoretical computer science community at large – we must also seek the comparison with lower bounds in a more permissive model of computation.

We present *lazy search trees*. The lazy search tree is the first data structure to support the general operations of a binary search tree while providing superior insertion time when permitted by query distribution. We show that the theory of efficient priority queues can be generalized to support queries for any rank, via a connection with the multiple selection problem. Instead of sorting elements upon insertion, as does a binary search tree, the lazy search delays sorting to be completed incrementally while queries are answered. A binary search tree and an efficient priority queue are special cases of our data structure that result when queries are frequent and uniformly distributed or only for the minimum or maximum element, respectively. While previous work has considered binary search trees in a "lazy" setting (known as "deferred data structures") [157, 60] and multiple selection in a dynamic setting [19, 20], no existing attempts fully distinguish between insertion and query operations, severely limiting the generality of their approaches. The model we consider gives all existing results as corollaries, unifying several research directions and providing more efficient runtimes in many cases, all with the use of a single data structure.

Before we can precisely state our results, we must formalize the model in which they are attained.

3.1.1 Model and Results

We consider comparison-based data structures on the pointer machine. While we suggest the use of arrays in the implementation of our data structure in practice, constant time array access is not needed for our results. Limiting operations to a pointer machine has been seen as an important property in the study of efficient priority queues, particularly in the invention of strict Fibonacci heaps [46] compared to an earlier data structure with the same worst-case time complexities [41].

We consider data structures supporting the following operations on a dynamic multiset S with (current) size $n = |S|$. We represent elements $e \in S$ as key value pairs (k, v). Some operations require pointers to a specific singular element. We call the corresponding data type *sorted dictionaries*:

- `Construction(`S`)` $:=$ Construct a sorted dictionary on the set S.
- `Insert(`e`)` $:=$ Add key-value element $e = (k, v)$ to S, using key k for comparisons; (this increments n).
- `RankBasedQuery(`r`)` $:=$ Perform a rank-based query pertaining to rank r on S.
- `Delete(ptr)` $:=$ Delete the element pointed to by `ptr` from S; (this decrements n).
- `ChangeKey(ptr,` k'`)` $:=$ Change the key of the element pointed to by `ptr` to k'.
- `Split(`r`)` $:=$ Split S at rank r, returning two sorted dictionaries T_1 and T_2 of r and $n - r$ elements, respectively, such that for all $x \in T_1$, $y \in T_2$, $x \le y$.
- `Merge(`T_1`,` T_2`)` $:=$ Merge sorted dictionaries T_1 and T_2 and return the result, given that for all $x \in T_1$, $y \in T_2$, $x \le y$.

We formalize what queries are possible within the stated operation `RankBasedQuery(`r`)` in Section 3.4. For now, we informally define a rank-based query as any query computable in $O(\log n)$ time on a (possibly augmented) binary search tree and in $O(n)$ time on an unsorted array. Operations rank, select, contains, successor, predecessor, minimum, and maximum fit our definition. To each operation, we associate a rank r: for membership and rank queries, r is the rank of the queried element (in the case of duplicate elements, an implementation can break ties arbitrarily), and for select, successor, and predecessor queries, r is the rank of the element returned; minimum and maximum queries are special cases of select with $r = 1$ and $r = n$, respectively.

The idea of lazy search trees is to maintain a partition of current elements in the data structure into what we will call *gaps*. We maintain a set of m gaps $\{\Delta_i\}$, $1 \le i \le m$, where a gap Δ_i contains a bag of elements. Gaps satisfy a total order, so that for any elements $x \in \Delta_i$ and $y \in \Delta_{i+1}$, $x \le y$. Internally, we will maintain structure within a gap, but the interface of the data structure and the complexity of the operations is based on the distribution of elements into gaps, assuming nothing about the order of elements within a gap. Intuitively, binary search trees fit into our framework by restricting $|\Delta_i| = 1$, so each element is in a gap of its own, and we will see that priority queues correspond to a single gap Δ_1 which contains all elements. Multiple selection corresponds to gaps where each selected rank marks a separation between adjacent gaps.

To insert an element $e = (k, v)$, where k is its key and v its value, we find a gap Δ_i in which it belongs without violating the total order of gaps (if $x \le k$ for all $x \in \Delta_i$ and $k \le y$ for all $y \in \Delta_{i+1}$, we may place e in either Δ_i or Δ_{i+1}; implementations can make either choice). Deletions remove an element from a gap; if the gap is now empty we can remove the gap. When we perform a query, we first narrow the search down to the gap Δ_i in which the query rank r falls (formally, $\sum_{j=1}^{i-1} |\Delta_j| < r \le \sum_{j=1}^{i} |\Delta_j|$). We then answer the query using the elements of Δ_i and *restructure* the gaps in the process. We split gap Δ_i into two gaps Δ'_i and Δ'_{i+1} such that the total order on gaps is satisfied and the rank r element is either the largest in gap Δ'_i or the smallest in gap Δ'_{i+1}; specifically, either $|\Delta'_i| + \sum_{j=1}^{i-1} |\Delta_j| = r$ or $|\Delta'_i| + \sum_{j=1}^{i-1} |\Delta_j| = r - 1$. (Again, implementations can take either choice. We will assume a maximum query to take the latter choice and all other queries the former. More on the choice of r for a given query is discussed in Section 3.4. Our analysis will assume two new gaps replace a former gap as a result of each query. Duplicate queries or queries that fall in a gap of size one follow similarly, in $O(\log n)$ time.) We allow duplicate insertions.

Our performance theorem is the following.

Theorem 13 (Lazy search tree runtimes)**.** *Let n be the total number of elements currently in the data structure and let $\{\Delta_i\}$ be defined as above (thus $\sum_{i=1}^{m} |\Delta_i| = n$). Let q denote the total number of queries. Lazy search trees support the operations of a sorted dictionary on a dynamic set S in the following runtimes:*

- `Construction(`S`)` *in $O(n)$ worst-case time, where $|S| = n$.*
- `Insert(`e`)` *in $O(\min(\log(n/|\Delta_i|) + \log\log|\Delta_i|,\ \log q))$ worst-case time*[1]*, where $e = (k, v)$ is such that $k \in \Delta_i$.*
- `RankBasedQuery(`r`)` *in $O(x \log c + \log n)$ amortized time, where the larger resulting gap from the split is of size cx and the other gap is of size x.*

[1] To simplify formulas, we distinguish between $\log_2(x)$, the binary logarithm for any $x > 0$, and $\log(x)$, which we define as $\max(\log_2(x), 1)$.

- `Delete(ptr)` *in* $O(\log n)$ *worst-case time.*

- `ChangeKey(ptr,` k'`)` *in* $O(\min(\log q, \log\log|\Delta_i|))$ *worst-case time, where the element pointed to by* `ptr`*,* $e = (k, v)$*, has* $k \in \Delta_i$ *and* k' *moves* e *closer to its closest query rank*[2] *in* Δ_i*; otherwise,* `ChangeKey(ptr,` k'`)` *takes* $O(\log n)$ *worst-case time.*

- `Split(`r`)` *in time according to* `RankBasedQuery(`r`)`*.*

- `Merge(`T_1`,` T_2`)` *in* $O(\log n)$ *worst-case time.*

Define[3] $B = \sum_{i=1}^{m} |\Delta_i| \log_2(n/|\Delta_i|)$*. Then over a series of insertions and queries with no duplicate queries, the total complexity is* $O(B + \min(n \log\log n, n \log q))$*.*

We can also bound the number of pointers needed in the data structure.

Theorem 14 (Pointers)**.** *An array-based lazy search tree implementation requires* $O(\min(q, n))$ *pointers.*

By reducing multiple selection to the sorted dictionary problem, we can show the following lower bound.

Theorem 15 (Lower bound)**.** *Suppose we process a sequence of operations resulting in gaps* $\{\Delta_i\}$*. Again define* $B = \sum_{i=1}^{m} |\Delta_i| \log_2(n/|\Delta_i|)$*. Then this sequence of operations requires* $B - O(n)$ *comparisons and* $\Omega(B + n)$ *time in the worst case.*

Theorem 15 indicates that lazy search trees are at most an additive $O(\min(n \log\log n, n \log q))$ term from optimality over a series of insertions and distinct queries. This gives a lower bound on the per-operation complexity of `RankBasedQuery(`r`)` of $\Omega(x \log c)$; the bound can be extended to $\Omega(x \log c + \log n)$ if we amortize the total work required of splitting gaps to each individual query operation. A lower bound of $\Omega(\min(\log(n/|\Delta_i|), \log m))$ can be established on insertion complexity via information theory. We describe all lower bounds in Section 3.5.

We give specific examples of how lazy search trees can be used and how to analyze its complexity according to Theorem 13 in the following subsection.

[2] The closest query rank of e is the closest boundary of Δ_i that was created in response to a query. For gaps Δ_i with $1 \neq i \neq m$, this is the boundary of Δ_i that is closer with respect to the rank of k. Gaps Δ_1 and Δ_m may follow similarly to $i \neq 1, m$ if a minimum or maximum has been extracted. With a single gap Δ_1, increase-key is supported efficiently if maximums have been removed and decrease-key is supported efficiently if minimums have been removed. If both have been removed, the gap functions as in the general case for $i \neq 1, m$. Intuitively, this is configured to support the behavior of decrease-key/increase-key without special casing when the data structure is used as a min-heap/max-heap.

[3] The bound B relates to entropy in the following way: Consider a random variable X denoting the gap to which an element of S selected uniformly at random currently resides. Then $B = n \cdot H(X)$.

3.1.2 Example Scenarios

Below, we give examples of how the performance of Theorem 13 is realized in different operation sequences. While tailor-made data structures for many of these applications are available, lazy search trees provide a *single* data structure that seamlessly adapts to the actual usage pattern while achieving optimal or near-optimal performance for all scenarios in a uniform way.

1. **Few Queries:** The bound $B = \sum_{i=1}^{m} |\Delta_i| \log_2(n/|\Delta_i|)$ satisfies $B = O(n \log q + q \log n)$. In the worst case, queries are uniformly distributed, and the lower bound $B = \Theta(n \log q + q \log n)$. Over a sequence of insertions and queries without duplicate queries, our performance is optimal $O(n \log q + q \log n)$. If $q = n^{\epsilon}$ for constant $\epsilon > 0$, lazy search trees improve upon binary search trees by a factor $1/\epsilon$. If $q = O(\log^c n)$ for some c, lazy search trees serve the operation sequence in $O(cn \log \log n)$ time and if $q = O(1)$, lazy search trees serve the operation sequence in linear time. Although it is not very difficult to modify a previous "deferred data structure" to answer a sequence of n insertions and q queries in $O(n \log q + q \log n)$ time (see Section 3.2.1), to the best of our knowledge, such a result has not appeared in the literature.

2. **Clustered Queries:** Suppose the operation sequence consists of q/k "range queries", each requesting k consecutive keys, with interspersed insertions following a uniform distribution. Here, $B = O(n \log(q/k) + q \log n)$, where q is the total number of keys requested. If the queried ranges are uniformly distributed, $B = \Theta(n \log(q/k) + q \log n)$, with better results possible if the range queries are non-uniform. Our performance on this operation sequence is $O(B + \min(n \log \log n, n \log q))$, tight with the lower bound if $k = \Theta(1)$ or $q/k = \Omega(\log n)$. Similarly to Scenario 1, we pay $O(n \log(q/k))$ in total for the first query of each batch; however, each successive query in a batch costs only $O(\log n)$ time as the smaller resulting gap of the query contains only a single element. We will see in Section 3.5 that we must indeed pay $\Omega(\log n)$ amortized time per query in the worst case; again our advantage is to reduce insertion costs. Note that if an element is inserted within the elements of a previously queried batch, these insertions take $O(\log n)$ time. However, assuming a uniform distribution of element insertion throughout, this occurs on only an $O(q/n)$ fraction of insertions in expectation, at total cost $O(n \cdot q/n \cdot \log n) = O(q \log n)$. Other insertions only need an overall $O(n \log(q/k) + \min(n \log \log n, n \log q))$ time.

3. **Selectable Priority Queue:** If every query is for a minimum element, each query takes $O(\log n)$ time and separates the minimum element into its own gap and all other elements into another single gap. Removal of the minimum destroys the gap

containing the minimum element, leaving the data structure with a single gap Δ_1. All inserted elements fall into this single gap, implying insertions take $O(\log\log n)$ time. Further, the `ChangeKey(ptr,` k'`)` operation supports decrease-key in $O(\log\log n)$ time, since all queries (and thus the closest query) are for rank 1. Queries for other ranks are also supported, though if queried, these ranks are introduced into the analysis, creating more gaps and potentially slowing future insertion and decrease-key operations, though speeding up future selections. The cost of a selection is $O(x\log c + \log n)$ amortized time, where x is the distance from the rank selected to the nearest gap boundary (which was created at the rank of a previous selection) and $c = |\Delta_i|/x - 1$, where the selection falls in gap Δ_i. If no selections have previously occurred, x is the smaller of the rank or n minus the rank selected and $c = n/x - 1$.

Interestingly, finding the kth smallest element in a binary min-heap can be done in $O(k)$ time [97], yet we claim our runtime optimal! The reason is that neither runtime dominates in an amortized sense over the course of n insertions. Our lower bound indicates that $\Omega(B + n)$ time must be taken over the course of multiple selections on n elements in the worst case. In Frederickson's algorithm, the speed is achievable because a binary heap is more structured than an unprocessed set of n elements and only a single selection is performed; the ability to perform further selection on the resulting pieces is not supported. On close examination, lazy search trees can be made to answer the selection query alone without creating additional gaps in $O(x + \log n)$ amortized time or only $O(x)$ time given a pointer to the gap in which the query falls (such modification requires fulfilling Rules (B) and (C) on category A intervals in Section 3.7.2).

4. **Double-Ended Priority Queue:** If every query is for the minimum or maximum element, again each query takes $O(\log n)$ time and will separate either the minimum or maximum element into its own gap and all other elements into another single gap. The new gap is destroyed when the minimum or maximum is extracted. As there is only one gap Δ_1 when insertions occur, insertions take $O(\log\log n)$ time. In this case, our data structure natively supports an $O(\log\log n)$ time decrease-key operation for keys of rank $n/2$ or less and an $O(\log\log n)$ time increase-key operation for keys of rank greater than $n/2$. Further flexibility of the change-key operation is discussed in Section 3.7.4.

5. **Online Dynamic Multiple Selection:** Suppose the data structure is first constructed on n elements. (A close analysis of insert in Section 3.7.1 shows that alternatively, we can construct the data structure on an empty set and achieve $O(1)$ time insertion before a query is performed.) After construction, a set of ranks $\{r_i\}$ are se-

lected, specified online and in any order. Lazy search trees will support this selection in $O(B)$ time, where $B = \sum_{i=1}^{m} |\Delta_i| \log_2(n/|\Delta_i|)$ is the lower bound for the multiple selection problem [147]. We can further support additional insertions, deletions and queries. Data structures for online dynamic multiple selection were previously known [19, 20], but the way we handle dynamism is more efficient, allowing for all the use cases mentioned here. We discuss this in Section 3.2.

6. **Split By Rank:** Lazy search trees can function as a data structure for repeated splitting by rank, supporting construction on an initial set of n elements in $O(n)$ time, insertion into a piece of size n in $O(\log \log n)$ time, and all splitting within a constant factor of the information-theoretic lower bound. Here, the idea is that we would like to support the operations insert and split at rank r, returning two pieces of a data structure of the same form. In a sense, this is a generalization of priority queues, where instead of extracting the minimum, we may extract the k smallest elements, retaining the ability to perform further extractions on either of the two pieces returned. As in scenario 3, the cost of splitting is $O(x \log c + \log n)$, where x is the number of elements in the smaller resulting piece of the split, and we define c so that the number of elements in the larger resulting piece of the split is cx. Again, $O(x \log c + \log n)$ is optimal. Note that we could also use an $O(\log \log n)$ time change-key operation for this application, though this time complexity only applies when elements are moved closer to the nearest split rank. If repeatedly extracting the k smallest elements is desired, this corresponds to an $O(\log \log n)$ time decrease-key operation.

7. **Incremental Quicksort:** A version of our data structure can perform splits internally via selecting random pivots with expected time complexity matching the bounds given in Theorem 13. (We initially describe a version using exact selection, which is conceptually simpler but less practical.) The data structure can then be used to extract the q smallest elements in sorted order, online in q, via an incremental quicksort. Here, $B = \Theta(q \log n)$ and our overall time complexity is $O(n + q \log n)$, which is optimal up to constant factors[4]. Previous algorithms for incremental sorting are known [196, 190, 208, 15]; however, our algorithm is extremely flexible, progressively sorting any part of the array in optimal time $O(B + n)$ while also supporting insertion, deletion, and efficient change-key. The heap operations insert and decrease-key are performed in $O(\log \log n)$ time instead of $O(\log n)$, compared to existing heaps based on incremental sorting [189, 190]; see also [77, 42]. Our data structure also uses

[4]Note that $n + q \log n = \Theta(n + q \log q)$. If the $q \log n$ term dominates, $q = \Omega(n/\log n)$ and so $\log n = \Theta(\log q)$.

only $O(\min(q, n))$ pointers, providing many of the same advantages of sorting-based heaps. A more-complicated priority queue based on similar ideas to ours achieves Fibonacci heap amortized complexity with only a single extra word of space [181].

We discuss the advantages and disadvantages of our model and data structure in the following subsections.

3.1.3 Advantages

The advantages of lazy search trees are as follows:

1. Superior runtimes to binary search trees can be achieved when queries are infrequent or non-uniformly distributed.

2. A larger operation set, with the notable exception of efficient general merging, is made possible when used as a priority queue, supporting operations within an additive $O(n \log \log n)$ term of optimality, in our model.

3. Lazy search trees can be implemented to use only $O(\min(q, n))$ pointers, operating mostly on arrays. This suggests smaller memory footprint, better constant factors, and better cache performance compared to many existing efficient priority queues or binary search trees. Our data structure is not built on the heap-ordered tree blueprint followed by many existing efficient priority queues [99, 101, 53, 46, 82, 128, 41, 129]. Instead, we develop a simple scheme based on unordered lists that may of independent interest. In particular, we are hopeful our data structure or adaptations thereof may provide a theoretically-efficient priority queue that gets around the practical inefficiencies associated with Fibonacci heaps [99] and its derivatives.

4. While not a corollary of the model we consider, lazy search trees can be made to satisfy all performance theorems with regards to access time satisfied by splay trees. In this way, lazy search trees can be a powerful alternative to the splay tree. Locality of access can decrease both access and insertion times. This is discussed in Section 3.11.

3.1.4 Disadvantages

The weaknesses of lazy search trees are as follows:

1. Our gap-based model requires inserted elements be placed in a gap immediately instead of delaying all insertion work until deemed truly necessary by query operations. In particular, a more powerful model would ensure that the number of comparisons

performed on an inserted element depends only on the queries executed *after* that element is inserted. There are operation sequences where this can make a $\Theta(\log n)$ factor difference in overall time complexity, but it is not clear whether this property is important on operation sequences arising in applications.

2. We currently do not know whether the additive $O(\min(n \log q, n \log \log n))$ term in the complexity described in Theorem 13 over a sequence of insertions and queries is necessary. Fibonacci heaps and its variants show better performance is achievable in the priority queue setting. In Section 3.9, we show the (essentially) $O(\log \log |\Delta_i|)$ terms for insertion and change-key can be improved to a small constant factor if the (new) rank of the element is drawn uniformly at random from valid ranks in Δ_i. As a priority queue, this corresponds with operation sequences in which binary heaps [236] provide constant time insertion.

3. The *worst-case* complexity of a single `RankBasedQuery`(r) can be $O(n)$. Further, unlike amortized search trees like the splay tree [217], the average case complexity is not necessarily $O(\log n)$. By delaying sorting, our lower bound indicates that we may need to spend $\Theta(n)$ time to answer a query that splits a gap of size $|\Delta_i| = \Theta(n)$ into pieces of size x and cx for $c = \Theta(1)$. Further, aside from an initial $O(\log n)$ time search, the rest of the time spent during query is on writes, so that over the course of the operation sequence the number of writes is $\Theta(B + n)$. In this sense, our algorithm functions more similarly to a lazy quicksort than a red-black tree [22], which requires only $\Theta(n)$ writes regardless of operation sequence.

3.1.5 Chapter Organization

We organize the remainder of the chapter as follows. In the following section, Section 3.2, we discuss related work. In Section 3.3, we give a high-level overview of the technical challenge. In Section 3.4, we formalize the definition of the queries we support. In Section 3.5, we discuss lower bounds in our gap-based model. In Section 3.6, we show how lazy search trees perform insertions, queries, deletions, and change-key operations. We analyze the costs of these operations in Section 3.7. In Section 3.8, we explain how binary search tree bulk-update operations split and merge can be performed on lazy search trees. We show in Section 3.9 that the complexity of insertion and change-key can be improved with a weak average-case assumption. In Section 3.10, we show that exact selection in our query algorithm can be replaced with randomized pivoting while achieving the same expected time complexity. In Section 3.11, we show how splay trees can be used with lazy search trees and show that lazy search trees can be made to support efficient access theorems. We give concluding remarks, open problems, and briefly discuss a proof-of-concept implementation in Section 3.12.

3.2 Related Work

Lazy search trees unify several distinct research fields. The two largest, as previously discussed, are the design of efficient priority queues and balanced binary search trees. We achieved our result by developing an efficient priority queue and lazy binary search tree simultaneously. There are no directly comparable results to our work, but research in *deferred data structures* and *online dynamic multiple selection* comes closest. We further discuss differences between dynamic optimality and our work.

3.2.1 Deferred Data Structures

To our knowledge, the idea of deferred data structures was first proposed by Karp, Motwani, and Raghavan in 1988 [157]. Similar ideas have existed in slightly different forms for different problems [219, 34, 44, 17, 21, 13, 125, 6]. The term "deferred data structure" has been used more generally for delaying processing of data until queries make it necessary, but we focus on works for one-dimensional data here, as it directly pertains to the problem we consider.

Karp, Motwani and Raghavan [157] study the problem of answering membership queries on a static, unordered set of n elements in the comparison model. One solution is to construct a binary search tree of the data in $O(n \log n)$ time and then answer each query in $O(\log n)$ time. This is not optimal if the number of queries is small. Alternatively, we could answer each query in $O(n)$ time, but this is clearly not optimal if the number of queries is large. Karp et al. determine the lower bound of $\Omega((n+q)\log(\min(n,q))) = \Omega(n \log q + q \log n)$ time to answer q queries on a static set of n elements in the worst case and develop a data structure that achieves this complexity.

This work was extended in 1990 to a dynamic model. Ching, Melhorn, and Smid show that q' membership queries, insertions, and deletions on an initial set of n_0 unordered elements can be answered in $O(q' \log(n_0+q')+(n_0+q')\log q') = O(q' \log n_0 + n_0 \log q' + q' \log q')$ time [60]. When membership, insertion, and deletion are considered as the same type of operation, this bound is optimal.

It is not very difficult (although not explicitly done in [60]) to modify the result of Ching et al. to obtain a data structure supporting n insertions and q'' membership or deletion operations in $O(q'' \log n + n \log q'')$ time, the runtime we achieve for uniform queries. We will see in Section 3.3 that the technical difficulty of our result is to achieve the fine-grained complexity based on the query-rank distribution. For more work in one-dimensional deferred data structures, see [219, 34, 44, 17, 21, 125].

3.2.2 Online Dynamic Multiple Selection

The optimality of Karp et al. [157] and Ching et al. [60] is in a model where the ranks requested of each query are not taken into account. In the multiple selection problem, solutions have been developed that consider this information in the analysis. Suppose we wish to select the elements of ranks $r_1 < r_2 < \cdots < r_q$ amongst a set of n unordered elements. Define $r_0 = 0$, $r_{q+1} = n$, and Δ_i as the set of elements of rank greater than r_{i-1} and at most r_i. Then $|\Delta_i| = r_i - r_{i-1}$ and as in Theorem 13, $B = \sum_{i=1}^{m} |\Delta_i| \log_2(n/|\Delta_i|)$. The information-theoretic lower bound for multiple selection is $B - O(n)$ comparisons [72]. Solutions have been developed that achieve $O(B+n)$ time complexity [72] or $B+o(B)+O(n)$ comparison complexity [147].

The differences between the multiple selection problem and deferred data structuring for one-dimensional data are minor. Typically, deferred data structures are designed for online queries, whereas initial work in multiple selection considered the setting when all query ranks are given at the same time as the unsorted data. Solutions to the multiple selection problem where the ranks $r_1, \ldots, r_q$ are given online and in any order have also been studied, however [18]. Barbay et al. [19, 20] further extend this model to a dynamic setting: They consider online dynamic multiple selection where every insertion is preceded by a search for the inserted element. Deletions are ultimately performed in $O(\log n)$ time. Their data structure uses $B + o(B) + O(n + q' \log n)$ comparisons, where q' is the number of search, insert, and delete operations. The crucial difference between our solution and that of Barbay et al. [19, 20] is how we handle insertions. Their analysis assumes every insertion is preceded by a search and therefore insertion must take $\Omega(\log n)$ time. Thus, for their result to be meaningful (i.e., allow $o(n \log n)$ performance), the algorithm must start with an initial set of $n_0 = n \pm o(n)$ elements. While Barbay et al. focus on online dynamic multiple selection algorithms with near-optimal comparison complexity, the focus of lazy search trees is on generality. We achieve similar complexity as a data structure for online multiple selection while also achieving near-optimal performance as a priority queue. We discuss the technical challenges in achieving this generality in Section 3.3.

3.2.3 Dynamic Optimality

As mentioned, the dynamic optimality conjecture has received vast attention in the past four decades [8, 67, 68, 143, 35, 218, 234, 164, 52]. The original statement conjectures that the performance of the splay tree is within a constant factor of the performance of any binary search tree on any sufficiently long access sequence [217]. To formalize this statement, in particular the notion of "any binary search tree", the BST model of computation has been introduced, forcing the data structure to take the form of a binary tree with access from the root and tree rotations for updates. Dynamic optimality is enticing because it conjectures splay trees [217] and a related "greedy BST" [67] to be within a constant factor of optimality on *any* sufficiently long access sequence. This *per-instance* optimality [90] is more powerful than the sense of optimality used in less restricted models, where it is often unattainable. Any sorting algorithm, for example, must take $\Omega(n \log n)$ time in the *worst case*, but on any particular input permutation, an algorithm designed to first check for that specific permutation can sort it in $O(n)$ time: simply apply the inverse permutation and check if the resulting order is monotonic.

The bounds we give in Section 3.5 are w. r. t. the *worst case* over operation sequences based on distribution of gaps $\{\Delta_i\}$, but hold for *any* comparison-based data structure. Hence, lazy search trees achieve a weaker notion of optimality compared to dynamic optimality, but do so against a vastly larger class of algorithms.

Since splay trees, greedy BSTs, and lazy search trees are all implementations of sorted dictionaries and conjectured dynamically optimal, it is insightful to contrast the access theorems of dynamically-optimal BSTs with the improvements given in Theorem 13. Superficially, the two notions are orthogonal, with dynamic optimality allowing only queries, and our bound becoming interesting mostly when insertions and queries are mixed. On the other hand, the form of performance improvements achievable are indeed quite similar, as the following property shows.

Definition 16 (Static Optimality [162, 8, 217])**.** *Let S denote the set of elements in the data structure and let q_x denote the number of times element x is accessed in a sequence of m accesses. Assume every element is accessed at least once. A data structure is said to achieve static optimality if the cost to perform any such access sequence is*

$$O(m + \sum_{x \in S} q_x \log(m/q_x)).$$

Historically, research into optimal binary search trees started with this notion of static optimality, and both splay trees and greedy BSTs have been shown to be statically optimal [217, 95]. Contrast the bound given in Definition 16 with the bound $O(B + n)$, where again we define $B = \sum_{i=1}^{m} |\Delta_i| \log_2(n/|\Delta_i|)$. If we replace q_x and m in Definition 16 with $|\Delta_i|$ and n, respectively, they are exactly the same: the savings for query costs arising from repeated accesses with nonuniform access probabilities equal the savings for insertion costs when query ranks are nonuniform.

3.3 Technical Overview

This research started with the goal of generalizing a data structure that supports n insertions and $q \le n$ rank-based queries in $O(n \log q)$ time. Via a reduction from multiple selection, $\Omega(n \log q)$ comparisons are necessary in the worst case. However, by applying the fine-grained analysis based on rank distribution previously employed in the multiple selection literature [72], a new theory which generalizes efficient priority queues and binary search trees is made possible.

As will be discussed in Section 3.5, to achieve optimality on sequences of insertion and distinct queries with regards to the fine-grained multiple selection lower bound, insertion into gap Δ_i should take $O(\log(n/|\Delta_i|))$ time. A query which splits a gap Δ_i into two gaps of sizes x and cx ($c \ge 1$), respectively, should take $O(x \log c + \log n)$ time. These complexities are the main goals for the design of the data structure.

The high-level idea will be to maintain elements in a gap Δ_i in an auxiliary data structure (the *interval data structure* of Section 3.6). All such auxiliary data structures are then stored in a biased search tree so that access to the ith gap Δ_i is supported in $O(\log(n/|\Delta_i|))$ time. This matches desired insertion complexity and is within the $O(\log n)$ term of query complexity. The main technical difficulty is to support efficient insertion and repeated splitting of the auxiliary data structure.

Our high-level organization is similar to the selectable sloppy heap of Dumitrescu [74]. The difference is that while the selectable sloppy heap keeps fixed quantile groups in a balanced search tree and utilizes essentially a linked-list as the auxiliary data structure, in our case the sets of elements stored are dependent on previous query ranks, the search tree is biased, and we require a more sophisticated auxiliary data structure.

Indeed, in the priority queue case, the biased search tree has a single element Δ_1, and all operations take place within the auxiliary data structure. Thus, we ideally would like to support $O(1)$ insertion and $O(x \log c)$ split into parts of size x and cx ($c \ge 1$) in the auxiliary data structure. If the number of elements in the auxiliary data structure is $|\Delta_i|$, we can imagine finding the minimum or maximum as a split with $x = 1$ and $c = |\Delta_i| - 1$, taking $O(\log |\Delta_i|)$ time. However, the ability to split at any rank in optimal time complexity is not an operation typically considered for priority queues. Most efficient priority queues store elements in heap-ordered trees, providing efficient access to the minimum element but otherwise imposing intentionally little structure so that insertion, decrease-key, and merging can all be performed efficiently.

Our solution is to group elements within the auxiliary data structure in the following way. We separate elements into groups ("intervals") of unsorted elements, but the elements between each group satisfy a total order. Our groups are of exponentially increasing size as distance to the gap boundary increases. Within a gap Δ_i, we maintain $O(\log |\Delta_i|)$ such groups. Binary search then allows insertion and key change in $O(\log\log|\Delta_i|)$ time. While not $O(1)$, the structure created by separating elements in this way allows us to split the data structure in about $O(x)$ time, where x is the distance from the split point to the closest query rank. Unfortunately, several complications remain.

Consider if we enforce the exponentially-increasing group sizes in the natural way in data structure design. That is, we select constants $c_1 \leq c_2$ such that as we get farther from the gap boundary, the next group is at least a factor $c_1 > 1$ larger than the previous but at most a factor c_2. We can maintain this invariant while supporting insertion and deletion, but splitting is problematic. After splitting, we must continue to use both pieces as a data structure of the same form. However, in the larger piece, the x elements removed require restructuring not only the new closest group to the gap boundary but could require a cascading change on all groups. Since the elements of each group are unstructured, this cascading change could take $\Omega(|\Delta_i|)$ time.

Thus, we must use a more flexible notion of “exponentially increasing" that does not require significant restructuring after a split. This is complicated by guaranteeing fast insertion and fast splits in the future. In particular, after a split, if the larger piece is again split close to where the previous split occurred, we must support this operation quickly, despite avoiding the previous cascading change that would guarantee this performance. Further, to provide fast insertion, we must keep the number of groups at $O(\log|\Delta_i|)$, but after a split, the best way to guarantee fast future splits is to create more groups.

We will show that it is possible to resolve all these issues and support desired operations efficiently by applying amortized analysis with a careful choice of structure invariants. While we do not achieve $O(1)$ insertion and decrease-key cost, our data structure is competitive as an efficient priority queue while having to solve the more complicated issues around efficient repeated arbitrary splitting.

3.4 Rank-Based Queries

We formalize operation `RankBasedQuery(r)` as follows. We first describe what we call an *aggregate function*.

Definition 17 (Aggregate function)**.** *Let S be a multiset of comparable elements and let $f(S)$ be a function*[5] *computed on those elements. Suppose S' is such that S' differs from S by the addition or removal of element x. Let $n = \max(|S|, |S'|)$. Then f is an aggregate*

[5]We do not actually require a strict function $f(S) = y$ for a set S, but rather can let the aggregate function depend on the queries that dynamically change that set. In particular, we can (initially) map $f(S) = \min S$ or $f(S) = \max S$ and change this value to decrease/increase monotonically as S is updated, even if the minimum/maximum is removed.

function maintainable in $g(n)$ time if $f(S')$ can be computed from $f(S)$ and x in $g(n)$ time.

We focus on aggregates with $g(n) = O(1)$, though in principle any $g(n)$ can be supported with appropriate changes to overall runtime complexity. We formalize *rank-based queries* as follows.

Definition 18 (Rank-based query)**.** *Call a query on a multiset of comparable elements S such that $|S| = n$ a rank-based query pertaining to rank r if the following two conditions are satisfied:*

1. *Consider if S is split into two sets X and Y such that for all $x \in X$, $y \in Y$, $x \leq y$. It must be possible, based on an aggregate function f on X and Y, to reduce the query to a query that can be answered considering only the elements of X or only the elements of Y. The query rank r should be such that if the query is reduced to X, $r \leq |X|$, and if the query is reduced to Y, $|X| < r$.*

2. *It must be possible to answer the query on S in $O(n)$ time.*

Critical to our analysis is the rank r associated with each `RankBasedQuery(`r`)` operation. We associate with each operation a rank r which must be contained in each subproblem according to a recursion based on Definition 18. Amongst a set of unsorted elements, r can be chosen arbitrarily, but whichever rank is chosen will affect the restructuring and change the complexity of future operations. Implementation decisions may change r to be $r-1$ or $r+1$; such one-off errors do not have measured effect on complexity, as long as the extract minimum or extract maximum queries result in a single gap Δ_1.

The following well-studied operations fit our definition of rank-based query with ranks r as described; the aggregate function is either the cardinality of the set or a range of keys for the set.

- `Rank(`k`)` := Determine the rank of key k. Rank r is the rank of k in S.
- `Select(`r`)` := Select the element of rank r in S. Rank r is the rank selected.
- `Contains(`k`)` := Determine if an element (k, v) is represented in S (and if so, returns v). Rank r is the rank of k in S.
- `Successor(`k`)` := Determine the successor of key k in S. Rank r is the rank of the successor of k.
- `Predecessor(`k`)` := Determine the predecessor of key k in S. Rank r is the rank of the predecessor of k.

- `Minimum()` := Return a minimum element of S. Rank r is 1.
- `Maximum()` := Return a maximum element of S. Rank r is n.

On edge cases where the successor or predecessor does not exist, we can define r to be n or 1, respectively. Similarly, in the case (k, v) is represented in S on a `Rank(`k`)` or `Contains(`k`)` query, we must pick a tie-breaking rule for rank r returned consistent with the implemented recursion following Definition 18.

3.5 Lower and Upper Bounds

The balanced binary search tree is the most well-known solution to the sorted dictionary problem. It achieves $O(\log n)$ time for a rank-based query and $O(\log n)$ time for all dynamic operations. Via a reduction from sorting, for a sequence of n arbitrary operations, $\Omega(n \log n)$ comparisons and thus $\Omega(n \log n)$ time is necessary in the worst case.

However, this time complexity can be improved by strengthening our model. The performance theorems of the splay tree [217] show that although $\Omega(q \log n)$ time is necessary on a sequence of q arbitrary queries on n elements, many access sequences can be answered in $o(q \log n)$ time. Our model treats sequences of element *insertions* similarly to the splay tree's treatment of sequences of element access. Although $\Omega(n \log n)$ time is necessary on a sequence of n insert or query operations, on many operation sequences, $o(n \log n)$ time complexity is possible, as the theory of efficient priority queues demonstrates.

Our complexities are based on the distribution of elements into the set of gaps $\{\Delta_i\}$. We can derive a lower bound on a sequence of operations resulting in a set of gaps $\{\Delta_i\}$ via reducing multiple selection to the sorted dictionary problem. We prove Theorem 15 below.

Proof of Theorem 15. We reduce multiple selection to the sorted dictionary problem. The input of multiple selection is a set of n elements and ranks $r_1 < r_2 < \cdots < r_q$. We are required to report the elements of the desired ranks. We reduce this to the sorted dictionary problem by inserting all n elements in any order and then querying for the desired ranks $r_1, \ldots, r_q$, again in any order.

Define $r_0 = 0$, $r_{q+1} = n$, and Δ_i as the set of elements of rank greater than r_{i-1} and at most r_i. (This definition coincides with the gaps resulting in our data structure when query rank r falls in the new gap Δ'_i, described in Section 3.1.1.) Then $|\Delta_i| = r_i - r_{i-1}$ and as in Theorem 13, $B = \sum_{i=1}^{m} |\Delta_i| \log_2(n/|\Delta_i|)$. Note that here, $m = q + 1$. The information-theoretic lower bound for multiple selection is $B - O(n)$ comparisons [72]. Since any data structure must spend $\Omega(n)$ time to read the input, this also gives a lower bound of $\Omega(B+n)$

time. This implies the sorted dictionary problem resulting in a set of gaps $\{\Delta_i\}$ must use at least $B - O(n)$ comparisons and take $\Omega(B+n)$ time, in the worst case. $\square$

Remark 19 (Multiple selection inputs)**.** *For the operation sequence from the proof of Theorem 15, Theorem 13 states our performance as* $O(B + \min(n \log q, n \log\log n))$*. A closer examination of our data structure in Section 3.7.1 shows we actually achieve* $O(B+n)$ *complexity on such sequences, since insertions performed before any queries actually take* $O(1)$ *time.*

To achieve the performance stated in Theorem 15 on any operation sequence, we will first consider how the bound $\Omega(B+n)$ changes with insertions and queries. This will dictate the allotted (amortized) time we can spend per operation to achieve an optimal complexity over the entire operation sequence.

We give the following regarding insertion time; recall our convention from Footnote 1 (page 41) that $\log(x) = \max(\log_2(x), 1)$ and $\log_2$ is the binary logarithm.

Lemma 20 (Influence of insert on lower bound)**.** *Suppose we insert an element into gap* Δ_i*. Then the bound* $\Omega(B+n)$ *increases by* $\Omega(\log(n/|\Delta_i|))$*.*

Proof. The insertion simultaneously increases $|\Delta_i|$ and n, but we will consider the effect of these changes separately. We first keep n unchanged and consider how B changes in gap Δ_i. Before insertion, the contribution to B for gap Δ_i is $|\Delta_i|\log_2(n/|\Delta_i|)$; after the insertion it is $(|\Delta_i|+1)\log_2(n/(|\Delta_i|+1))$. Therefore, the change is

$$(|\Delta_i|+1)\log_2(n/(|\Delta_i|+1)) - |\Delta_i|\log_2(n/|\Delta_i|). \tag{3.1}$$

Consider the function $f(x) = x\log_2(n/x)$, where we treat n as a constant. Then (3.1) is at least the minimum value of the derivative $f'(x)$ with $x \in [|\Delta_i|, |\Delta_i|+1]$. The derivative of $f(x)$ is $f'(x) = -\log_2(e) + \log_2(n/x)$. This gives that the change in B is at least $-\log_2(e) + \log_2(n/|\Delta_i|)$.

Now consider the effect of making n one larger. This will only increase B; by the bound $\Omega(B+n)$, this change is (at least) $\Omega(1)$. We may therefore arrive at an increase of $\Omega(\log_2(n/|\Delta_i|)+1) = \Omega(\log(n/|\Delta_i|))$. $\square$

Lemma 20 implies that optimal insertion complexity is $\Omega(\log(n/|\Delta_i|))$. This bound is using the fact the change in the set of gaps $\{\Delta_i\}$ resulting from an insertion corresponds to a multiple selection problem with lower bound greater by $\Omega(\log(n/|\Delta_i|))$. Since the multiple selection problem itself has insertions preceding queries, this lower bound is in some sense artificial. However, we can alternatively consider the problem of determining in which gap an inserted element falls. Here, information theory dictates complexities of

$\Omega(\log m)$ if each gap is weighted equally or $\Omega(\log(n/|\Delta_i|))$ if gap Δ_i is weighted with weight $|\Delta_i|$ [24]. The latter corresponds with the change in B noted above.

We now give the following regarding query time.

Lemma 21 (Influence of query on lower bound)**.** *Suppose a query splits a gap Δ_i into two gaps of size x and cx, respectively, with $c \geq 1$. Then the bound $\Omega(B+n)$ increases by $\Omega(x \log c)$.*

Proof. The change in B is

$$x \log_2\left(\frac{n}{x}\right) + cx \log_2\left(\frac{n}{cx}\right) - (c+1)x \log_2\left(\frac{n}{(c+1)x}\right). \tag{3.2}$$

By manipulating logarithms and canceling terms, we can rearrange (3.2) to $x((c+1)\log_2(c+1) - c\log_2 c)$, which is greater than $x \log_2(c+1)$. Thus the increase in $\Omega(B+n)$ is $\Omega(x \log c)$. □

Lemma 21 gives a lower bound of $\Omega(x \log c)$ per rank-based query operation. Here, the bound is not artificial in any sense: insertions precede queries in the reduction of multiple selection to the sorted dictionary problem. We must spend time $\Omega(x \log c)$ to answer the query as more queries may follow and the total complexity must be $\Omega(B+n)$ in the worst case.

We can improve the query lower bound by considering the effect on B over a sequence of gap-splitting operations. Consider the overall bound $B = \sum_{i=1}^{m} |\Delta_i| \log_2(n/|\Delta_i|)$. It can be seen that $B = \Omega(m \log n)$. Therefore, we can afford amortized $O(\log n)$ time whenever a new gap is created, even if it is a split say with $x = 1$, $c = 1$.

Consider the lower bound given by the set of gaps $\{\Delta_i\}$ in Theorem 15 combined with the above insight that queries must take $\Omega(\log n)$ time. If query distribution is not considered, the worst case is that $|\Delta_i| = \Theta(n/q)$ for all i. Then $B + q \log n = \Omega(n \log q + q \log n)$. This coincides with the lower bound given in [157].

It is worth noting that both Lemma 20 and Lemma 21 can also be proven by information-theoretic arguments, without appealing to the algebraic bound B given in multiple selection. The number of comparisons to identify the x largest elements in a set of $(c+1)x$ elements is $\log\binom{(c+1)x}{x}$, which is $\Omega(x \log c)$. A similar argument can be made that increasing n by 1 and category Δ_i by 1 implies the number of comparisons required of the underlying selection problem increases by $\Omega(\log(n/\Delta_i))$.

3.6 Data Structure

We are now ready to discuss the details of lazy search trees. The high-level idea was discussed in Section 3.3. The data structure as developed is relatively simple, though it requires a somewhat tricky amortized time analysis given in the following section.

We split the data structure into two levels. At the top level, we build a data structure on the set of gaps $\{\Delta_i\}$. In the second level, actual elements are organized into a set of *intervals* within a gap. Given a gap Δ_i, intervals within Δ_i are labeled $\mathcal{I}_{i,1}, \mathcal{I}_{i,2}, \ldots, \mathcal{I}_{i,\ell_i}$, with ℓ_i the number of intervals in gap Δ_i. The organization of elements of a gap into intervals is similar to the organization of elements into a gap. Intervals partition a gap by rank, so that for elements $x \in \mathcal{I}_{i,j}$, $y \in \mathcal{I}_{i,j+1}$, $x \le y$. Elements within an interval are unordered. By convention, we will consider both gaps and intervals to be ordered from left to right in increasing rank. A graphical sketch of the high-level decomposition is given in Figure 3.1.

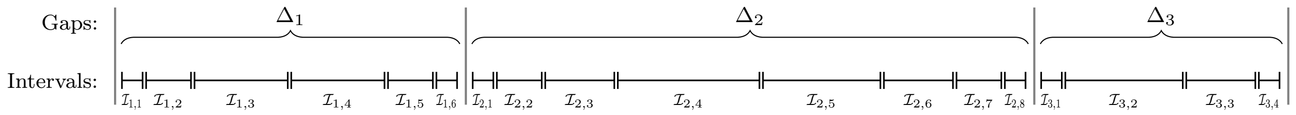
Gaps:
Δ_1
Δ_2
Δ_3
Intervals:
$\mathcal{I}_{1,1}$ $\mathcal{I}_{1,2}$ $\mathcal{I}_{1,3}$ $\mathcal{I}_{1,4}$ $\mathcal{I}_{1,5}$ $\mathcal{I}_{1,6}$
$\mathcal{I}_{2,1}$ $\mathcal{I}_{2,2}$ $\mathcal{I}_{2,3}$ $\mathcal{I}_{2,4}$ $\mathcal{I}_{2,5}$ $\mathcal{I}_{2,6}$ $\mathcal{I}_{2,7}$ $\mathcal{I}_{2,8}$
$\mathcal{I}_{3,1}$ $\mathcal{I}_{3,2}$ $\mathcal{I}_{3,3}$ $\mathcal{I}_{3,4}$

Figure 3.1: The two-level decomposition into gaps $\{\Delta_i\}$ and intervals $\{\mathcal{I}_{i,j}\}$.

3.6.1 The Gap Data Structure

We will use the following data structure for the top level.

Lemma 22 (Gap Data Structure)**.** *There is a data structure for the set of gaps $\{\Delta_i\}$ that supports the following operations in the given worst-case time complexities. Note that $\sum_{i=1}^{m} |\Delta_i| = n$.*

1. *Given an element $e = (k, v)$, determine the index i such that $k \in \Delta_i$, in $O(\log(n/|\Delta_i|))$ time.*

2. *Given a Δ_i, increase or decrease the size of Δ_i by 1, adjusting n accordingly, in $O(\log(n/|\Delta_i|))$ time.*

3. *Remove Δ_i from the set, in $O(\log n)$ time.*

4. *Add a new Δ_i to the set, in $O(\log n)$ time.*

It is also possible to store aggregate functions within the data structure (on subtrees), as required by some queries that fit Definition 18.

Proof. We can use, for example, a globally-biased $2, b$ tree [24]. We assign gap Δ_i the weight $w_i = |\Delta_i|$; the sum of weights, W, is thus equal to n. Access to gap Δ_i, operation 1, is handled in $O(\log(n/|\Delta_i|))$ worst-case time [24, Thm. 1]. By [24, Thm. 11], operation 2 is handled via weight change in $O(\log(n/|\Delta_i|))$ worst-case time. Again by [24, Thm. 11], operations 3 and 4 are handled in $O(\log n)$ worst-case time or better. □

Remark 23 (Alternative implementations)**.** *A variety of biased search trees can be used as the data structure of Lemma 22. In Section 3.11, we suggest splay trees for that purpose, which greatly simplifies implementation at the cost of making the runtimes amortized. What is more, we show that efficient access properties of the data structure of Lemma 22 can be inherited by the lazy search tree, hence the (orthogonal) efficiency gains for insertions in lazy search trees and for structured access sequences in splay trees can be had simultaneously. We also show that a suitable implementation can treat the gap data structure as a merely conceptual abstraction that is not itself implemented; instead it can operate directly on intervals. That further reduces the overhead of an implementation.*

The top level data structure allows us to access a gap in the desired time complexity for insertion. However, we must also support efficient queries. In particular, we need to be able to split a gap Δ_i into two gaps of size x and cx ($c \geq 1$) in amortized time $O(x \log c)$. We must build additional structure amongst the elements in a gap to support such an operation efficiently. At the cost of this organization, in the worst case we pay an additional $O(\log \log |\Delta_i|)$ time on insertion and key-changing operations.

3.6.2 The Interval Data Structure

We now discuss the data structure for the intervals. Given a gap Δ_i, intervals $\mathcal{I}_{i,1}, \mathcal{I}_{i,2}, \ldots, \mathcal{I}_{i,\ell_i}$ are contained within it and maintained in a data structure as follows. We maintain with each interval the two splitting keys (k_l, k_r) that separate this interval from its predecessor and successor (using $-\infty$ and $+\infty$ for the outermost ones), respectively; the interval only contains elements $e = (k, v)$ with $k_l \leq k \leq k_r$. Note that we also must maintain similar router keys at the gap level in the data structure of Lemma 22 to determine in which gap an element falls. We store intervals in sorted order in an array (see Remark 25), sorted with respect to (k_l, k_r). We can then find an interval containing a given key k, i.e., with $k_l \leq k \leq k_r$, using binary search in $O(\log \ell_i)$ time.

Remark 24 (Handling duplicate keys)**.** *Recall that we allow repeated insertions, i.e., elements with the same key k. As detailed in Section 3.6.5, intervals separated by a* *splitting key k can then both contain elements with key k. To guide the binary search in these cases, we maintain for each interval the number of elements with keys equal to the splitting keys k_l and k_r.*

As we will see below, the number of intervals in one gap is always $O(\log n)$, and only changes during a query, so we can afford to update this array on query in linear time.

Remark 25 (Avoiding arrays)**.** *Note that, to stay within the pointer-machine model, we can choose to arrange the intervals within any gap in a balanced binary search tree, thus providing the binary search without array accesses. This also allows to add new intervals efficiently. In practice, binary search on an array is likely to be preferred.*

We conceptually split the intervals into two groups: intervals on the *left* side and intervals on the *right* side. An interval is defined to be in one of the two groups by the following convention.

(A) Left and right intervals: An interval $\mathcal{I}_{i,j}$ in gap Δ_i is on the *left side* if the closest query rank (edge of gap Δ_i if queries have occurred on both sides of Δ_i) is to the left. Symmetrically, an interval $\mathcal{I}_{i,j}$ is on the *right side* if the closest query rank is on the right. An interval with an equal number of elements in Δ_i on its left and right sides can be defined to be on the left or right side arbitrarily.

Recall the definition of closest query rank stated in Footnote 2. The closest query rank is the closest boundary of gap Δ_i that was created in response to a query.

We balance the sizes of the intervals within a gap according to the following rule:

(B) Merging intervals: Let $\mathcal{I}_{i,j}$ be an interval on the left side, not rightmost of left side intervals. We merge $\mathcal{I}_{i,j}$ into adjacent interval to the right, $\mathcal{I}_{i,j+1}$, if the number of elements left of $\mathcal{I}_{i,j}$ in Δ_i equals or exceeds $|\mathcal{I}_{i,j}| + |\mathcal{I}_{i,j+1}|$. We do the same, reflected, for intervals on the right side.

The above rule was carefully chosen to satisfy several components of our analysis. As mentioned, we must be able to answer a query for a rank r near the edges of Δ_i efficiently. This implies we need small intervals near the edges of gap Δ_i, since the elements of each interval are unordered. However, we must also ensure the number of intervals within a gap does not become too large, since we must determine into which interval an inserted element falls at a time cost outside of the increase in B as dictated in Lemma 20. We end up using the structure dictated by Rule (B) directly in our analysis of query complexity, particularly in Section 3.7.2.

Note that Rule (B) causes the loss of information. Before a merge, intervals $\mathcal{I}_{i,j}$ and $\mathcal{I}_{i,j+1}$ are such that for any $x \in \mathcal{I}_{i,j}$ and $y \in \mathcal{I}_{i,j+1}$, $x \le y$. After the merge, this information is lost. Surprisingly, this does not seem to impact our analysis. Once we pay the initial $O(\log \ell_i)$ cost to insert an element via binary search, the merging of intervals happens seldom enough that no additional cost need be incurred.

Rule (B) ensures the following.

Lemma 26 (Few intervals)**.** *Within a gap Δ_i, there are at most $4\log(|\Delta_i|)$ intervals.*

Proof. First consider intervals on the left side. Let intervals $\mathcal{I}_{i,j+1}$ and $\mathcal{I}_{i,j+2}$ be on the left side. It must be that the number of elements in intervals $\mathcal{I}_{i,j+1}$ and $\mathcal{I}_{i,j+2}$ together is equal to or greater than the number of elements in the first j intervals, by Rule (B). Indeed, the worst-case sequence of interval sizes is $1, 1, 1, 2, 2, 4, 4, 8, 8, 16, 16, \ldots$, obtained recursively as $a_1 = a_2 = 1$ and $a_j = a_1 + \cdots + a_{j-2} + 1 - a_{j-1}$. It follows that with every two intervals, the total number of elements at least doubles; indeed we can show that the first k intervals contain at least $(\sqrt{2})^{k+2}$ elements, therefore n elements are spread over at most $2\log_2 n - 2$ intervals. To count intervals on the left resp. right side in Δ_i, we observe that the maximal number of intervals occurs if half of the elements are on either side, so there can be at most $2 \cdot (2\log_2(|\Delta_i|/2) - 2) \le 4\log(|\Delta_i|)$ intervals in gap Δ_i. □

For ease of implementation, we will invoke Rule (B) only when a *query* occurs in gap Δ_i. In the following subsection, we will see that insertion does not increase the number of intervals in a gap, therefore Lemma 26 will still hold at all times even though Rule (B) might temporarily be violated after insertions. We can invoke Rule (B) in $O(\log|\Delta_i|)$ time during a query, since $|\Delta_i| \le n$ and we can afford $O(\log n)$ time per query.

3.6.3 Representation of Intervals

It remains to describe how a single interval is represented internally. Our analysis will require that merging two intervals can be done in $O(1)$ time and further that deletion from an interval can be performed in $O(1)$ time ($O(\log n)$ time actually suffices for $O(\log n)$ time delete overall, but on many operation sequences the faster interval deletion will yield better runtimes). Therefore, the container in which elements reside in intervals should support such behavior. An ordinary linked list certainly suffices; however, we can limit the number of pointers used in our data structure by representing intervals as a linked list of arrays. Whenever an interval is constructed, it can be constructed as a single (expandable) array. As intervals merge, we perform the operation in $O(1)$ time by merging the two linked lists of arrays. Deletions can be performed lazily, shrinking the array when a constant fraction of the entries have been deleted.

We analyze the number of pointers required of this method and the resulting improved bounds on insertion and key change in Section 3.7.5. If we choose not to take advantage of this directly, we can alternatively replace standard linked lists with linked list/array hybrids such as unrolled linked lists [215], which will likely outperform standard linked lists in practice.

3.6.4 Insertion

Insertion of an element $e = (k, v)$ can be succinctly described as follows. We first determine the gap Δ_i such that $k \in \Delta_i$, according to the data structure of Lemma 22. We then binary search the $O(\log|\Delta_i|)$ intervals (by maintaining "router" keys separating the intervals) within Δ_i to find the interval $\mathcal{I}_{i,j}$ such that $k \in \mathcal{I}_{i,j}$. We increase the size of Δ_i by one in the gap data structure.

Remark 27 (A single data structure)**.** *The attentive reader may wonder why we must first perform a binary search for gap Δ_i and then perform another binary search for interval $\mathcal{I}_{i,j}$ within Δ_i. It seems likely these two binary searches can be compressed into one, and indeed, this intuition is correct. If preferred, we can use the data structure of Lemma 22 directly on intervals within gaps, so that weight $|\Delta_i|$ is evenly distributed over intervals $\mathcal{I}_{i,1}, \mathcal{I}_{i,2}, \ldots, \mathcal{I}_{i,\ell_i}$. (Alternatively, assigning weight $|\Delta_i|/\ell_i + |\mathcal{I}_{i,j}|$ to interval $\mathcal{I}_{i,j}$ can provide better runtimes in average case settings.) Unfortunately, doing so means only an $O(\log n)$ time change-key operation can be supported (unless the data structure is augmented further), and (small) weight changes must be performed on the full set of intervals within gap Δ_i on insertion and deletion. While such a data structure is possible, we find the current presentation more elegant and simpler to implement.*

Remark 28 (Lazy insert)**.** *One seemingly-obvious way to improve insertion complexity, improving perhaps either of the first two disadvantages listed in Section 3.1.4, is to* insert lazily. *That is, instead of performing a full insert of $e = (k, v)$ through the gap data structure and then again through the interval data structure, we keep a buffer at each node of the respective BSTs with all the elements that require processing at a later time. While this can improve overall time complexity on some simple operation sequences, it seems difficult to make this strategy efficient overall, when insertions, deletions and queries can be mixed arbitrarily.*

So while improving either of the two disadvantages listed in Section 3.1.4 (and indeed, an improvement in one may imply an improvement in the other) would likely utilize aspects of lazy insertion, we do not currently see a way to achieve this by maintaining buffers on nodes of the BSTs we use.

3.6.5 Query

To answer a query with associated rank r, we proceed as follows. We again determine the gap Δ_i such that $r \in \Delta_i$ according to the process described in Definition 18 on the data structure of Lemma 22. While we could now choose to rebalance the intervals of Δ_i via Rule (B), our analysis will not require application of Rule (B) until the *end* of the query procedure. We recurse into the interval $\mathcal{I}_{i,j}$ such that $r \in \mathcal{I}_{i,j}$, again using the process described in Definition 18 on the intervals of Δ_i (this may use aggregate information stored in the data structure for intervals).

We proceed to process $\mathcal{I}_{i,j}$ by answering the query on $\mathcal{I}_{i,j}$ and replacing interval $\mathcal{I}_{i,j}$ with smaller intervals. First, we partition $\mathcal{I}_{i,j}$ into sets L and R, such that all elements in L are less than or equal to all elements in R and there are r elements in the entire data structure which are either in L or in an interval or gap left of L. This can typically be done in $O(|\mathcal{I}_{i,j}|)$ time using the result of the query itself; otherwise, linear-time selection suffices [32].

We further partition L into two sets of equal size L_l and L_r, again using linear-time selection, such that all elements in L_l are smaller than or equal to elements in L_r; if $|L|$ is odd, we give the extra element to L_l (unsurprisingly, this is not important). We then apply the same procedure *one more time* to L_r, again splitting into equal-sized intervals. Recursing further is not necessary. We do the same, reflected, for set R; after a total of 5 partitioning steps the interval splitting terminates. An example is shown in Figure 3.2.

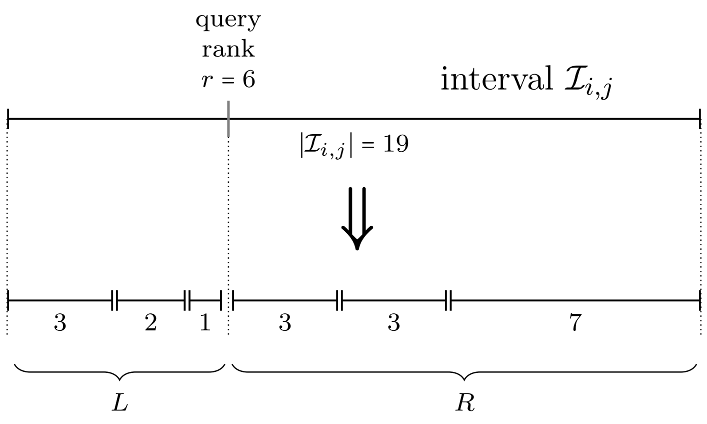

Figure 3.2: An interval $\mathcal{I}_{i,j}$ is split and replaced with a set of intervals.

Remark 29 (Variants of interval replacement)**.** *There is some flexibility in designing this interval-replacement procedure; the critical property needed for our result is the following;*

(details of which will become clear in Section 3.7.2): It yields (1) at most $O(\log|\Delta_i|)$ intervals in gap Δ_i (typically by application of Rule (B)), (2) it satisfies an invariant involving a credit system – Invariant (C) on page 65 – and (3) splitting takes time $O(|\mathcal{I}_{i,j}|)$. In Section 3.10, we show that exact median selection (when splitting L, L_r, R, and R_l) can be replaced with pivoting on a randomly chosen element. On a set of n elements, this requires only n comparisons instead of the at least $1.5n$ required by median-finding in expectation [63], and it is substantially faster in practice.

After splitting the interval $\mathcal{I}_{i,j}$ as described above, we answer the query itself and update the gap and interval data structures as follows. We create two new gaps Δ'_i and Δ'_{i+1} out of the intervals of gap Δ_i including those created from sets L and R. Intervals that fall left of the query rank r are placed in gap Δ'_i, and intervals that fall right of the query rank r are placed in gap Δ'_{i+1}. We update the data structure of Lemma 22 with the addition of gaps Δ'_i and Δ'_{i+1} and removal of gap Δ_i. Finally, we apply Rule (B) to gaps Δ'_i and Δ'_{i+1}.

3.6.6 Deletion

To delete an element $e = (k, v)$ pointed to by a given pointer `ptr`, we first remove e from the interval $\mathcal{I}_{i,j}$ such that $k \in \mathcal{I}_{i,j}$. If e was the only element in $\mathcal{I}_{i,j}$, we remove interval $\mathcal{I}_{i,j}$ from gap Δ_i (we can do so lazily, when Rule (B) is next run on gap Δ_i). Then we decrement Δ_i in the gap data structure of Lemma 22; if that leaves an empty gap, we remove Δ_i from the gap data structure.

3.6.7 Change-Key

The change-key operation can be performed as follows. Suppose we wish to change the key of element $e = (k, v)$, given by pointer `ptr`, to k', and that e currently resides in interval $\mathcal{I}_{i,j}$ in gap Δ_i. We first check if k' falls in Δ_i or if e should be moved to a different gap. If the latter, we can do so as in deletion of e and re-insertion of (k', v). If the former, we first remove e from $\mathcal{I}_{i,j}$. If necessary, we (lazily) delete $\mathcal{I}_{i,j}$ from Δ_i if $\mathcal{I}_{i,j}$ now has no elements. We then binary search the $O(\log|\Delta_i|)$ intervals of Δ_i and place e into the new interval in which it belongs.

Note that although this operation can be performed to change the key of e to anything, Theorem 13 only guarantees runtimes faster than $O(\log n)$ when e moves closer to its nearest query rank within gap Δ_i. Efficient runtimes are indeed possible in a variety of circumstances; this is explored in more detail in Section 3.7.4.

3.7 Analysis

We use an amortized analysis [224]. We will use a potential function with a built-in credit system. Recall that our desired insertion complexity is about $O(\log(n/|\Delta_i|))$ time. On a query that splits a gap into two gaps of size x and cx, we attempt to do so in (amortized) $O(\log n + x \log c)$ time. We require several definitions before we may proceed.

We distinguish between 0-sided, 1-sided, and 2-sided gaps. A 2-sided gap is a gap Δ_i such that queries have been performed on either side of Δ_i; thus, intervals in Δ_i are split into intervals on the left side and intervals on the right side. This is the typical case. A 1-sided gap Δ_i is such that queries have only been performed on one side of the gap; thus, intervals are all on the side towards the query rank in Δ_i. There can be at most two 1-sided gaps at any point in time. In the priority queue case, there is a single 1-sided gap. The final category is a 0-sided gap; when the data structure has performed no queries, all elements are represented in a single interval in a single 0-sided gap.

We now give the following functional definitions.

$$\begin{aligned}
c(\mathcal{I}_{i,j}) &:= \#\text{ of \textit{credits} associated with interval } \mathcal{I}_{i,j}. \\
o(\mathcal{I}_{i,j}) &:= \#\text{ of elements \textit{outside} } \mathcal{I}_{i,j} \text{ in } \Delta_i, \text{ i. e.,} \\
&\ \#\text{ of elements in } \Delta_i \text{ that are left (right) of } \mathcal{I}_{i,j} \text{ if } \mathcal{I}_{i,j} \text{ is on the left (right) side.} \\
M &:= \text{total } \#\text{ of elements in 0-sided or 1-sided gaps.}
\end{aligned}$$

As previously mentioned, intervals are defined to be on either the left or right side according to Rule (A) (page 59). For an interval $\mathcal{I}_{i,j}$ in a 2-sided gap Δ_i, $o(\mathcal{I}_{i,j})$ hence is the minimum number of elements either to the left (less than) or to the right (greater than) $\mathcal{I}_{i,j}$ in gap Δ_i.

The rules for assigning credits are as follows: A newly created interval has no credits associated with it. During a merge, the credits associated with both intervals involved in the merge may be discarded; they are not needed. When an interval $\mathcal{I}_{i,j}$ is split upon a query, it is destroyed and new intervals (with no credits) are created from it; by destroying $\mathcal{I}_{i,j}$, the $c(\mathcal{I}_{i,j})$ credits associated with it are released.

We use the following potential function:

$$\Phi = 10M + 4 \sum_{\substack{1 \le i \le m, \\ 1 \le j \le \ell_i}} c(\mathcal{I}_{i,j}).$$

Credits accumulated when an operation is cheaper than its amortized bound increase Φ; in this way, we use credits to pay for work that will need to be performed in the future. We do so by maintaining the following invariant:

(C) Credit invariant: Let $\mathcal{I}_{i,j}$ be an interval. Then $|\mathcal{I}_{i,j}| \leq c(\mathcal{I}_{i,j}) + o(\mathcal{I}_{i,j})$.

Remark 30 (Intuition behind Invariant (C)). *The intuition behind Invariant (C) is that the cost of splitting $\mathcal{I}_{i,j}$ is entirely paid for by the credits associated with $\mathcal{I}_{i,j}$ and by outside elements, i.e., either released potential or by the distance to previous queries causing a corresponding increase in B. The intervals constructed from the elements of $\mathcal{I}_{i,j}$ are constructed in such a way that they satisfy Invariant (C) at cost a constant fraction of the cost of splitting $\mathcal{I}_{i,j}$.*

Remark 31 (Alternative potential function). *It is possible to remove the credits in our potential function and Invariant (C) and instead use the potential function*

$$\Phi = 10M + 4\sum_{\substack{1\leq i\leq m,\\ 1\leq j\leq \ell_i}} \max(|\mathcal{I}_{i,j}| - o(\mathcal{I}_{i,j}), 0).$$

We opt for the current method as we believe it is easier to work with.

Observe that before any elements are inserted, $\Phi = 0$, and we have a single 0-sided gap with one interval containing no elements. Thus Invariant (C) is vacuously true. We proceed with an amortized analysis of the operations. For our amortization arguments, we assume the potential function to be adjusted to the largest constant in the $O(\cdot)$ notation necessary for the complexity of our operations. In the interest of legibility, we will drop this constant and compare outside of $O(\cdot)$ notation, as is standard in amortized complexity analysis.

3.7.1 Insertion

Insertion of element $e = (k, v)$ can be analyzed as follows. As stated in Lemma 22, we pay $O(\log(n/|\Delta_i|))$ time to locate the gap Δ_i that e belongs into. We adjust the size of Δ_i and n by one in the data structure of Lemma 22 in $O(\log(n/|\Delta_i|))$ time. By Lemma 26, there are $O(\log|\Delta_i|)$ intervals in gap Δ_i, and so we spend $O(\log\log|\Delta_i|)$ time to do a binary search to find the interval $\mathcal{I}_{i,j}$ that e belongs into. We increase the size of $\mathcal{I}_{i,j}$ by one and add one credit to $\mathcal{I}_{i,j}$ to ensure Invariant (C). Thus the total amortized cost of insertion[6] (up to constant factors) is $\log(n/|\Delta_i|) + \log\log|\Delta_i| + 4 + 10 = O(\log(n/|\Delta_i|) + \log\log|\Delta_i|)$. Note that if the data structure for Lemma 22 supports operations in worst-case time, insertion complexity is also worst-case. We show in Section 3.7.5 that the bound $O(\log q)$ also holds.

We use the following lemma to show that Invariant (C) holds on insertion.

[6]Note that although $\log\log|\Delta_i|$ can be $o(\log\log n)$, there is no difference between $O(\sum_{i=1}^{q+1}|\Delta_i|(\log(n/|\Delta_i|) + \log\log n))$ and $O(\sum_{i=1}^{q+1}|\Delta_i|(\log(n/|\Delta_i|) + \log\log|\Delta_i|)$. When the $\log\log n$ term in a gap dominates, $|\Delta_i| = \Omega(n/\log n)$, so $\log\log n = \Theta(\log\log|\Delta_i|)$.

Lemma 32 (Insert maintains Invariant (C))**.** *Updating side designations according to Rule (A) after insertions preserves Invariant (C).*

Proof. The insertion of additional elements may cause an interval $\mathcal{I}_{i,j'}$ in the middle of Δ_i to change sides. This occurs exactly when the number of elements on one side exceeds the number of elements on the other side. However, before this insertion occurred, Invariant (C) held with an equal number of elements on both sides of $\mathcal{I}_{i,j'}$. Since we do not change the number of credits associated with $\mathcal{I}_{i,j'}$, in effect, $o(\mathcal{I}_{i,j'})$ just changes which side it refers to, monotonically increasing through all insertions. It follows Invariant (C) holds according to redesignations via Rule (A) after insertions. □

Invariant (C) then holds on insertion due to Lemma 32 and since $o(\mathcal{I}_{i,j'})$ only possibly increases for any interval $\mathcal{I}_{i,j'}$, $j' \neq j$, with $|\mathcal{I}_{i,j'}|$ remaining the same; recall that an extra credit was added to interval $\mathcal{I}_{i,j}$ to accommodate the increase in $|\mathcal{I}_{i,j}|$ by one.

Note that from an implementation standpoint, no work need be done for intervals $\mathcal{I}_{i,j'}$ on insertion, even if they change sides. Any readjustment can be delayed until the following query in gap Δ_i.

3.7.2 Query

We now proceed with the analysis of a query. We split the analysis into several sections. We first assume the gap Δ_i in which the query falls is a 2-sided gap. We show Invariant (C) implies we can pay for the current query. We then show how to ensure Invariant (C) holds after the query. Finally, we make the necessary adjustments for the analysis of queries in 0-sided and 1-sided gaps. Recall that our complexity goal to split a gap into gaps of size x and cx ($c \geq 1$) is $O(\log n + x \log c)$ amortized time.

Current Query

For the moment, we assume the gap in which the query rank r satisfies $r \in \Delta_i$ is a 2-sided gap. Further, assume the query rank r falls left of the median of gap Δ_i, so that the resulting gaps are a gap Δ_i' of size x and a gap Δ_{i+1}' of size cx ($c \geq 1$). A picture is given in Figure 3.3. The case of query rank r falling right of the median of Δ_i is symmetric.

It takes $O(\log(n/|\Delta_i|)) = O(\log n)$ time via the data structure of Lemma 22 to find the gap Δ_i. We then find the interval $\mathcal{I}_{i,j}$ such that $r \in \mathcal{I}_{i,j}$. By Definition 18, answering the query on the set of unsorted elements $\mathcal{I}_{i,j}$ can be done in $O(|\mathcal{I}_{i,j}|)$ time. Splitting interval $\mathcal{I}_{i,j}$ as described in Section 3.6.5 can also be done in $O(|\mathcal{I}_{i,j}|)$ time.

Updating the data structure of Lemma 22 with the addition of gaps Δ_i' and Δ_{i+1}' and removal of gap Δ_i can be done in $O(\log n)$ time. Similarly, the total number of intervals

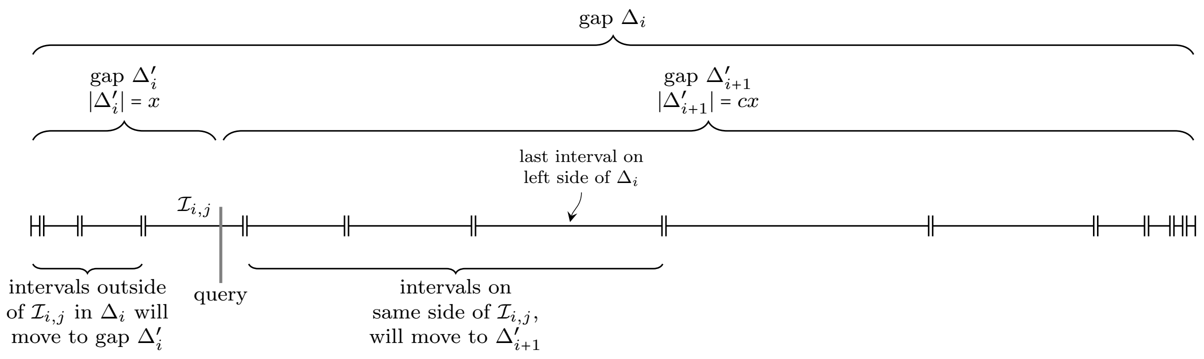
gap Δ_i
gap Δ'_i
$|\Delta'_i| = x$
gap Δ'_{i+1}
$|\Delta'_{i+1}| = cx$
last interval on left side of Δ_i
$\mathcal{I}_{i,j}$
intervals outside of $\mathcal{I}_{i,j}$ in Δ_i will move to gap Δ'_i
query
intervals on same side of $\mathcal{I}_{i,j}$, will move to Δ'_{i+1}

Figure 3.3: A query that splits $\mathcal{I}_{i,j}$ in gap Δ_i.

created from the current query is no more than 6, and no more than $O(\log|\Delta_i|)$ intervals existed in gap Δ_i prior to the query, again by Lemma 26. Thus, applying Rule (B) to gaps Δ'_i and Δ'_{i+1} after the query takes no more than $O(\log|\Delta_i|) = O(\log n)$ time, because merging two intervals can be done in $O(1)$ time.

We next show that merging of intervals according to Rule (B) will preserve Invariant (C).

Lemma 33 (Merge maintains (C))**.** *Suppose interval $\mathcal{I}_{i,j}$ is merged into interval $\mathcal{I}_{i,j'}$ (note $j' = j + 1$ if $\mathcal{I}_{i,j}$ is on the left side and $j' = j - 1$ if $\mathcal{I}_{i,j}$ is on the right side), according to Rule (B). Then the interval $\mathcal{I}_{i,j'}$ after the merge satisfies Invariant (C).*

Proof. Suppose interval $\mathcal{I}_{i,j}$ is merged into interval $\mathcal{I}_{i,j'}$ according to Rule (B). Then $o(\mathcal{I}_{i,j}) \geq |\mathcal{I}_{i,j}| + |\mathcal{I}_{i,j'}|$. This implies that after the merge, $o(\mathcal{I}_{i,j'}) \geq |\mathcal{I}_{i,j'}|$, since elements outside the merged interval $\mathcal{I}_{i,j'}$ are outside both of the original intervals. Thus $\mathcal{I}_{i,j'}$ satisfies Invariant (C) without any credits. □

In total, we pay $O(\log n + |\mathcal{I}_{i,j}|)$ actual time. As the $O(\log n)$ component is consistent with the $O(\log n)$ term in our desired query complexity, let us focus on the $O(|\mathcal{I}_{i,j}|)$ term. We have the following.

Lemma 34 (Amortized splitting cost)**.** *Consider a query which falls in interval $\mathcal{I}_{i,j}$ and splits gap Δ_i into gaps of size x and cx. Then $|\mathcal{I}_{i,j}| - c(\mathcal{I}_{i,j}) \leq x$.*

Proof. By Invariant (C), $|\mathcal{I}_{i,j}| - c(\mathcal{I}_{i,j}) \leq o(\mathcal{I}_{i,j})$. Now, since Δ_i is a 2-sided gap, $o(\mathcal{I}_{i,j})$ is the lesser of the number of elements left or right of $\mathcal{I}_{i,j}$. Since the query rank r satisfies $r \in \mathcal{I}_{i,j}$, this implies $o(\mathcal{I}_{i,j}) \leq x$ (See Figure 3.3 for a visual depiction). □

We can apply amortized analysis with Lemma 34 as follows. Interval $\mathcal{I}_{i,j}$ is destroyed and intervals that are built from its contents have no credits. Thus, $4c(\mathcal{I}_{i,j})$ units of potential are released. By applying Lemma 34, we can use $c(\mathcal{I}_{i,j})$ units of this released potential to bound the cost $|\mathcal{I}_{i,j}|$ with x. This gives an amortized cost thus far of $\log n + x - 3c(\mathcal{I}_{i,j})$. We will use the extra $3c(\mathcal{I}_{i,j})$ units of potential in the following section, ensuring Invariant (C) holds for future operations.

Ensuring Invariant (C)

We must ensure Invariant (C) holds on all intervals in gaps Δ'_i and Δ'_{i+1}. Again, we will suppose Δ'_i is the smaller gap of the two, so that Δ'_i has x elements and Δ'_{i+1} has cx elements; the other case is symmetric.

Let us first consider gap Δ'_i. This gap contains intervals from Δ_i outside of $\mathcal{I}_{i,j}$ as well as intervals made from the elements of $\mathcal{I}_{i,j}$. Observe (cf. Figure 3.3) that gap Δ'_i has in total x elements. Therefore, we can trivially ensure Invariant (C) holds by adding enough credits to each interval of Δ'_i to make it so, at total amortized cost at most $4x$. Let us do this after applying Rule (B) to Δ'_i, so it is balanced and satisfies Invariant (C).

We now consider gap Δ'_{i+1} after rebalancing according to Rule (B). The application of Rule (B) after the query may cause some intervals to change sides towards the query rank r and subsequently merge. Intervals created from $\mathcal{I}_{i,j}$ may also merge (this may be because Rule (B) was applied lazily or even because the largest interval created from $\mathcal{I}_{i,j}$ may be on the opposite side of the rest of the intervals created from interval $\mathcal{I}_{i,j}$). In total, the intervals of Δ'_{i+1} fall into four distinct categories. Recall that when we apply Rule (B), we merge an interval $\mathcal{I}_{i,j'}$ into interval $\mathcal{I}_{i,j''}$, so we assume the identity of the merged interval as $\mathcal{I}_{i,j''}$, and interval $\mathcal{I}_{i,j'}$ ceases to exist.

We call the four categories A, B, C, and D, and show how to ensure Invariant (C) on each of them. Category A are intervals that are created from $\mathcal{I}_{i,j}$ that fall on the side of the query rank r so as to become Δ'_{i+1} intervals after the query. Category B are intervals on the same side as $\mathcal{I}_{i,j}$ before the query which were located inward from $\mathcal{I}_{i,j}$ in Δ_i. Category C are intervals that were on the opposite side of interval $\mathcal{I}_{i,j}$ before the query, but now switch sides due to the removal of the gap Δ'_i. Finally, category D are intervals that lie on the opposite side of interval $\mathcal{I}_{i,j}$ both before and after the query. A picture is given in Figure 3.4.

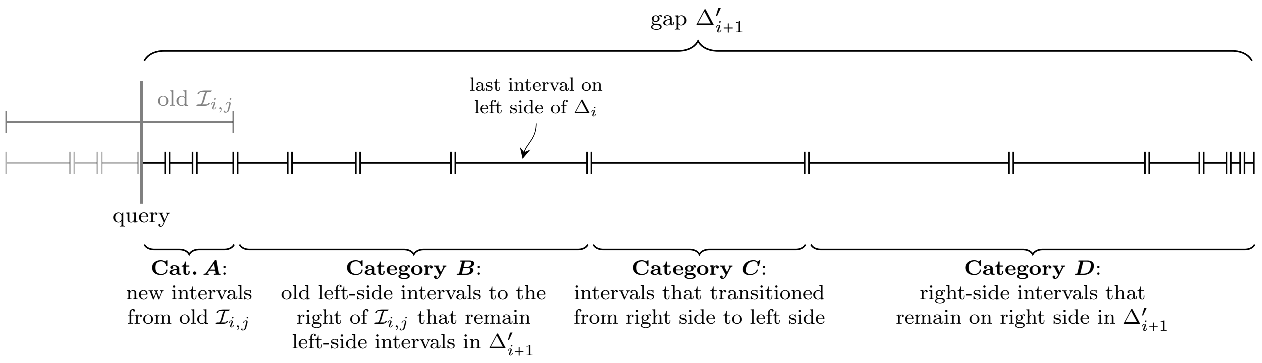
gap Δ'_{i+1}
old $\mathcal{I}_{i,j}$
last interval on left side of Δ_i
query
Cat. *A*: new intervals from old $\mathcal{I}_{i,j}$
Category *B*: old left-side intervals to the right of $\mathcal{I}_{i,j}$ that remain left-side intervals in Δ'_{i+1}
Category *C*: intervals that transitioned from right side to left side
Category *D*: right-side intervals that remain on right side in Δ'_{i+1}

Figure 3.4: Gap Δ'_{i+1} after query within interval $\mathcal{I}_{i,j}$ of Δ_i. The picture assumes $\mathcal{I}_{i,j}$ was a left-side interval.

We proceed with ensuring Invariant (C) on each category.

- **Category *D***: Category D intervals are easiest. These intervals are not affected by the query and thus still satisfy Invariant (C) with no additional cost.

- **Category *A***: Now consider category A intervals. Three such intervals, $\mathcal{I}'_{i+1,1}$, $\mathcal{I}'_{i+1,2}$, and $\mathcal{I}'_{i+1,3}$, are created in the query algorithm stated in Section 3.6.5. The leftmost and middlemost intervals, $\mathcal{I}'_{i+1,1}$ and $\mathcal{I}'_{i+1,2}$, have size $\frac{1}{4}|R| \pm 1$, and the rightmost interval $\mathcal{I}'_{i+1,3}$ has size $\frac{1}{2}|R| \pm 1$; (the ± 1 addresses the case that $|R|$ is not divisible by 4).

  Up to one element, $\mathcal{I}'_{i+1,2}$ has at least as many elements outside of it as within it. Thus after giving it one credit, $\mathcal{I}'_{i+1,2}$ satisfies Invariant (C). Similarly, $\mathcal{I}'_{i+1,3}$ will remain on the same side in most cases, and thus will also have enough elements outside it from the other two intervals (potentially after giving it one credit, too). But we always have to assign credits to $\mathcal{I}'_{i+1,1}$. Moreover, if interval $\mathcal{I}_{i,j}$ was very large, then $\mathcal{I}'_{i+1,3}$ may actually switch sides in the new gap Δ'_{i+1}.

  In the worst case, we will require credits to satisfy Invariant (C) on both $\mathcal{I}'_{i+1,1}$ and $\mathcal{I}'_{i+1,3}$. As their sizes total $\frac{3}{4}|R| + O(1)$, at 4 units of potential per credit the amortized cost to do so is no more than $3|\mathcal{I}_{i,j}| + O(1)$. We can use the extra $3c(\mathcal{I}_{i,j})$ units of potential saved from Section 3.7.2 to pay for this operation. By applying Lemma 34 again, we can bound $3|\mathcal{I}_{i,j}| - 3c(\mathcal{I}_{i,j})$ with $3x$, bringing the amortized cost of satisfying Invariant (C) on category A intervals to $O(x)$.

- **Categories *B* and *C***: We'll handle category B and C intervals together. First observe that since x elements were removed with the query, we can bound the number of credits necessary to satisfy Invariant (C) on a single interval in category B or C with either x or the size of that interval. For category C intervals, this follows because they had more elements on their left side prior to the query, thus upon switching sides after the query, x credits will suffice to satisfy Invariant (C), similarly to the proof of Lemma 32. In the new gap Δ'_{i+1}, let j' be the smallest index such that $|\mathcal{I}_{i+1,j'}| \geq x$. We will handle category B and C intervals left of $\mathcal{I}_{i+1,j'}$ and right of $\mathcal{I}_{i+1,j'}$ differently.

Let us first consider category B and C intervals left of interval $\mathcal{I}_{i+1,j'}$. All such intervals have size less than x. If there are less than two such intervals, we may apply x credits to each to ensure Invariant (C) at total cost $O(x)$. Otherwise, consider intervals $\mathcal{I}_{i+1,j'-2}$ and $\mathcal{I}_{i+1,j'-1}$. Due to application of Rule (B) after the query, intervals $\mathcal{I}_{i+1,j'-2}$ and $\mathcal{I}_{i+1,j'-1}$ make up more than half of the total number of elements left of interval $\mathcal{I}_{i+1,j'}$. Since $|\mathcal{I}_{i+1,j'-2}| < x$ and $|\mathcal{I}_{i+1,j'-1}| < x$, it follows there are no more than $4x$ elements located in intervals left of interval $\mathcal{I}_{i+1,j'}$ in gap Δ'_{i+1}. For each such interval, we add at most the size of the interval in credits so that Invariant (C) holds on all intervals left of $\mathcal{I}_{i+1,j'}$ in gap Δ'_{i+1}. The total cost is $O(x)$.

Now consider intervals right of $\mathcal{I}_{i+1,j'}$. If there are less than two such intervals, we may apply x credits to each to ensure Invariant (C) at total cost $O(x)$. Otherwise, consider intervals $\mathcal{I}_{i+1,j'+1}$ and $\mathcal{I}_{i+1,j'+2}$. By Rule (B) after the query, $|\mathcal{I}_{i+1,j'+1}| + |\mathcal{I}_{i+1,j'+2}| > x$, since interval $\mathcal{I}_{i+1,j'}$ is outside intervals $\mathcal{I}_{i+1,j'+1}$ and $\mathcal{I}_{i+1,j'+2}$ and $|\mathcal{I}_{i+1,j'}| \geq x$ by choice of j'. Similarly, if such intervals are category B or C intervals, then $|\mathcal{I}_{i+1,j'+3}| + |\mathcal{I}_{i+1,j'+4}| > 2x$ and $|\mathcal{I}_{i+1,j'+5}| + |\mathcal{I}_{i+1,j'+6}| > 4x$. In general, $|\mathcal{I}_{i+1,j'+2k-1}| + |\mathcal{I}_{i+1,j'+2k}| > 2^{k-1}x$ for any k where intervals $\mathcal{I}_{i+1,j'+2k-1}$ and $\mathcal{I}_{i+1,j'+2k}$ are category B or C intervals. Since there are cx total elements in gap Δ'_{i+1}, it follows the number of category B and C intervals right of $\mathcal{I}_{i+1,j'}$ is $O(\log c)$. We may then apply x credits to all such intervals and interval $\mathcal{I}_{i+1,j'}$ for a total cost of $O(x \log c)$.

Altogether, we can ensure Invariant (C) for future iterations at total $O(x \log c)$ amortized cost.

0-Sided and 1-Sided Gaps

We proceed with a generalization of the previous two sections for when the gap Δ_i in which the query falls is a 0-sided or 1-sided gap. If gap Δ_i is 0-sided, we spend $O(n)$ time to answer the query, according to Definition 18 on a set of n unsorted elements. Since Invariant (C) is satisfied prior to the query, $4n$ credits are released. Quantity M does not change. Thus, $4n$ units of potential are released, giving amortized time $n - 4n = -3n$. All intervals in the data structure resulting from the query are category A intervals. The analysis of the preceding section for category A intervals applies. We can pay $O(x)$ to satisfy Invariant (C) on the smaller gap, and the remaining $3n$ units of released potential are enough to guarantee Invariant (C) holds on all intervals in the larger gap.

Now suppose Δ_i is 1-sided. If the query rank r is closer to the side of Δ_i on which queries have been performed, then the same analysis of the preceding sections suffices. Note that there will be neither category C nor category D intervals. The creation of 2-sided gap Δ_i' out of elements of 1-sided gap Δ_i will cause $10x$ additional units of potential to be released due to the decrease in M; these units are not used in this case.

We are left with the case Δ_i is 1-sided and the query rank r is closer to the side of Δ_i on which queries have not been performed; suppose without loss of generality that previously only the right endpoint of Δ_i has been queried and r is closer to the left endpoint. In this case, the creation of 2-sided gap Δ_{i+1}' out of elements of 1-sided gap Δ_i will cause $10cx$ units of potential to be released due to the decrease in M. Since $c \geq 1$, this is at least $5|\Delta_i|$ units of potential. We use them as follows. Answering the query takes no more than $O(|\Delta_i|)$ time, and ensuring intervals satisfy Invariant (C) in new gaps Δ_i' and Δ_{i+1}' after the query similarly takes no more than $|\Delta_i|$ credits, which costs $4|\Delta_i|$ units of potential. Thus, in total this takes no more than $|\Delta_i| + 4|\Delta_i| - 5|\Delta_i| = O(1)$ amortized time.

Putting the preceding three sections together, we may answer a query in $O(\log n + x \log c)$ time while ensuring Invariant (C) for future operations.

3.7.3 Deletion

The analysis of deletion of $e = (k, v)$ pointed to by `ptr` is as follows. The element e can be removed from the interval in which it resides in $O(1)$ time. Removing said interval lazily, if applicable, takes $O(1)$ time. If the gap in which e resides also needs removal, Lemma 22 says doing so will take $O(\log n)$ time.

In any case, when element $e \in \Delta_i$ is deleted, we must ensure Invariant (C) on the remaining intervals of Δ_i. If e was outside of an interval $\mathcal{I}_{i,j}$, $o(\mathcal{I}_{i,j})$ decreases by one. Thus, for any such intervals, we pay one credit to ensure Invariant (C) remains satisfied. Thus in accordance with Lemma 26, this takes $O(\log |\Delta_i|)$ total credits.

The total amortized cost is thus no more than $O(\log n + \log |\Delta_i|) = O(\log n)$. If the data structure of Lemma 22 supports operations in worst-case time, this runtime is also worst-case.

3.7.4 Change-Key

We analyze the change-key operation as follows. Suppose `ptr` points to element $e = (k, v)$ and we change its key as described in Section 3.6.7 to k'. If k' falls outside gap Δ_i, $O(\log n)$

complexity follows from deletion and re-insertion of (k', v). Otherwise, the binary search in Δ_i takes $O(\log\log|\Delta_i|)$ time, again by Lemma 26. To ensure Invariant (C) on the intervals of Δ_i, as is the case for deletion, we must pay one credit per interval e is no longer outside of. Thus, the key-change operation takes at most $O(\log|\Delta_i|)$ time; however, if we change the key of e towards the nearest query rank, we can show Invariant (C) is satisfied without spending any credits.

At any point in time, all intervals in Δ_i are classified as being on the left side or the right side according to the closest query rank, in accordance to Rule (A). Any element of a left-side interval can have its key decreased, while only increasing or keeping constant the number of elements outside of any other left-side interval. The same is true for key increases of elements in right side intervals.

Now consider if $e \in \mathcal{I}_{i,j}$ and $\mathcal{I}_{i,j}$ is the rightmost interval on the left side. Then we can also increase the key of e while keeping the same or increasing the number of elements outside of any interval in Δ_i. The same is true of decreasing the key of an element in the leftmost interval on the right side. Since the median of Δ_i falls in either the leftmost interval of the right side or the rightmost interval of the left side, it follows that we can ensure Invariant (C) as long as the element whose key changes moves closer to its nearest query rank. Note that this analysis holds even as intervals change side designations due to insertions; for a refresher of this analysis see the proof of Lemma 32. This is despite delaying the application of Rule (B) until the following query in gap Δ_i.

This proves our statement in Theorem 13 about change-key. The dichotomy displayed therein between cheap and expensive key changes can be refined as follows. Suppose $c \geq 2$ is such that e is located between (gap-local) ranks $|\Delta_i|/c$ and $|\Delta_i| - |\Delta_i|/c$ in Δ_i; then we can change its key *arbitrarily* in $O(\log\log\Delta_i + \log c)$ time. This is because of the geometric nature of interval sizes. Intervals are highly concentrated close to the edges of gap Δ_i in order to support queries that increase B very little, efficiently. Thus, we can support arbitrary key changes in $O(\log\log|\Delta_i|)$ time for the vast majority of the elements of gap Δ_i, since ensuring Invariant (C) will only require a constant number of credits, and the performance smoothly degrades as the changed elements get closer to previous query ranks.

A second refinement is that we can change e arbitrarily without paying any credits if an insertion closer to the endpoint of gap Δ_i has happened before said key-change, but after the query that created Δ_i: such insertion increases the number of elements outside of all intervals that are potentially affected by moving e closer to the middle of Δ_i, thus no credits have to be supplied. A similar argument shows that the time complexity of deletion is only $O(1)$ if an element was previously inserted closer to the gap endpoint than the deleted element. We point out again that, from the perspective of the data structure, these savings are realized automatically and the data structure will always run as efficiently as possible; the credits are only an aspect of the analysis, not our algorithms.

In the following section, we show that a bound on the number of created intervals can bound the number of pointers required of the data structure and the insertion and change-key complexities when the number of queries is small.

3.7.5 Pointer Bound and Improved Insertion and Change-Key

The preceding sections show insertion into gap Δ_i in $O(\log(n/|\Delta_i|) + \log\log|\Delta_i|)$ time and a change-key time complexity of $O(\log\log|\Delta_i|)$. A bound of $O(\log q)$ can also be made, which may be more efficient when q is small. We also prove the bound stated in Theorem 14 on the total number of pointers required of the data structure. We address the latter first.

Proof of Theorem 14. Each query (including `Split(`r`)` queries) creates at most 6 intervals, and no other operations create intervals. The number of pointers required of all interval data structures is linear in the number of total intervals created, bounded to at most n. This is because elements within an interval are contiguous (in the sense an expandable array is contiguous) unless the interval is a result of merged intervals, where we assume that intervals are implemented as linked lists of arrays. Each merged interval must have been created at some point in time, thus the bound holds. The number of pointers required in the data structure of Lemma 22 is linear in the number of gaps (or intervals, if the data structure operates directly over intervals), taking no more than $O(\min(q, n))$ pointers, as the number of intervals is $O(\min(q, n))$. □

The above proof shows that the number of intervals and gaps in the entire data structure can be bounded by q. This implies the binary searches during insertion (both in the data structure of Lemma 22 and in Section 3.6.4) and change-key operations take no more than $O(\log q)$ time. This gives a refined insertion time bound of $O(\min(\log(n/|\Delta_i|) + \log\log|\Delta_i|, \log q))$ and a change-key time bound of $O(\min(\log q, \log\log|\Delta_i|))$. To guarantee an $O(\log q)$ time bound in the gap data structure, we can maintain all gaps additionally in a standard balanced BST, with pointers between corresponding nodes in both data structures. A query can alternatively advance from the root in both structures, succeeding as soon as one search terminates. Updates must be done on both structures, but the claimed $O(\log n)$ time bounds (for queries, delete, split, and merge) permit this behavior.

3.8 Bulk Update Operations

Lazy search trees can support binary search tree bulk-update operations. We can split a lazy search tree at a rank r into two lazy search trees T_1 and T_2 of r and $n - r$ elements, respectively, such that for all $x \in T_1$, $y \in T_2$, $x \leq y$. We can also support a merge of two lazy search trees T_1 and T_2 given that for all $x \in T_1$, $y \in T_2$, $x \leq y$.

We state this formally in Lemma 35.

Lemma 35. *Operation* `Split(`r`)` *can be performed on a lazy search tree in time the same as* `RankBasedQuery(`r`)`*. Operation* `Merge(`T_1`,`T_2`)` *can be performed on lazy search trees in* $O(\log n)$ *worst-case time.*

Proof. To perform operation `Split(`r`)`, we first query for rank r in the lazy search tree. We then split the data structure of Lemma 22 at the separation in gaps induced by the query for rank r. Two lazy search trees result, with their own future per-operation costs according to the number of elements and gaps that fall into each tree. Using a globally-biased $2, b$ tree [24] with weights as in the proof of Lemma 22, the split takes $O(\log n)$ worst-case time (Theorem 10 of [24]). The overall time complexity is dominated by the query for rank r in the original tree, since queries take $\Omega(\log n)$ time.

To perform operation `Merge(`T_1`,`T_2`)`, we perform a merge on the data structures of Lemma 22 associated with each lazy search tree. Future per-operation costs are adjusted according to the union of all gaps and totaling of elements in the two lazy search trees that are combined. Using a globally-biased $2, b$ tree [24] with weights as in the proof of Lemma 22, the merge takes $O(\log n)$ worst-case time or better (Theorem 8 of [24]). □

Lemma 35 completes the analysis for the final operations given in Theorem 13.

3.9 Average Case Insertion and Change-Key

Our time bounds from Theorem 13 are an additive $O(\log \log n)$ away from the optimal time of insertion and change-key; it turns out that in certain average-case scenarios, we can indeed reduce this time to an optimal *expected* amortized time. The essential step will be to refine the binary search within a gap to an exponential search.

3.9.1 Insert

Recall that we store intervals in a sorted array. We modify the insertion algorithm of the interval data structure in Section 3.6.4 so that we instead perform a *double binary search* (also called *exponential search* [27]), outward from the last interval on the left side and first interval on the right side. This is enough to prove the following result.

Theorem 36 (Average-case insert)**.** *Suppose the intervals within a gap are balanced using Rule (B) and further suppose insertions follow a distribution such that the gap in which an* *inserted element falls can be chosen adversarially, but amongst the elements of that gap, its rank is chosen uniformly at random. Then insertion into gap Δ_i takes expected time $O(\log(n/|\Delta_i|))$.*

Proof. First note that the double binary search during insertion finds an interval that is k intervals from the middlemost intervals in time $O(\log k)$; apart from constant factors, this is never worse than the $O(\log \ell_i)$ of a binary search.

The assumption on insertion ranks implies that the probability to insert into interval $\mathcal{I}_{i,j}$ (out of the possible ℓ_i intervals in gap Δ_i) is $|\mathcal{I}_{i,j}|/|\Delta_i| \pm O(1/|\Delta_i|)$, i.e., proportional to its size. Recall that in a gap Δ_i satisfying Lemma 26, interval sizes grow at least like $(\sqrt{2})^k$; that implies the largest (middlemost) intervals contain a constant fraction of the elements in Δ_i; for these, insertion takes $O(1)$ time. The same applies recursively: With every outward step taken, the insertion procedure takes $O(1)$ more time, while the number of elements that fall in these intervals decreases by a constant factor. The expected insertion time in the interval data structure is proportional to

$$\sum_{k=1}^{\infty} \frac{\log k}{(\sqrt{2})^k} \le \sum_{k=1}^{\infty} \frac{k}{(\sqrt{2})^k} = 4 + 3\sqrt{2},$$

i.e., constant overall. Adding the $O(\log(n/|\Delta_i|))$ time to find the gap yields the claim. $\square$

Observe that walking from the largest intervals outward, instead of performing an exponential search [27], is sufficient for the above analysis. However, the exponential search also satisfies the worst case $O(\log\log n)$ bound (more precisely $O(\min(\log\log|\Delta_i|, \log q))$) described in Sections 3.6.4 and 3.7.1.

Remark 37 (Fast insertion without arrays)**.** *We can achieve the same effect if intervals are stored in another biased search tree so that interval $\mathcal{I}_{i,j}$ receives weight $|\Delta_i|/\ell_i + |\mathcal{I}_{i,j}|$.*

Theorem 36 assumes that intervals are balanced according to Rule (B). In Section 3.6, we described balancing according to Rule (B) lazily. Keeping (B) balanced while insertions or change-key operations occur, in the required time complexity, is nontrivial. We show it can be done in $O(1)$ amortized time below.

Lemma 38 (Strict merging)**.** *Given a gap Δ_i, we can keep intervals in Δ_i balanced according to within a constant factor of the guarantee of Rule (B) in $O(1)$ amortized time per insertion into Δ_i.*

Proof. We utilize the exponentially-increasing interval sizes due to Lemma 26. We check the outermost intervals about every operation and exponentially decrease checking frequency as we move inwards. The number of intervals checked over k operations is $O(k)$.

The guarantee of Rule (B) is changed so that the number of elements left of $\mathcal{I}_{i,j}$ in Δ_i is no more than a constant times $|\mathcal{I}_{i,j}| + |\mathcal{I}_{i,j+1}|$ (reflected for right side intervals), to which previous analysis holds. □

3.9.2 Change-Key

If we apply Lemma 38, we can also support improved average-case change-key operations in the following sense.

Theorem 39 (Average-case change-key)**.** *If a* `ChangeKey(ptr, `k'`)` *operation is performed such that the element pointed to by* `ptr`*,* $e = (k, v)$*, moves closer to its closest query rank within its gap and the rank of* k' *is selected uniformly at random from valid ranks, the operation can be supported in* $O(1)$ *expected time.*

Proof. We again perform a double binary search (exponential search [27]) for the new interval of e; this time we start at the interval $\mathcal{I}_{i,j}$ in which e currently resides and move outwards from there. The analysis follows similarly to Theorem 36. □

When used as a priority queue, Theorem 39 improves the average-case complexity of decrease-key to $O(1)$.

3.10 Randomized-Selection Variant

We can improve the practical efficiency of lazy search trees by replacing exact median-finding in the query procedure with randomized pivoting. Specifically, after finding sets L and R as described in Section 3.6.5, we then partition L into sets L_l and L_r by picking a random element $p \in L$ and pivoting so that all elements less than p are placed in set L_l and all elements greater than p are placed in set L_r. To avoid biasing when elements are not unique, elements equal to p should be split between L_l or L_r. We then repeat the procedure one more time on set L_r. We do the same, reflected, for set R.

Remark 40 (Partitioning with equal keys)**.** *In our analysis, we assume for simplicity that the number of elements with same key as* p*, including* p *itself, that are assigned to the left segment is chosen uniformly at random from the number of copies. That implies overall a uniform distribution for the size of the segments. Partitioning procedures as used in standard implementations of quicksort [212] actually lead to slightly more balanced splits [211]; they will only perform better. For practical implementations of lazy search trees, choosing the partitioning element* p *as the median of a small sample is likely to improve overall performance.*

Changing the query algorithm in this way requires a few changes in our analysis. The analysis given in Section 3.7 is amenable to changes in constant factors in several locations. Let us generalize the potential function as follows, where α is a set constant, such as $\alpha = 4$ in Section 3.7. One can see from Section 3.7.2 that this will imply the constant in front of M must be at least $2(\alpha + 1)$.

$$\Phi \;=\; 2(\alpha+1)M \;+\; \alpha \sum_{\substack{1\le i\le m,\\ 1\le j\le \ell_i}} c(\mathcal{I}_{i,j}).$$

Insertion still takes $O(\min(\log(n/|\Delta_i|) + \log\log|\Delta_i|, \log q))$ time. As before, splitting into sets L and R can typically be done in $O(|\mathcal{I}_{i,j}|)$ deterministic time via the result of the query, but if not, quickselect can be used for $O(|\mathcal{I}_{i,j}|)$ expected (indeed with high probability) time performance [136, 94, 159]. The modified pivoting procedure described above for L_l and L_r is repeated in total 4 times. We can thus bound the complexity of these selections at $O(|\mathcal{I}_{i,j}|)$, regardless of the randomization used.

Then by application of Lemma 34, we reduce the current amortized time to split $\mathcal{I}_{i,j}$ to $O(x)$, leaving $(\alpha - 1)c(\mathcal{I}_{i,j})$ units of potential to handle ensuring Invariant (C) on category A intervals in Section 3.7.2.

The number of credits necessary to satisfy Invariant (C) on category A intervals is now a random variable. Recall the arguments given in Section 3.7.2 and Section 3.7.2 regarding category A intervals. As long as the (expected) number of credits to satisfy Invariant (C) on category A intervals is at most a constant fraction γ of $|\mathcal{I}_{i,j}|$, we can set $\alpha = \frac{1}{1-\gamma}$ and the amortized analysis carries through.

We have the following regarding the expected number of credits to satisfy Invariant (C) on category A intervals using the randomized splitting algorithm.

Lemma 41. *Suppose a query falls in interval $\mathcal{I}_{i,j}$ and the intervals built from the elements of $\mathcal{I}_{i,j}$ are constructed using the randomized splitting algorithm. The expected number of credits necessary to satisfy Invariant (C) on category A intervals after a query is no more than $\frac{143}{144}|\mathcal{I}_{i,j}| + O(1)$.*

Proof. We prove the loose bound considering only one random event in which a constant fraction of $|\mathcal{I}_{i,j}|$ credits are necessary, which happens with constant probability.

We orient as in Section 3.7.2, assuming the larger new gap, Δ'_{i+1}, is right of the smaller new gap, Δ'_i. We must consider the number of credits necessary to satisfy Invariant (C) on the three category A intervals $\mathcal{I}'_{i+1,1}$, $\mathcal{I}'_{i+1,2}$, and $\mathcal{I}'_{i+1,3}$ of new gap Δ'_{i+1}. The rightmost interval $\mathcal{I}'_{i+1,3}$ has size drawn uniformly at random in $1, \ldots, |R|$, the leftmost, $\mathcal{I}'_{i+1,1}$, takes size uniformly at random from the remaining elements, and the middlemost interval $\mathcal{I}'_{i+1,2}$ takes whatever elements remain.

Suppose the rightmost interval $\mathcal{I'}_{i+1,3}$ comprises a fraction of $x = |\mathcal{I}'_{i+1,3}|/|R| \in \left[\frac{1}{3}, \frac{2}{3}\right]$ of all elements in R, and further suppose the leftmost interval $\mathcal{I'}_{i+1,1}$ takes between 1/2 and 3/4 of the remaining elements, i.e., a fraction $y = |\mathcal{I}'_{i+1,1}|/|R| \in \left[\frac{1}{2}(1-x), \frac{3}{4}(1-x)\right]$ of the overall elements in R. In this case, it is guaranteed we require no credits to satisfy Invariant (C) on the middlemost interval. The number of credits to satisfy Invariant (C) on the rightmost and leftmost intervals is $(x+y)|R|$, which is maximized at $\frac{11}{12}|R|$. This event happens with probability $\frac{1}{3} \cdot \frac{1}{4} - O(1/|R|) = \frac{1}{12} - O(1/|R|)$, where we include the $O(1/|R|)$ term to handle rounding issues with integer values of $|R|$. As we never require more than $|R|$ credits in any situation and $|R| \le |\mathcal{I}_{i,j}|$, we can then bound the expected number of necessary credits at $\frac{11}{12} \cdot |\mathcal{I}_{i,j}| + \frac{1}{12} \cdot \frac{11}{12}|\mathcal{I}_{i,j}| + O(1) = \frac{143}{144}|\mathcal{I}_{i,j}| + O(1)$. □

With Lemma 41, we can set $\alpha = 144$ and use the remaining $143c(\mathcal{I}_{i,j})$ credits from destroying $\mathcal{I}_{i,j}$ and bound $143|\mathcal{I}_{i,j}| - 143c(\mathcal{I}_{i,j})$ with $143x$ via Lemma 34. All other query analysis in Section 3.7.2 is exactly as before. This gives total expected amortized query time $O(\log n + x \log c)$ on 2-sided gaps. With a constant of $2(\alpha+1)$ in front of M in the generalized potential function, the analysis for 0 and 1-sided gaps in Section 3.7.2 carries through.

Putting it all together, we get the following result.

Theorem 42 (Randomized splitting)**.** *If partitioning by median in the query algorithm is replaced with splitting on random pivots, lazy search trees satisfy the same time bounds, in worst-case time, as in Theorem 13, except that* `RankBasedQuery(r)` *and* `Split(r)` *now take $O(\log n + x \log c)$ expected amortized time.*

Note that another possible approach is to change Invariant (C) to something like $c(\mathcal{I}_{i,j}) + 2o(\mathcal{I}_{i,j}) \ge |\mathcal{I}_{i,j}|$, which gives further flexibility in the rest of the analysis. This is, however, not necessary to prove Theorem 42.

3.11 Lazy Splay Trees

Splay trees [217] are arguably the most suitable choice of a biased search tree in practice; we thereby explore their use within lazy search trees in this section. We show that an amortized-runtime version of Lemma 22 can indeed be obtained using splay trees. We also show that by using a splay tree, the efficient access theorems of the splay tree are achieved automatically by the lazy search tree. This generalizes to any biased search tree that is used as the data structure of Lemma 22.

3.11.1 Splay Trees For The Gap Data Structure

We show that splay trees can be used as the gap data structure.

Lemma 43 (Splay for Gaps)**.** *Using splay trees as the data structure for the set of gaps* $\{\Delta_i\}$ *allows support of all operations listed in Lemma 22, where the time bounds are satisfied as amortized runtimes over the whole sequence of operations.*

Proof. We use a splay tree [217] and weigh gap Δ_i with $w_i = |\Delta_i|$. The sum of weights, W, is thus equal to n. Operation 1 can be supported by searching with $e = (k, v)$ into the tree until gap Δ_i is found and then splayed. According to the Access Lemma [217], this is supported in $O(\log(n/|\Delta_i|))$ amortized time. Operation 2 requires a weight change on gap Δ_i. By first accessing gap Δ_i, so that it is at the root, and then applying a weight change, this operation can be completed in time proportional to the access. According to the Access Lemma [217] and the Update Lemma [217], this will then take $O(\log(n/|\Delta_i|))$ amortized time. Note that for our use of operation 2, the element will already have just been accessed, so the additional access is redundant. Operations 3 and 4 are supported in $O(\log n)$ time by the Update Lemma [217]. Note that when the gap data structure is used in a lazy search tree, it always starts empty and more gaps are added one by one when answering queries. Hence any sequence of operations arising in our application will access every element in the splay tree at least once. □

Note that a bound of $O(\log q)$ amortized cost for all operations also holds by using equal weights in the analysis above (recall that in splay trees, the node weights are solely a means for analysis and do not change the data structure itself).

3.11.2 Efficient Access Theorems

We now specify a few implementation details to show how lazy search trees can perform accesses as fast as the data structure of Lemma 22 (resp. Lemma 43).

If an element e is the result of a query for a second time, then during that second access, e is the largest element in its gap. Instead of destroying that gap, we can assume the identity of the gap e falls into after the query to be the same gap in which e previously resided (depending on implementation, this may require a key change in the data structure of Lemma 22, but the relative ordering of keys does not change). In this way, repeated accesses to elements directly correspond to repeated accesses to nodes in the data structure of Lemma 22. Further, implementation details should ensure that no restructuring occurs in the interval data structure when an element previously accessed is accessed again. This is implied by the algorithms in Section 3.6, but care must be taken in the case of duplicate elements. This will ensure accessing a previously-accessed element will take $O(1)$ time in the interval data structure.

With these modifications, the lazy search tree assumes the efficient access properties of the data structure of Lemma 22. We can state this formally as follows.

Theorem 44 (Access Theorem)**.** *Given a sequence of element accesses, lazy search trees perform the access sequence in time no more than an additive linear term from the data structure of Lemma 22, disregarding the time to access each element for the first time.*

Proof. Once every item has been accessed at least once, the data structures are the same, save for an extra $O(1)$ time per access in the interval data structure. The cost of the first access may be larger in lazy search trees due to necessary restructuring. $\square$

While we would ideally like to say that lazy search trees perform within a constant factor of splay trees on any operation sequence, this is not necessarily achieved with the data structure as described here. Time to order elements on insertion is delayed until queries, implying on most operation sequences, and certainly in the worst case, that lazy search trees will perform within a constant factor of splay trees, often outperforming them by more than a constant factor. However, if, say, elements $1, 2, \ldots, n$ are inserted in order in a splay tree, then accessed in order $n, n-1, \ldots, 1$, splay trees perform the operation sequence in $O(n)$ time, whereas lazy search trees as currently described will perform the operation sequence in $O(n \log n)$ time.

Theorem 44 shows using a splay tree for the gap data structure (Lemma 43) allows lazy search trees to achieve its efficient-access theorems. Observing that the initial costs of first access to elements total $O(n \log n)$, we achieve Corollary 45 below.

Corollary 45. *Suppose a splay tree is used as the gap data structure. Then lazy search trees achieve the efficient access theorems of the splay tree, including static optimality, static finger, dynamic finger, working set, scanning theorem, and the dynamic optimality conjecture [217, 62, 61, 81].*

3.12 Conclusion and Open Problems

We have discussed a data structure that improves the insertion time of binary search trees, when possible. Our data structure generalizes the theories of efficient priority queues and binary search trees, providing powerful operations from both classes of data structures. As either a binary search tree or a priority queue, lazy search trees are competitive. From a theoretical perspective, our work opens the door to a new theory of insert-efficient order-based data structures.

This theory is not complete. Our runtime can be as much as an additive $O(n \log\log n)$ term from optimality in the model we study, providing $O(\log\log n)$ time insert and decrease-key operations as a priority queue when $O(1)$ has been shown to be possible [99]. Further room for improvement is seen in our model itself, where delaying insertion work further can yield improved runtimes on some operation sequences. We see several enticing research directions around improving these shortcomings and extending our work. We list them as follows:

1. Extend our model and provide a data structure so that the order of operations performed is significant. A stronger model would ensure that the number of comparisons performed on an inserted element depends only on the queries performed after that element is inserted.

2. Within the model we study, improve the additive $O(n \log\log n)$ term in our analysis to worst-case $O(n)$, or give a lower bound that shows this is not possible while supporting all the operations we consider.

3. Explore and evaluate competitive implementations of lazy search trees. In the priority queue setting, evaluations should be completed against practically efficient priority queues such as binary heaps [236], Fibonacci heaps [99], and pairing heaps [101]. On binary search tree workloads with infrequent or non-uniformly distributed queries, evaluations should be completed against red-black trees [22], AVL trees [5], and splay trees [217].

4. Support efficient general merging of unordered data. Specifically, it may be possible to support $O(1)$ or $O(\log n)$ time merge of two lazy search trees when both are used as either min or max heaps.

5. Although the complexity of a rank-based query must be $\Omega(n)$ when the query falls in a gap of size $|\Delta_i| = \Omega(n)$, the per-operation complexity of `RankBasedQuery(`r`)` could potentially be improved to $O(x \log c + \log n)$ worst-case time instead of amortized time, with x and c defined as in Theorem 13.

6. Develop an external memory version of lazy search trees for the application of replacing B-trees [22], B^ϵ trees [43], or log-structured merge trees [193] in a database system.

7. Investigate multidimensional geometric data structures based off lazy search trees. Range trees [28], segment trees [26], interval trees [78, 176], kd-trees [25], and priority search trees [177] are all based on binary search trees. By building them off lazy search trees, more efficient runtimes as well as space complexity may be possible.

Regarding point 3, we have implemented a proof-of-concept version of a lazy search tree in C++, taking no effort to optimize the data structure. Our implementation is roughly 400 lines of code not including the gap data structure, to which we use a splay tree [217]. Intervals are split via randomized pivoting, as described in Section 3.10. The optimization to support $O(1)$ time average case insertion into the interval data structure is implemented, and the data structure also satisfies the $O(\min(q, n))$ pointer bound by representing data within intervals in a linked list of C++ vectors.

Our implementation has high constant factors for both insertion and query operations. For insertion, this is likely due to several levels of indirection, going from a gap, to an interval, to a linked list node, to a dynamically-sized vector. For query, this is likely due to poor memory management. Instead of utilizing swaps, as in competitive quicksort routines, our implementation currently emplaces into the back of C++ vectors, a much slower operation. The current method of merging also suggests some query work may be repeated, which although we have shown does not affect theoretical analysis, may have an effect in practice.

Still, initial experiments are promising. Our implementation outperforms both the splay tree which our implementation uses internally as well as C++ `set`, for both low query load scenarios and clustered queries. To give a couple data points, on our hardware, with $n = 1\,000\,000$, our implementation shaves about 30% off the runtime of the splay tree when no queries are performed and remains faster for anything less than about 2 500 uniformly distributed queries. When $n = 10\,000\,000$, our implementation shaves about 60% off the runtime of the splay tree when no queries are performed and remains faster for anything less than about 20 000 uniformly distributed queries. The C++ `set` has runtime about 30% less than our splay tree on uniformly distributed query scenarios. Our experiments against C++ STL `priority_queue` show that our current implementation is not competitive.

Finally, regarding points 2 and 4, we have succeeded in devising a solution that removes the $O(\log \log n)$ factors of the approach discussed herein; the new solution also supports constant time priority queue merge.

Part II

Range Mode

Chapter 4

Improved Time/Space Bounds for Dynamic Range Mode

4.1 Introduction

The mode of a sample is a fundamental data statistic along with median and mean. Given an ordered sequence, the range mode of interval $[l, r]$ is the mode of the subsequence from index l to r. Building a data structure to efficiently compute range modes allows data analysis to be conducted over any window of one-dimensional data. Such queries are common and important in database systems.

The range least frequent problem can be seen as a low-frequency variant of range mode. Instead of searching for the most frequent element in an interval of the sequence, we query for an element that occurs the fewest number of times. We may either restrict our attention to elements that occur at least once in the query range or allow the answer to be an element that does not occur in the interval but is present elsewhere in the sequence.

A third range frequency query we consider in this chapter is the problem of identifying an element with given frequency k in the specified query interval. This problem has been called the range k-frequency problem.

Both range mode query and range least frequent query have theoretical connections to matrix multiplication. In particular, the ability to answer n range mode queries on an array of size $O(n)$ in faster than $O(n^{\omega/2})$ time, where ω is the constant in the exponent of the running time of matrix multiplication, would imply a faster algorithm for boolean matrix multiplication [55, 56]. Upon closer examination, this lower bound also applies to the range k-frequency problem.

The range mode, range least frequent, and range k-frequency problems are part of a set of questions one can ask on a subrange of a sequence. Other queries in this area include:

- Sum: Return the sum of elements in the query range.
- Min/ Max: Return the minimum/ maximum in the query range.
- Median: Determine the median element in the query range.
- Majority: Return the element that occurs more than 1/2 the time, if such an element exists.

Note that the mean of a range can be reduced to the range sum problem. Table 4.1 gives an overview of known upper bounds on related static and dynamic range query data structures.

Query Type	Query Time	Update Time	Space	Citation
Sum	$O(1)$	-	$O(n)$	trivial
	$O(\lg n)$	$O(\lg n)$	$O(n)$	[200]
Min/ Max	$O(1)$	-	$O(n)$	[102]
	$O(\lg n/ \lg\lg n)$	$O(\lg^{1/2+\varepsilon} n)$	$O(n)$	[57]
Median	$O(\lg n/ \lg\lg n)$	-	$O(n)$	[44]
	$O((\lg n/ \lg\lg n)^2)$	$O((\lg n/ \lg\lg n)^2)$	$O(n)$	[131]
Majority	$O(1)$	-	$O(n)$	[83]
	$O(\lg n/ \lg\lg n)$	$O(\lg n)$	$O(n)$	[83, 104]
Mode	$O(n^\varepsilon \lg\lg n)$	-	$O(n^{2-2\varepsilon})$	[167]
	$O(1)$	-	$O(n^2 \lg\lg n/ \lg n)$	[167]
	$O(\sqrt{n/ \lg n})$	-	$O(n)$	[55]
	$O(n^{3/4} \lg n/ \lg\lg n)$	$O(n^{3/4} \lg\lg n)$	$O(n)$	[55]
	$O(n^{2/3} \lg n/ \lg\lg n)$	$O(n^{2/3} \lg n/ \lg\lg n)$	$O(n^{4/3})$	[55]
	$O(n^{2/3})$	$O(n^{2/3})$	$O(n)$	new
Least Frequent	$O(\sqrt{n})$	-	$O(n)$	[56]
	$O(n^{2/3} \lg n \lg\lg n)$	$O(n^{2/3})$	$O(n)$	new
k-Frequency	$O(n^{2/3})$	$O(n^{2/3} \lg n)$	$O(n)$	new

Table 4.1: Known static and dynamic range query upper bounds. Solutions are deterministic or Las Vegas randomized.

4.1.1 Our Results

We improve the results of Chan et al. [55] by giving a dynamic range mode data structure that takes $O(n)$ space and supports updates and queries in $O(n^{2/3})$ time. This improves the query/ update time of their linear space data structure by a polynomial factor and additionally improves the query/ update time of their $O(n^{4/3})$ space data structure by an $O(\log n/ \log\log n)$ factor. We also include in our update procedures the ability to insert or delete elements in the middle of the array, an operation not addressed in previous dynamic range mode data structures.

Our improvements are based on the observation that knowing how many of a type of element occur in an interval can be as valuable as knowing the elements themselves. Specifically, instead of storing the frequency counts of elements per span, we store the number of elements with a particular frequency count per span. This information can be dynamically maintained, and uses $O(\log n)$ bits per frequency per span, rather than $O(\log n)$ bits per unique element per span.

Our technique is general enough to also apply to the range least frequent problem in a dynamic setting. To our knowledge, this is the first data structure to do so. In the version of the problem where we allow an answer to not occur in the specified query interval, our data structure supports queries in $O(n^{2/3} \log n \log\log n)$ time, updates in $O(n^{2/3})$ time, and occupies $O(n)$ space. In the version where the least frequent element must be present in the query interval, we develop a Monte Carlo data structure that supports queries in $O(n^{2/3})$ time, updates in $O(n^{2/3} \log n)$ time, and occupies $O(n \log^2 n)$ space. This data structure is correct with high probability for any polynomial sequence of updates and queries, with the restriction that updates are made independently of the results of previous queries. Notably, if the set of queries and updates is fixed in advance or given to the algorithm all at once, this property holds.

Furthermore, our Monte Carlo data structure is powerful enough to apply to the dynamic range k-frequency problem, also supporting queries in $O(n^{2/3})$ time, updates in $O(n^{2/3} \log n)$ time, and occupying $O(n \log^2 n)$ space. This data structure can be augmented to count the number of elements below, above, or at a given frequency, supporting both queries and updates in $O(n^{2/3})$ time and using $O(n)$ space, without the need for an independence assumption.

We organize our results as follows. In Section 4.2, we review previous work on static and dynamic range mode and least frequent element queries. In Section 4.3, we briefly give some notation that will be used for the rest of the chapter. In Section 4.4, we give the basic setup of the $O(n)$ space data structure. Section 4.5 describes how to answer a range mode query in $O(n^{2/3})$ time. Section 4.6 explains how to support updates to our base data structures in $O(n^{2/3})$ time. In Section 4.7, we discuss how to answer range least frequent queries in $\tilde{O}(n^{2/3})$ time. Section 4.8 describes how to find an element of a given frequency in a specified range in $O(n^{2/3})$ time. Here we also mention additional frequency operations our data structure can support. In Section 4.9, we describe how the data structures can be made to support insertion and deletion of elements, also in $O(n^{2/3})$ time. Finally, in Section 4.10 we describe a modified dynamic array data structure used in our results, and in Section 4.11 we describe a polylogarithmic space Monte Carlo data structured used in our results.

4.2 Previous Work

4.2.1 Static Range Mode Query

The static range mode query problem was first studied by Krizanc et al. [167]. Their focus is primarily on subquadratic solutions with fast queries, achieving $O(n^{2-2\varepsilon})$ space and $O(n^{\varepsilon} \log n)$ query time, with $0 < \varepsilon \leq 1/2$, and $O(n^2 \log\log n / \log n)$ space and $O(1)$ query time. If we set $\varepsilon = 1/2$ with the first approach, this gives a linear space static range mode data structure with query time $O(\sqrt{n} \log n)$. By substituting an $O(\log\log n)$ data structure for predecessor search, such as van Emde Boas trees [230], the query time can immediately be improved to $O(\sqrt{n} \log\log n)$.

Chan et al. [55] focus on linear space solutions to static range mode. They achieve a clever array-based solution with $O(n)$ space and $O(\sqrt{n})$ query time. By using bit-packing tricks and more advanced data structures, they reduce the query time to $O(\sqrt{n/\log n})$.

As with many range query data structures, the range mode problem has also been studied in an approximate setting [123, 37].

4.2.2 Static Range Least Frequent Query

The range least frequent problem was first studied by Chan et al. [56]. They again focus on linear-space solutions, this time achieving $O(n)$ space and $O(\sqrt{n})$ time query. In their paper, they focus on the version of range least frequent element where the element must occur in the query range.

4.2.3 Dynamic Range Mode

Chan et al. [55] also study the dynamic range mode problem. They give a solution tradeoff that at linear space, achieves $O(n^{3/4} \log n / \log\log n)$ worst-case time range mode query and $O(n^{3/4} \log\log n)$ amortized expected time update. At minimal update/ query time, this tradeoff gives $O(n^{4/3})$ space and $O(n^{2/3} \log n / \log\log n)$ worst-case time range mode query and amortized expected time update.

4.2.4 Lower Bounds

Both [55] and [56] give conditional lower bounds for range mode and range least frequent problems, respectively. Chan et al. [55] reduces multiplication of two $\sqrt{n} \times \sqrt{n}$ boolean matrices to n range mode queries in an array of size $O(n)$. This indicates that with current knowledge, preprocessing an $O(n)$-sized range mode query data structure and answering n range mode queries cannot be done in better than $O(n^{\omega/2})$ time, where ω is the constant in the exponent of the running time of matrix multiplication. With $\omega = 2.3727$ [237], this implies with current knowledge that a range mode data structure must either have preprocessing time at least $\Omega(n^{1.18635})$ or query time at least $\Omega(n^{0.181635})$. Since we may choose to update an array rather than initializing it, this lower bound also indicates that a dynamic range mode data structure must have update/ query time at least $\Omega(n^{\omega/2-1})$.

In [55], Chan et al. also give another conditional lower bound for dynamic range mode. They reduce the multiphase problem of Pătraşcu [206] to dynamic range mode. A reduction from 3-SUM (given n integers, find three that sum to zero) is given by Pătraşcu [206] to the multiphase problem. Based on the conjecture that the 3-SUM problem cannot be solved in $O(n^{2-\varepsilon})$ time for any positive constant ε, this chain of reductions implies a dynamic range mode data structure must have polynomial time query or update.

The reduction of $\sqrt{n} \times \sqrt{n}$ boolean matrix multiplication to n $O(n)$-sized range mode queries can be adopted to achieve the same conditional lower bound for the range least frequent problem [56]. Upon examination, the conditional lower bound also applies to k-frequency queries.

An unconditional lower bound also exists in the cell probe model for the range mode and k-frequency problem. Any range mode/ k-frequency data structure that uses S memory cells of w-bit words needs $\Omega(\frac{\log n}{\log(Sw/n)})$ time to answer a query [123].

4.3 Preliminaries

Before we discuss the technical details of our results, it will be helpful to develop some notation.

As in [55, 56], we will denote the subarray of A from index i to index j as $A[i:j]$ and use array notation $A[i]$ to denote the element of A at index i. We will assume zero-based indexing throughout this chapter.

Furthermore, for range frequency data structures, the actual type that array A stores is irrelevant; we only care about how many times each element occurs. It will be useful to think of the identity of an element as a color. Therefore, we may say the color c occurs with frequency f in range $A[l : r]$. This is to distinguish that we are not referring to a particular index but rather the identity of multiple indices in the range.

For simplicity, we will assume the number of elements n is a perfect cube; however, the results discussed easily generalize for arbitrary n. All log's in this chapter are assumed to be base 2.

In Section 4.9 we will discuss operations that insert and delete elements into and from array A. This effectively changes the indexing of elements after the point of operation. It will be helpful to develop an indexing scheme that does not change with the effect of insertion or deletion. To differentiate between the interface of the operations and the internals used by the data structure, we will denote with every position of A two values: the *rank* and *index* of the corresponding position. This distinction only becomes important when elements are inserted into positions of A, and thus are only relevant to Section 4.9.

As will be explained in Section 4.9, to accommodate insertion and deletion in $O(n^{2/3})$ time, empty space must be made in a series of arrays that ultimately represent A. We will refer to the elements that reside in the extra space of each section as "empty" elements. We associate two properties with each position in A: the index, which we refer to as the absolute position when all array sections are arranged in order, including the "empty" elements; and the rank, which, when the position is occupied by a color, denotes the number of non-empty elements that precede the position in the total order.

By $A[i]$ we refer to the ith index of A, not rank; and similarly, we let $A[i : j]$ denote the subrange of A from index i to index j. The interface of all our operations to the user function on rank, but the internals of all our data structures function on index. As will be seen in Section 4.9, we split A into sections of $O(n^{1/3})$ elements. This allows conversion from rank to index and back in $O(n^{1/3})$ time, so the distinction is not important for the complexity of our operations.

In all our proofs, we analyze space cost in words, that is, $O(\log n)$ collections of bits.

4.4 Data Structure Setup

The idea of our data structure will be to break array A into $O(n^{1/3})$ evenly-spaced endpoints, so that there are $O(n^{2/3})$ elements between each endpoint. We will use capital letters L and R when referring to particular endpoints. We will also occasionally refer to the elements between two consecutive endpoints as a *segment*; therefore, there are $O(n^{1/3})$ segments in A.

Each color that occurs in A will be split into the following two disjoint categories:

- **Frequent Colors**: Any color that appears more than $n^{1/3}$ times in A.
- **Infrequent Colors**: Any color that appears at most $n^{1/3}$ times in A.

Note that there can be at most $n/n^{1/3} = n^{2/3}$ frequent colors in A at any point in time.

Our data structure will need to use dynamic arrays as auxiliary data structures. These dynamic arrays will be a modification of the simple two-level version of the data structure described by Goodrich and Koss [117]. We have the following lemma regarding the performance of these dynamic arrays:

Lemma 46. *There is a dynamic array data structure D that occupies $O(n)$ space and supports:*

1. *$D[i]$: Retrieve/ Set the element at rank i, in $O(1)$ time.*
2. *$Insert(i, x)$: Insert the element x at rank i, in $O(\sqrt{n})$ time.*
3. *$Delete(i, x)$: Delete the element x at rank i, in $O(\sqrt{n})$ time.*
4. *$Rank(ptr)$: Determine the rank of the element pointed to by ptr, in $O(1)$ time.*

We leave the proof of Lemma 46 and discussion of the dynamic array data structure to Section 4.10. If desired, the data structure can be replaced by a balanced binary search tree, at the cost of additional logarithmic factors in the query times of our data structure.

We can now describe the *base set* of auxiliary data structures used throughout the chapter:

1. Arrays $B_{L,R}$, for all pairs of endpoints L, R, indexed from 0 to $n^{1/3}$, so that

$$B_{L,R}[i] := \text{The number of infrequent colors with frequency } i \text{ in } A[L:R].$$

2. Dynamic arrays D_c, for every color c, so that

$$D_c[i] := \text{The index in } A \text{ of the } i\text{th occurrence of color } c.$$

3. An array E parallel to A so that

$$E[i] := \text{A pointer to the location in memory of index } i \text{ in dynamic array } D_{A[i]}.$$

4. A binary search tree F of endpoints. At each endpoint R, we store

$$F[R] := \text{A binary search tree on frequent colors, giving their frequency in } A[0:R].$$

In regards to $B_{L,R}$, we will sometimes refer to the set of elements between endpoints L and R as the span of L and R.

We now analyze the space complexity and construction time.

Lemma 47. *The base data structures take $O(n)$ space.*

Proof. The arrays $B_{L,R}$ have size $O(n^{1/3})$ and there are $O(n^{2/3})$ of them, so in total these take $O(n)$ space. Every index of A is present in exactly one of the D_c arrays, and each dynamic array takes linear space. Therefore, in total all D_c arrays take $O(n)$ space. Array E has exactly the same size as A and thus takes $O(n)$ space. The binary search trees in each node of F have size equal to the number of frequent colors, which is at most $n^{2/3}$. Since there are $O(n^{1/3})$ endpoints and thus nodes of F, this structure takes $O(n)$ space. □

Lemma 48. *The base data structures can be initialized in $O(n^{4/3})$ time.*

Proof. We can count the number of occurrences of each color in $O(n \log n)$ time and determine for each color whether it is frequent or infrequent. Let A' be the array A without any frequent colors. We can scan A' $n^{1/3}$ times, starting from each endpoint, to build the arrays $B_{L,R}$. This will be done as follows. For each scan, we maintain an array T so that $T[c]$ denotes the number of occurrences of color c found so far. We also maintain the array $B_{L,*}$ which is the array B with endpoint L and a variable right endpoint, that is maintained as elements are scanned. When $A[i] = c$, we check the number of occurrences of c to update $B_{L,*}$ to the correct state. In total, each element scanned results in $O(1)$ operations, until we reach a right endpoint. When we reach a right endpoint, we write $B_{L,*}$ to array $B_{L,R}$, where R denotes the right endpoint just encountered. In this way, for all endpoints L, R, we spend $O(n^{1/3})$ time to create the array. Thus the element scan dominates the time complexity, requiring $O(n^{4/3})$ time to create all $B_{L,R}$ arrays.

The dynamic arrays D_c can be built in linear time overall by walking through A and appending indices to the ends of D_c arrays. At this same time E can be built. The BSTs for all endpoints in F can also be built in linear time overall, since there are at most $n^{2/3}$ frequent colors per list, there are $n^{1/3}$ lists in total, and counting each frequent element in each interval can be done at the same time in one scan through A. □

Throughout the next few sections, we will use the following Lemma, as in [55]:

Lemma 49. *Let $A[i] = c$. Then, given frequency f and right endpoint j, our base data structures may answer the following questions in constant time:*

$$\text{Does color } c \text{ occur at least } f \text{ times in } A[i:j]? \tag{4.1}$$

$$\text{Does color } c \text{ occur at most } f \text{ times in } A[i:j]? \tag{4.2}$$

$$\text{Does color } c \text{ occur exactly } f \text{ times in } A[i:j]? \tag{4.3}$$

Proof. We call $rank(E[i])$ on D_c to get the rank r of i in D_c. Since array D_c stores the indices of every occurrence of color c in A, the values of $D_c[r+f]$ and $D_c[r+f-1]$ determine the answers to the above questions. □

A similar strategy can be used to answer the above questions given an index j and left endpoint i.

4.5 Range Mode Query

The range mode query will make use of the following lemma, originating from [167] and also used by [55]:

Lemma 50 (Krizanc et al. [167])**.** *Let A_1 and A_2 be any multisets. If c is a mode of $A_1 \cup A_2$ and $c \notin A_1$, then c is a mode of A_2.*

The query algorithm is described in Algorithm 2.

Theorem 51. *Algorithm 2 finds the current range mode of $A[l:r]$ in $O(n^{2/3})$ time.*

Proof. Endpoints L and R can be found from $[l,r]$ by appropriate floors and ceilings in constant time.

In step 1, we can iterate through $F[R]$ and $F[L]$ in $O(n^{2/3})$ time, determining frequency counts for all frequent elements.

In step 2, answering question (4.1) takes $O(1)$ time per element via Lemma 49. Note that we use left endpoint l for elements in range $A[R+1:r]$ and right endpoint r for elements in range $A[l:L-1]$. Although we do not check (4.1) for the full range $[l,r]$, we will check the first/ last occurrence of any color in $A[l:L-1] \cup A[R+1:r]$, thereby effectively checking for all of $[l,r]$. When (4.1) is answered in the affirmative at index i, we linearly scan D_c, starting at $D_c[i+f+1]$ ($D_c[i-f-1]$ if $i \in [R+1:r]$) to determine a new highest frequency. Let us determine the cost of these linear scans. Let c be the most frequent color found from steps 1 and 2. If c is an infrequent color, it cannot occur more than $n^{1/3}$ times, so the total cost of the linear scans is no more than $O(n^{1/3})$. If it

Algorithm 2 Range Mode Query in $A[l:r]$

Let L and R be the first and last endpoints in $[l,r]$, respectively.

1. Check the frequency of every frequent color in $A[L:R]$ via BSTs $F[R]$ and $F[L]$. Let f be the highest frequency found so far.
2. Ask question (4.1) for all colors in $A[l:L-1] \cup A[R+1:r]$ with frequency f and right endpoint r/ left endpoint l. If (4.1) is answered in the affirmative for color c, linearly scan D_c to count the number of occurrences of color c in $[l,r]$, update f, and continue.
3. Find the largest nonzero index of $B_{L,R}$. If this is larger than f, update f and do the following:
   (a) Find the next endpoint R' to the left of R.
   (b) Check $B_{L,R'}[f]$. If $B_{L,R'}[f] < B_{L,R}[f]$, search $A[R'+1:R]$ for a color that occurs f times in $A[L:R]$, via question (4.3) with left endpoint L.
   (c) Otherwise, repeat from step (a) with $R \leftarrow R'$.
4. Return f and the corresponding color found from either step 1, 2, or 3(b).

is a frequent color, its frequency cannot have increased by more than $O(n^{2/3})$, since its frequency was checked in step 1 and only $O(n^{2/3})$ elements exist in $A[l:L-1] \cup A[R+1:r]$. Thus the total cost of the linear scans is no more than $O(n^{2/3})$. In either case, step 2 takes $O(n^{2/3})$ time.

For step 3, finding the largest nonzero index of $B_{L,R}$ takes $O(n^{1/3})$ time. If this is larger than the frequencies found in steps 1 or 2, we execute steps 3(a) - 3(c). We can only repeat the steps at most $O(n^{1/3})$ times. The condition $B_{L,R'}[f] < B_{L,R}[f]$ will happen for one of these iterations, since eventually $R' = L$ in which case there are no elements accounted for in the interval. When $B_{L,R'}[f] < B_{L,R}[f]$, this implies a color with frequency f in range $[L,R]$ appears somewhere in $A[R'+1:R]$. There are only $O(n^{2/3})$ elements in $A[R'+1:R]$. Checking if color c occurs exactly f times in $A[L:R]$ can be done by asking question (4.3).

For the correctness of the value returned in step 4, note that the mode of $A[l:r]$ is either an element in $A[l:L-1] \cup A[R+1:r]$ or the mode of $A[L:R]$, by Lemma 50. The frequency of all colors in $A[l:L-1] \cup A[R+1:r]$ is checked. Further, the frequency of infrequent and frequent colors for interval $A[L:R]$ is also checked in steps 1 and 3, respectively. Therefore the color and frequency returned in step 4 must be the mode of $A[l : r]$. Putting it together, we see Algorithm 2 is correct and takes $O(n^{2/3})$ time. □

4.6 Update Operation

The update operation will require us to keep the base data structures up to date. Given update $A[i] \leftarrow c$, there are two similar procedures that must occur: adjusting the data structures for the removal of current color $A[i]$ and adjusting the data structures for the addition of color c at index i.

The following algorithm can be used for both procedures. We note that if either the color removed is the last occurrence of its type or the color added is a new color, the list D_c will have to also be constructed/ deleted and $B_{L,R}$ should be modified to only reflect colors present in A. For simplicity we omit these details from Algorithm 3.

Algorithm 3 Update Base Data Structures for Addition/ Removal of Color c at Index i.

1. If color c is infrequent prior to this operation, count how many times c occurs in each span via D_c, decrementing the corresponding index of each B array.

2. Adjust D_c by adding/ removing i. If this is an add operation, set $E[i]$ to the memory location of i in D_c.

3. If color c remains or becomes infrequent after step 2, again count how many times c occurs in each span via D_c, incrementing the corresponding index of each B array.

4. If color c became infrequent in step 2, delete its entry in all nodes of F. If color c became frequent after step 2, add its frequencies to F. If color c remained frequent after step 2, increment the frequencies of all prefixes including index i in F.

Theorem 52. *Algorithm 3 updates the base data structures for addition/ removal of color c at index i in $O(n^{2/3})$ time.*

Proof. If c is an infrequent color prior to the update, step 1 removes its contribution to all B arrays. Counting the frequency of color c in each interval can be done via $O(n^{1/3})$ searches through D_c, each taking $O(n^{1/3})$ time. We first start at the beginning, finding its frequency for all right endpoints with left endpoint fixed, then move the left endpoint to the next endpoint and repeat, etc. Step 1 in total takes $O(n^{2/3})$ time.

Adding or removing i from D_c takes $O(\sqrt{n})$ time, as given in Lemma 46. Adjusting $E[i]$ can be done during this operation, so in total step 2 takes $O(\sqrt{n})$ time. The analysis of step 3 is identical to step 1. In step 4, adding, deleting, or modifying the entry of color c in all/ some nodes of F takes $O(n^{1/3} \log n)$ time, since each node stores the list of frequent colors and their frequency counts in a BST. If color c just became frequent, it only occurs $O(n^{1/3})$ times in A, and thus counting its frequency from the beginning to each endpoint can be done in $O(n^{1/3})$ time. In total, step 4 takes at most $O(n^{1/3} \log n)$ time.

After the completion of Algorithm 3, B arrays, dynamic array D_c, array E, and binary search tree F has been updated to reflect the current state of array A. Since all steps execute in no more than $O(n^{2/3})$ time, Algorithm 3 is correct and runs in $O(n^{2/3})$ time. □

4.7 Range Least Frequent Query, Allowing Zero

To answer range least frequent queries, we require the use of one more auxiliary data structure: an array $C_{L,R}$ for all pairs of endpoints L, R, indexed from 1 to $n^{1/3}$. At index $C_{L,R}[i]$ we store a list of all colors that occur i times in A such that the smallest span enclosing all occurrences is $[L, R]$.

Since each infrequent color is represented exactly once in the $C_{L,R}$ lists and there are $O(n^{2/3} \cdot n^{1/3}) = O(n)$ total indices present, this data structure takes linear space. It can also be initialized in linear time, and we can modify the update procedure of Algorithm 3 to update $C_{L,R}$ at the same time as $B_{L,R}$ for no additional time cost.

It is additionally worth noting that since the Range Least Frequent Query will require $\tilde{O}(n^{2/3})$ time, the use of dynamic arrays for data structures D_c is not necessary for this section. Instead, we can use binary search trees, augmented to support lookup by index in $O(\log n)$ time, or Dietz' data structure [71]. For the best time complexity, we will use augmented binary search trees to count occurrences of colors in a specific range and an augmented dynamic linear-space van Emde Boas tree [178] to count occurrences of colors between endpoints. The van Emde Boas tree stores a single node for any endpoint R that a color appears in, which keeps the number of occurrences of that color from the beginning to endpoint R. The van Emde Boas tree can be updated in $O(n^{1/3} \log\log n)$ time upon insertion or removal and via predecessor/ successor queries, can support counting the number of occurrences of any color between any two endpoints in $O(\log\log n)$ time.

The Range Least Frequent Query procedure is described in Algorithm 4.

Theorem 53. *Algorithm 4 finds the least frequent element of $A[l : r]$, allowing zero, in $O(n^{2/3} \log n \log\log n)$ time.*

Proof. We can find a list of frequent colors in any node of the F BST. Using an augmented

Algorithm 4 Range Least Frequent Query in $A[l:r]$, Allowing Zero

Let L and R be the first and last endpoints in $[l, r]$, respectively.

1. Check the frequency of every frequent color in $A[l:r]$ via D_c. Let f be the lowest frequency found so far.

2. Find the set of all infrequent colors that occur in $A[l:L-1] \cup A[R+1:r]$; call it U. Count the frequencies of all colors of U in range $A[l:r]$ and update f with the lowest frequency found so far.

3. Compute $B'_{L,R}$, the array $B_{L,R}$ updated to erase the contribution of all colors in U.

4. If the smallest positive-valued index of $B'_{L,R}$ is less than f, update f and check if any list $C_{L',R'}[f]$, $[L', R'] \subseteq [L, R]$, is non-empty. If so, return a color from the appropriate list. Otherwise, binary search from R to the last endpoint of A in the following way:

   (a) Let R' be a value in the middle of the search range. If $B'_{L,R'}[f] < B'_{L,R}[f]$, let R' be the new upper bound; otherwise, continue the search in the half of the range after R'.

   (b) When the search range is two consecutive endpoints, we search the range for a color that occurs f times in $A[L:R]$ and return its identity.

   (c) If the condition in a. is never satisfied, we must repeat a binary search on the other side, with an initial search range of the beginning of A to L.

---

binary search tree for D_c, step 1 can then be done in $O(n^{2/3} \log n)$ time. In step 2, since there are $O(n^{2/3})$ elements in $A[l:L-1] \cup A[R+1:r]$, we can find color set U and complete step 2 similarly to step 1 in $O(n^{2/3} \log n)$ time. Step 3 requires counting the frequency of each color of U in range $A[L:R]$ and decrementing the corresponding index to make $B'_{L,R}$; thus it can also be done in $O(n^{2/3} \log \log n)$ time using the augmented van Emde Boas tree [178].

In step 4 we are looking for a least frequent element of $A[L:R]$ that does not occur in $A[l:L-1] \cup A[R+1:r]$. All colors of $A[l:L-1] \cup A[R+1:r]$ are represented in set U, found in step 3. We effectively erase the contribution of colors of U via computing $B'_{L,R}$; therefore, we can proceed as if colors of U do not exist in A. We will refer to A' as array A without any colors of U.

If the least frequent color in $A'[L:R]$ does not exist elsewhere in A, then its identity will be stored in a list $C_{L',R'}[f]$, $[L',R'] \subseteq [L,R]$. Note colors of U need not be special-cased for this lookup, since by appearing in $A[l:L-1] \cup A[R+1:r]$, they will not be present in any of the searched lists. Otherwise, we know the least frequent color in $A'[L:R]$ must occur somewhere else in A'.

Now, amongst all colors in A', we know frequency f is minimal in range $A'[L:R]$. Therefore if we increase the range to $A'[L:R']$, the frequency of colors can only increase. For this property to hold, we must allow $f = 0$.

In each iteration, the smallest positive-valued index of $B'_{L,R'}$ will be f or greater and $B'_{L,R'}[f]$ will be no more than $B'_{L,R}[f]$. If it is less, we know one of the colors that occurred f times in $A'[L:R]$ now occurs more than f times in $A'[L:R']$. Therefore we may find it in the half of the search range before R'. If it is the same, we know none of the colors that occurred f times in $A'[L:R]$ appear in the half of the search range before R'. Either way we decrease the search range by a factor of 2. When the range represents two consecutive endpoints, we can search it for a color that occurs f times in $A[l:r]$ in $O(n^{2/3}\log n)$ time.

However, if R' is the end of the array and $B'_{L,R'}[f] = B'_{L,R}[f]$, then none of the colors that appear f times in $A'[L:R]$ appear to the right of R in A'. In this case, we can repeat the same binary search on the other side, decreasing a left endpoint L' and checking the same condition. Since the least frequent color in $A'[L:R]$ must occur elsewhere in A, as it was not present in any of the lists $C_{L',R'}[f]$, the search on this side must identify a color that appears f times in $A'[L:R]$.

The time complexity of step 4 can be analyzed as follows. Checking the lists $C_{L',R'}$ takes $O(n^{2/3})$ time, since there can be $O(n^{2/3})$ endpoints $[L',R'] \subseteq [L,R]$. The binary search is on endpoints, of which there are $O(n^{1/3})$. Thus, there are $O(\log(n^{1/3})) = O(\log n)$ iterations of the binary search, and in each iteration we must compute $B'_{L,R'}$ or $B'_{L',R}$. This computation takes $O(n^{2/3}\log\log n)$ time as in step 3. Therefore the binary search process takes $O(n^{2/3}\log n\log\log n)$ time. In total, step 4 takes $O(n^{2/3}\log n\log\log n)$ time.

For correctness, the least frequent element in $A[l:r]$ is either a frequent or infrequent color. If it is a frequent color, it is identified in step 1. If it is an infrequent color, we have two cases. Either the color occurs in $A[l:L-1] \cup A[R+1:r]$, and thus set U, or it does not occur in set U. Step 2 accounts for all infrequent colors in set U. Steps 3 and 4 account for the last case. By the above, these steps find an infrequent color in $A[L:R]$ that does not occur in $A[l:L-1] \cup A[R+1:r]$ in $O(n^{2/3}\log n\log\log n)$ time. Thus, Algorithm 4 is correct and finds the least frequent element of $A[l:r]$, allowing zero, in $O(n^{2/3}\log n\log\log n)$ time. □

4.8 Range k-Frequency Query

The previous two sections make use of a monotonicity property to find a color of given frequency in a range: for range mode, we know the frequency of the most frequent element can only decrease if the query range is decreased; furthermore, for range least frequent, we know the frequency of the least frequent element can only increase if the query range is increased. For this monotonicity condition to hold for least frequent elements, we must allow answering with an element of frequency zero. To force our answer to be an element that occurs in the query range, we must use an additional data structure that allows retrieval of colors by frequency in the $B_{L,R}$ arrays. To achieve $\tilde{O}(n)$ space, we cannot afford to store a list of colors at each index $B_{L,R}[i]$, and storing a single color at each index will run into issues during updates. Instead, we will use the following data structure which is similar to randomized data structures in the dynamic streaming literature [96]:

Lemma 54. *There is a Monte Carlo data structure that occupies expected $O(\log^2 n)$ space and supports:*

1. *$Insert(x)$: Insert element x into the collection, in $O(\log n)$ expected time.*

2. *$Delete(x)$: Delete element x from the collection, in $O(\log n)$ expected time.*

3. *$Retrieve()$: Return an element in the collection, in $O(1)$ expected time.*

The data structure requires the $Delete(x)$ operation is executed independently of the results of $Retrieve()$.

We leave the proof of Lemma 54 and discussion of the Monte Carlo data structure to Section 4.11. With it, we can answer the general problem of finding an element of given frequency in a query range. The additional auxilliary data structures needed will be as follows:

1. An array $G_{L,R}$ parallel to $B_{L,R}$. At index $G_{L,R}[i]$, we store the number of infrequent colors with frequency i in $A[L : R]$, excluding colors that appear in segments immediately left of L or right of R.

2. An array $H_{L,R}$, parallel to $G_{L,R}$, so that at index $H_{L,R}[i]$, we store a collection of colors counted in $G_{L,R}[i]$ in the data structure of Lemma 54.

The array $G_{L,R}$ is similar to the item (ii) stored in table D of [56]. During preprocessing, G arrays can be built similarly to B; however, when we fix left endpoint, we check all colors that occur in the segment to the left. We avoid counting such colors. Similarly, before we finalize the count for $G_{L,R}$, we move the right endpoint out as if to count the next range, keeping track of colors encountered. We subtract the frequency of such colors in $G_{L,R}$. Whenever we add to/ subtract from $G_{L,R}$, we can insert or delete the color from $H_{L,R}$.

As explained above, our space cost now becomes $\tilde{O}(n)$. Furthermore, our updates to G and H can be done alongside the update to B; however, since each insertion takes $O(\log n)$ time, the update procedure now takes $O(n^{2/3} \log n)$ time. With this we have:

Algorithm 5 Range k-Frequency Query in $A[l : r]$

Let L and R be the first and last endpoints in $[l, r]$, respectively.

1. For color c at index i in the segment left of L, check via D_c to see if i is the first index of color c to appear in range $[l, r]$, or, if outside $[l, r]$, the next occurrence of color c lies in range $[l, r]$. If so, ask question (4.3) for the occurrence of color c in $[l, r]$ with frequency k and right endpoint r. If answered in the affirmative, return color c. Do the same, symmetrically, for colors in the segment right of R.

2. For each frequent color not addressed in step 1, check its frequency in $A[L : R]$ via BSTs $F[R]$ and $F[L]$. If any occur with frequency k, return the color.

3. If no color is found from step 1, check if $G_{L,R}[k] > 0$. If so, return $H_{L,R}[k].Retrieve()$. If not, return that no color has frequency k in range $A[l : r]$.

Theorem 55. *Algorithm 5 returns an element of frequency k in $A[l : r]$ or indicates no such element exists, assuming update operations have been executed independently of results of query operations, in $O(n^{2/3})$ time.*

Proof. In step 1, we look at two full segments of $O(n^{2/3})$ total elements. For color c at index i, if $i \in [l, r]$, we must determine if index i is the first occurrence of color c in $[l, r]$. Let $m = D_c.rank(E[i])$. Index i is the first occurrence of color c in $[l, r]$ if $D_c[m - 1]$ is outside $[l, r]$. Similarly, if $i \notin [l, r]$, we can again define $m = D_c.rank(E[i])$, then check if $D_c[m + 1]$ is in $[l, r]$. In any case, for any color that occurs in segments immediately left of L or right of R, one of the indices will be the first outside $[l, r]$ or the first within $[l, r]$. Thus the frequency of the color will be checked in $A[l : r]$. Since we do a constant number of constant time operations for $O(n^{2/3})$ elements, step 1 takes $O(n^{2/3})$ time.

As in Algorithm 2, step 2 takes $O(n^{2/3})$ time. Step 3 takes $O(1)$ time. In any case, in step 1 we check all elements in segments immediately left of L or right of R to see if they occur k times in $A[l : r]$. Furthermore, in steps 2 and 3, we check every frequent and infrequent color that occurs in $A[L:R]$ but not in segments immediately left of L or right of R, via array G. Thus we have checked every color if it occurs k times in $A[l:r]$. Since no step takes more than $O(n^{2/3})$ time, this proves Theorem 55. □

Algorithm 5 can be easily modified to return the least frequent element present in the query range with the same time complexity and independence assumption. It can also be modified to count the number of elements above, below, or at a given frequency, as well as only determine the frequency of the least frequent element. Since these queries do not ask for a color, arrays $H_{L,R}$ are not needed. This reduces the space cost to $O(n)$, update cost to $O(n^{2/3})$, and requires no independence assumption.

4.9 Insert/ Delete Operations

To allow for efficient insertion and deletion into A, instead of storing array A directly, we will split A into the $O(n^{1/3})$ sections between endpoints and maintain each section as its own array. When created, each section will be twice as large as necessary to provide space for insertion of elements. As stated in Section 4.3, we will refer to the elements that reside in the extra space of each section as "empty" elements.

Recall from Section 4.3 that we will associate two properties with each position in A: the index, which we refer to as the absolute position when all array sections are arranged in order, including the "empty" elements; and the rank, which, when the position is occupied by a color, denotes the number of non-empty elements that precede the position in the total order.

Then $A[i]$ refers to the ith index of A, not rank; and similarly, $A[i:j]$ denotes the subrange of A from index i to index j. As before, $A[L:R]$ denotes the subrange of A from endpoint L to endpoint R. Though the interface of all our operations in the preceding sections function on rank, the internals of all our data structures function on index. By storing the number of non-empty elements in each array section of A, we may convert from rank to index and back in $O(n^{1/3})$ time, so this distinction does not affect the running times of the previous sections.

Although the number of nonempty elements of A will change with each insert/ delete operation, we will assume the variable n, when used to distinguish between frequent and infrequent colors, is fixed, so that designations need not change during these operations. When the number of nonempty elements of A changes to require rebuilds, we may assume n changes during this rebuild. In this way, the value n and the number of nonempty elements of A will not differ by more than a constant factor.

We will assume the version of the proposed data structure that performs updates in $O(n^{2/3})$ time. In Section 4.8, some of the data structures discussed require $\tilde{O}(n^{2/3})$ time per update. In this case, the insert/ delete operations take the time complexity of the update operation, that is, $\tilde{O}(n^{2/3})$ time.

Lemma 56. *If an insertion/ deletion does not require resizing of an array segment, we may perform this operation in $O(n^{2/3})$ time.*

Proof. When we insert an element at rank i, we determine which array it falls in by checking the sizes of each array in $O(n^{1/3})$ time. We then find the appropriate index at which it should reside in the corresponding array and push elements back in the array in $O(n^{2/3})$ time. This requires updating the corresponding entries in D_c for the elements pushed back. The pointers stored in E allow these updates in $O(1)$ time per element moved.

At this point, index i is an open space to which we may set $A[i] = c$. We then must call Algorithm 3 to update the base data structures to the addition.

Deleting an element at rank i can be done similarly; pushing elements forward in the corresponding array segment, updating D_c for each color c that is moved, and calling Algorithm 3 to update the base data structures for the removal. □

When an array segment has no more space for element insertions, or has less than a quarter of its elements non-empty, we need to perform a more sophisticated operation to preserve functionality and the $O(n)$ space bound. The simple solution is to just rebuild the whole data structure. Since we can do this in $O(n^{4/3})$ time, by Lemma 48, and rebuilding will only be required after $O(n^{2/3})$ insertions/ updates, we can use this approach to get amortized $O(n^{4/3}/n^{2/3}) = O(n^{2/3})$ time per insertion and deletion.

It is possible to make this bound worst-case; however, the usual blackbox trick of rebuilding as we approach the threshold for needing a new data structure does not work. One reason for this is that we need to start rebuilding for each possible segment that can over/ under flow. The rebuilt structure has size $O(n)$, so maintaining this for each segment would take $O(n^{4/3})$ space. Instead, we can try to rebuild just what we need to either break a segment into two segments, or to absorb the segment into a surrounding segment. When the number of endpoints has doubled or halved, we can rebuild the entire data structure. In this case, since we need only store a single new copy, the blackbox technique applies.

Theorem 57. *We may perform insert and delete in $O(n^{2/3})$ worst-case time.*

Proof. Insertion/ deletion of elements without addressing resizing of a segment is handled in Lemma 56. When a segment either reaches full capacity or $1/4$ capacity, we need to either merge the segment with a surrounding segment or split the segment into two segments. If merging the segments would result in a segment that would be above full capacity, we can instead move elements from the larger segment to the smaller segment. In any of the three scenarios, we need to achieve $O(n^{4/3})$ worst-case time and $O(n^{2/3})$ work space to be able to apply blackbox dynamization techniques to make the overall bound worst-case.

Any process of physically moving elements from one segment to another can be done similarly to as in Lemma 56. This takes $O(n^{2/3})$ time per move and updates B arrays, D_c, E, and F via Algorithm 3. Therefore, physically moving elements in any of the above three scenarios will require $O(n^{4/3})$ time and no more than $O(n^{2/3})$ work space. We now need consider the costs of adding and deleting endpoints.

When a segment is absorbed, the endpoint between the smaller segment and its neighbor will need to be removed. Doing so takes $O(\log n)$ time to modify F, and no more than $O(n^{1/3})$ time to delete any B arrays that use this endpoint.

Similarly, when a segment is split, a new endpoint must be created. We can do this as follows. We need to populate the B array for this new endpoint. There are only $O(n^{1/3})$ other end points and each B array has $O(n^{1/3})$ values, so this takes $O(n^{2/3})$ work space for the structure we wish to keep. However, to create the new B array, we must count each color in the corresponding ranges. If we count each color individually, via D_c, we can count all colors in the corresponding ranges efficiently. Since each infrequent color occurs at most $n^{1/3}$ times, and there are at most n infrequent colors, this takes $O(n^{4/3})$ time and $O(n^{2/3})$ work space.

Updating F requires adding a new endpoint and counting frequent elements from the beginning of A to this endpoint. We can use the counts from the preceding endpoint and add the occurrences of elements from the beginning of the segment to the position of the new endpoint. This will then take $O(n^{2/3})$ time and work space.

In Section 4.7, we describe a data structure that can ultimately be updated alongside B, but this structure may take more than $O(n^{2/3})$ space for some segments. This is okay because when the space cost of each segment is totaled, the structure takes $O(n)$ space overall. The additional data structures in Section 4.8 can be updated alongside $B_{L,R}$.

Thus it is possible to create the new data structures we need in $O(n^{4/3})$ time and $O(n^{2/3})$ work space. To achieve worst-case insert/ delete operations, as we approach the threshold for needing to resize a segment, we rebuild the necessary structures at a rate of $O(n^{2/3})$ operations per insert/ delete in the corresponding segment. These partially-constructed data structures can be kept up to date alongside the base data structures when updates occur. If a threshold is reached in a segment, we swap the necessary structures to the global structure. One further point must also be addressed. If we create a new endpoint, then in the next iteration perform an update that creates another new endpoint, the new B array for these two endpoints will not have been created. To fix this, we can build B arrays to prospective endpoints as well as current endpoints during the building process. As each segment has only one prospective endpoint, this doesn't impact costs asymptotically.

Eventually, when the number of endpoints doubles or halves, we need to rebuild the whole structure to maintain that we have $O(n^{1/3})$ endpoints and $O(n^{2/3})$ elements between each endpoint. Since only one copy of the whole structure need be rebuilt, with the trick of rebuilding/ updating at twice the rate, this can be done in $O(n)$ extra space, and $O(n^{4/3}/n^{2/3}) = O(n^{2/3})$ extra time per operation. □

4.10 Modified Dynamic Array Data Structure

For the purposes of this chapter, we will only need the two-level structure of [117]. It should be possible to generalize the operations discussed for the two-level structure to the multi-level structure, if desired.

Suppose we wish to represent a dynamic array on n elements. We define a parameter $n_f = \Theta(n)$; the value of n_f changes as n becomes too large or too small. Moreover, we define m to be $\lceil\sqrt{n_f}\rceil$. We will maintain $\lceil n/m \rceil$ circular queues Q_i of exactly m elements each, except possibly for the last queue. Every element in each queue has a place in a total order represented by the queues. We say all elements of Q_i precede elements of Q_{i+1}, and within a queue Q_j, distance from the head of the queue determines the relative order of the elements. Each circular queue Q_i supports the following operations:

1. Insert element x at index j of Q_i, then pop the last element off the queue and return it.

2. Delete the element at index j of Q_i, sliding elements after j forward in the queue.

3. Push element x onto the front/back of the queue, then pop the first/last element off the queue and return it.

4. Return the element at index j of Q_i.

5. Determine the rank of object x in Q_i.

Standard array-based implementations of circular queues support operations 1 and 2 in $O(m)$ time and operations 3 and 4 in $O(1)$ time. Operation 5 can be implemented in $O(1)$ time as follows. Underlying the circular queue is an array. For each object x, we store its position in that array. The rank of object x is just the position of object x in the array, minus the position of the head of the queue in the array, modulo m.

We can now prove Lemma 46 from Section 4.4.

Proof of Lemma 46. Since each circular queue has exactly m elements, retrieving/ setting the element at rank i can be done via retrieving/ setting $Q_{i/m}[i \mod m]$. The arithmetic takes constant time, as does retrieving/ setting $Q_{i/m}[i \mod m]$, so this operation takes $O(1)$ time.

Insertion at rank i requires inserting into $Q_{i/m}$ at index $(i \mod m)$. This pops the last element off $Q_{i/m}$, which will then be pushed onto the following queue, and so on. The insertion into $Q_{i/m}$ takes $O(m) = O(\sqrt{n_f}) = O(\sqrt{n})$ time. The cascading insert-at-head, pop-from-tail operations take $O(1)$ time each and must be executed no more than n/m times. Thus insertion takes $O(n/m) = O(n/\sqrt{n_f}) = O(\sqrt{n})$ time.

Deletion at rank i is similar to insertion. We delete index $(i \mod m)$ from $Q_{i/m}$. We then remove the head of $Q_{n/m}$, and push it to the previous queue, and so on, until the vacant position at the end of queue $Q_{i/m}$ is filled. Deletion from $Q_{i/m}$ takes $O(m)$ time, and there are no more than n/m cascading operations that each take $O(1)$ time. In total, deletion takes $O(m + n/m) = O(\sqrt{n})$ time.

The rank of a given pointer *ptr* can be determined by the rank function in the queue in which it resides. Let r be its rank. We can then return $r + im$, where i is the index of the queue in which the object *ptr* points to resides.

The data structure must be resized when n becomes too large or too small with respect to n_f. We can build the above data structure in $O(n)$ time and therefore we can afford to rebuild from scratch when either condition occurs to achieve $O(\sqrt{n})$ amortized time insertion and deletion. By starting the rebuild at a fast pace when we approach the threshold on n_f, the amortized bound can be made worst-case. □

4.11 A $\operatorname{polylog} n$ Space Monte Carlo Data Structure

The idea of the data structure will be to sample at densities $1/2^i$ for i up to about $\log n$. No matter the size of the data structure, there is a constant probability that the top density has only one element in it. We repeat the data structure with $c\lceil \log n \rceil$ independent copies to achieve a high probability that at any point in time, we may sample one element from the data structure.

To make the $Delete(x)$ operation functional, we need to determine at which densities element x resides, as well as have the guarantee that x currently exists in the collection. We must use the same random bits we used upon insert so we remove x only from the correct densities. We can use the same model as universal families of hash functions to make the insert and delete functions dependent on the element, so behavior is deterministic, but analysis of expected results can be made over the choice of hash functions.

Note that independence of results of $Retrieve()$ and the $Delete(x)$ function is necessary; otherwise, we may repeatedly delete the element returned by $Retrieve()$. If this occurs, the elements are not sampled according to the desired exponential distribution, since elements at top densities have been specifically removed.

Our analysis will be similar to the cutset data structure of [148]. We can now prove Lemma 54.

Proof of Lemma 54. When we insert x into the data structure, we choose a number l so that $l = 1$ with probability $1/2$, $l = 2$ with probability $1/4$, and so on, so that $l = i$ with probability $1/2^i$. We refer to l as the *level* of element x. As explained above, we make the choice of l deterministic to element x so that we can determine the level of x if we delete x. With a universal family of hash functions, this can be accomplished by taking the hash of x in binary and using this as the random bit sequence. In this sense, our analysis beyond this point will reason with the expected behavior over the choice of hash functions.

Let l_{max} denote the maximum level currently represented in the data structure. We will maintain a mask for every level $1, \ldots, l_{max}$, as well as a counter at each mask. When we insert x into the data structure, we XOR its value with the mask at its level and increase the counter. If this causes l_{max} to increase, we add the necessary masks and counters to accomplish the task. When we execute $Delete(x)$, we find the level of x and again XOR its value with all masks at or below its level, this time decreasing the appropriate counters and possibly decreasing l_{max}. The $Retrieve()$ function works as follows. We check level l_{max} to see if the counter is at 1. If so, we return the element represented in the mask at level l_{max}. If not, we report `failure`.

Let n be an upper bound on the number of elements the collection will need to represent at any point in time. To achieve a high probability result, we will maintain $c\lceil \log n \rceil$ copies of the above data structure. Insert and delete will be done on each copy independently. The $Retrieve()$ function will only need to be executed until a single copy does not report `failure`. We analyze the probability of a single copy reporting failure as follows.

Let k be the number of elements currently represented in the collection. We can ignore all elements that were previously inserted and then deleted because, assuming independence of $Delete(x)$ and $Retrieve()$, this will not affect the resulting distribution. We can determine the chance of failure on $Retrieve()$ by analyzing a single probability: the probability the counter at level $\lfloor \log k \rfloor + 1$ is equal to 1.

This is the probability that exactly one out of the k elements has level $\lfloor \log k \rfloor + 1$. Since the probability a particular element has level $\lfloor \log k \rfloor + 1$ is $1/2^{\lfloor \log k \rfloor + 1}$, the probability exactly one out of the k elements has level $\lfloor \log k \rfloor + 1$ is

$$\binom{k}{1}\left(\frac{1}{2^{\lfloor \log k \rfloor + 1}}\right)\left(1 - \frac{1}{2^{\lfloor \log k \rfloor + 1}}\right)^{k-1} > \frac{1}{2}\left(1 - \frac{1}{k}\right)^{k-1}.$$

The right hand side approaches $1/2e$ and is above that value for all $k \geq 1$. Therefore the probability the counter at level $\lfloor \log k \rfloor + 1$ is equal to 1 is at least $1/2e$ for all values of k.

We can now reason about the success probability of *Retrieve*(). We can condition on the value of l_{max}. Clearly, $l_{max} \geq \lfloor \log k \rfloor + 1$ with probability at least $1/2e$, since the event the counter at level $\lfloor \log k \rfloor + 1$ is equal to 1 is included in this probability. Further, if $l_{max} > \lfloor \log k \rfloor + 1$, the probability the counter at l_{max} is equal to 1 must be at least $1/2e$, since it is less likely for an element to have a higher level. Therefore the conditional probability the counter at l_{max} is equal to 1 given that l_{max} is greater than $\lfloor \log k \rfloor + 1$ is at least $1/2e$. We can then conclude that with probability at least $1/4e^2$, the counter at l_{max} is equal to 1, though with more calculation we could certainly achieve a better constant.

By maintaining $c\lceil \log n \rceil$ copies of the data structure, the probability of failure in all of them is then

$$\left(1 - \frac{1}{4e^2}\right)^{c\lceil \log n \rceil} \leq n^{c \log(1 - \frac{1}{4e^2})} \leq 21 n^{-c}.$$

Therefore, for the right choice of c and by union bound, we can ensure a polynomial-length sequence of $Insert(x)$, *Delete*(x), and *Retrieve*() has arbitrarily low probability of failure.

To analyze space complexity, observe that the space complexity is the sum of the l_{max} variable for each copy of the data structure. Since each copy is independent and by linearity of expectation, the expected space complexity is $c\lceil \log n \rceil \mathbf{E}(l_{max})$. The value l_{max} is the maximum of exponential variables and it can be seen that $\mathbf{E}(l_{max})$ is $O(\log n)$. Therefore the space complexity is $O(\log^2 n)$.

The time complexity of $Insert(x)$ and *Delete*(x) is $O(1)$ + the number of additional masks/ counters inserted/ deleted. The expected number of masks/ counters inserted is no more than

$$\sum_{i=1}^{\infty} i/2^i = 2$$

therefore $Insert(x)$ and *Delete*(x) function in constant time per copy, for $O(\log n)$ time overall.

Since the *Retrieve*() function works with constant probability on each structure, in expectation we need only a constant number of attempts to have success. Therefore *Retrieve*() takes $O(1)$ time. This proves Lemma 54. □

Chapter 5

Faster Dynamic Range Mode

5.1 Introduction

Given a sequence of elements $a_1, a_2, \ldots, a_n$, the dynamic range mode problem asks to support queries for the most frequent element in a specified subsequence $a_l, a_{l+1}, \ldots, a_r$ while also supporting insertion or deletion of an element at a given index i. The mode of a sequence of elements is one of the most basic data statistics, along with the median and the mean. It is frequently computed in data mining, information retrieval, and data analytics.

The range mode problem seeks to answer multiple queries on distinct intervals of the data sequence without having to recompute each answer from scratch. Its study in the data structure community has shown that the mode is a much more challenging data statistic to maintain than other natural range queries: while range sum, min or max, median, and majority all support linear space dynamic data structures with poly-logarithmic or better time per operation [200, 57, 131, 104, 83], the current fastest dynamic range mode data structure prior to this work requires a stubborn $\Theta(n^{2/3})$ time per operation [80]. Indeed, range mode is one of few remaining classical range queries to which our currently known algorithms may be far from optimal. As originally stated by Brodal et al. [44] and mentioned by Chan et al. [55] in 2011 and 2014, respectively, "The problem of finding the most frequent element within a given array range is still rather open."

The current best conditional lower bound, by Chan et al. [55], reduces multiplication of two $\sqrt{n} \times \sqrt{n}$ boolean matrices to n range mode queries on a fixed array of size $O(n)$. This indicates that if the current algorithm for boolean matrix multiplication is optimal, then answering n range mode queries on an array of size $O(n)$ cannot be performed in time $O(n^{3/2-\epsilon})$ for $\epsilon > 0$ with combinatorial techniques, or $O(n^{\omega/2-\epsilon})$ time for $\epsilon > 0$ in general, where $\omega < 2.373$ [231, 106] is the square matrix multiplication exponent. This reduction can be strengthened for dynamic range mode by reducing from the online matrix-vector multiplication problem [132]. Using $O(n)$ dynamic range mode operations on a sequence of length $O(n)$, we can multiply a $\sqrt{n} \times \sqrt{n}$ boolean matrix with $\sqrt{n}$ boolean vectors given one at a time. This indicates that a dynamic range mode data structure taking $O(n^{1/2-\epsilon})$ time per operation for $\epsilon > 0$ is not possible with current knowledge.

Previous attempts indicate the higher $\Theta(n^{2/3})$ per operation cost as the bound to beat [55, 80]. Indeed, $\tilde{O}(n^{2/3})$ time per operation[1] can be achieved with a variety of techniques, but crossing the $\Theta(n^{2/3})$ barrier appears much harder.

Progress towards this goal has been established with the recent work of Williams and Xu [238]. They show that by appealing to Min-Plus product of structured matrices, n range mode queries on an array of size n can be answered in $\tilde{O}(n^{1.4854})$ time, thus beating the combinatorial lower bound for batch range mode. This result also shows a separation between batch range mode and dynamic range mode: while batch range mode can be completed in $O(n^{1/2-\epsilon})$ time per operation, such a result for dynamic range mode would imply a breakthrough in the online matrix-vector multiplication problem.

Range mode is not the first problem shown to be closely related to the Min-Plus product problem. It is well-known that the all-pairs shortest paths (APSP) problem is asymptotically equivalent to Min-Plus product [93], in the sense that a $T(n)$ time algorithm to compute the Min-Plus product of two $n \times n$ matrices implies an $O(T(n))$ time algorithm for APSP in n-node graphs and vice versa. Although it is not known how to perform Min-Plus product of two arbitrary $n \times n$ matrices in time $O(n^{3-\epsilon})$ for $\epsilon > 0$, several problems reduce to Min-Plus products of matrices A and B which have nice structures that can be exploited. The simplest examples result by restricting edge weights in APSP problems [213, 216, 244, 54, 243]. Bringmann et al. [39] show Language Edit Distance, RNA-folding, and Optimum Stack Generation can be reduced to Min-Plus product where matrix A has small difference between adjacent entries in each row and column. Finally, the recent work of Williams and Xu [238] reduces APSP in certain geometric graphs, batch range mode, and the maximum subarray problem with entries bounded by $O(n^{0.62})$ to a more general structured Min-Plus product, extending the result of Bringmann et al. All of the above structured Min-Plus products are solvable in truly subcubic $O(n^{3-\epsilon})$ time for $\epsilon > 0$, improving algorithms in the problems reduced to said product.

The connection and upper bound established by Williams and Xu [238] of batch range mode to Min-Plus product suggest other versions of the range mode problem may be amenable to similar improvements. In particular, the ability to efficiently compute a batch of range mode queries via reducing to a structured Min-Plus product suggests that one

[1] We use the $\tilde{O}(\cdot)$ notation to hide poly-logarithmic factors.

might be able to improve the update time of dynamic range mode in a similar way.

5.1.1 Our Results

In this chapter, we break the $\Theta(n^{2/3})$ time per operation barrier for dynamic range mode. We do so by adapting the result of Williams and Xu [238]. Specifically, we define the following new type of data structure problem on the Min-Plus product that can be applied to dynamic range mode, which may be of independent interest. Then we combine this data structure problem with the algorithm of Williams and Xu.

Problem 58 (*Min-Plus-Query* problem)**.** *During initialization, we are given two matrices A, B with dimensions compatible for matrix multiplication. For each query, we are given three parameters i, j, S, where i, j are two integers, and S is a set of integers. The query asks $\min_{k \notin S}\{A_{i,k} + B_{k,j}\}$.*

Our performance theorem is the following.

Theorem 59. *There exists a deterministic data structure for dynamic range mode on a sequence $a_1, \ldots, a_n$ that supports query, insertion, and deletion in worst-case $\tilde{O}(N^{0.655994})$ time per operation, where N is the maximum size of the sequence at any point in time. The space complexity of the data structure is $\tilde{O}(N^{1.327997})$.*

For a discussion of the per-operation time complexity and space complexity with reference to the rectangular matrix multiplication constant $\omega(k)$ (see Section 5.3 for a formal definition), see Section 5.4, under the second to last header "Time and Space Complexity".

Our result shows yet another application of the Min-Plus product to an independently-studied problem, ultimately showing a dependence of the complexity of dynamic range mode on the complexity of fast matrix multiplication. Further, in contrast to many other reductions to Min-Plus in which we must assume a structured input on the original problem [213, 216, 244, 54, 243, 238], our algorithm works on the fully general dynamic range mode problem. In this sense, our result is perhaps most directly comparable to the batch range mode reduction of Williams and Xu [238] and the Language Edit Distance, RNA-folding, and Optimum Stack Generation reductions of Bringmann et al. [39].

5.1.2 Discussion of Technical Difficulty

Despite the new $\tilde{O}(n^{1.4854})$ time algorithm for batch range mode [238], we cannot directly apply the result to dynamic range mode. The main issue is the element deletion operation. In the range mode algorithm of Williams and Xu (and in many other range mode algorithms), critical points are chosen evenly distributed in the array, and the algorithm precomputes the range mode of intervals between every pair of critical points. In [238], the improvement is achieved via a faster precomputation algorithm, which uses a Min-Plus product algorithm for structured matrices. However, if element deletion is allowed, the results stored in the precomputation will not be applicable. For example, an interval between two critical points could contain x copies of element a, $x-1$ copies of element b, and many other elements with frequencies less than $x-1$. During precomputation, the range mode of this interval would be a. However, if we delete two copies of a, there is no easy way to determine that the mode of this interval has now changed to b.

We overcome this difficulty by introducing the Min-Plus-Query problem, as defined in Section 5.1.1. Intuitively, in the Min-Plus-Query problem, a large portion of the work of the Min-Plus product is put off until the query. It also supports more flexible queries. Using the Min-Plus-Query problem as a subroutine, we will be able to query the most frequent element excluding a set S of forbidden elements. For instance, in the preceding example, we would be able to query the most frequent element that is not a. This is the main technical contribution of this chapter.

Another major difference between our algorithm for dynamic range mode and the batch range mode algorithm of Williams and Xu [238] is the need for rectangular matrix multiplication. In our algorithm, we treat elements that appear more than about $N^{2/3}$ times differently from the rest (a similar treatment is given in the dynamic range mode algorithm of Hicham et al. [80]). However, the number of critical points we use is about $N^{1/3}$; thus the number of critical points and frequent elements differ. This contrasts with batch range mode, where elements that appear more than about $\sqrt{n}$ times are considered frequent and the number of critical points used coincides with the number of frequent elements. The consequence of this difference is that a rectangular matrix product is required for dynamic range mode, while a square matrix product sufficed in [238].

5.2 Related Work

The range mode problem was first studied formally by Krizanc et al. [167]. They study space-efficient data structures for static range mode, achieving a time-space tradeoff of $O(n^{2-2\epsilon})$ space and $O(n^{\epsilon}\log n)$ query time for any $0 < \epsilon \le 1/2$. They also give a solution occupying $O(n^2 \log\log n/\log n)$ space with $O(1)$ time per query.

Chan et al. [55] also study static range mode, focusing on linear space solutions. They achieve a linear space data structure supporting queries in $O(\sqrt{n})$ time via clever use of arrays, which can be improved to $O(\sqrt{n/\log n})$ time via bit-packing tricks. Their paper also introduces the conditional lower bound which reduces multiplication of two $\sqrt{n} \times \sqrt{n}$ boolean matrices to n range mode queries on an array of size $O(n)$. As mentioned, combined with the presumed hardness of the online matrix vector problem [132], this result indicates a dynamic range mode data structure must take greater than $O(n^{1/2-\epsilon})$ for $\epsilon > 0$ time per operation. Finally, Chan et al. [55] also give the first data structure for dynamic range mode. At linear space, their solution achieves $O(n^{3/4} \log n / \log \log n)$ worst-case time per query and $O(n^{3/4} \log \log n)$ amortized expected time update, and at $O(n^{4/3})$ space, their solution achieves $O(n^{2/3} \log n / \log \log n)$ worst-case time query and amortized expected update time.

Recently, El-Zein et al. [80] improved the runtime of dynamic range mode to worst-case $O(n^{2/3})$ time per operation while simultaneously improving the space usage to linear. Prior to this work, this result was the fastest data structure for dynamic range mode.

A cell-probe lower bound for static range mode has been devised by Greve et al. [123]. Their result states that any range mode data structure that uses S memory cells of w-bit words needs $\Omega(\frac{\log n}{\log(Sw/n)})$ time to answer a query.

Via reduction to a structured Min-Plus product, Williams and Xu [238] recently showed that n range mode queries on a fixed array of size n can be answered in $\tilde{O}(n^{(27+2\omega)/(19+\omega)})$ time, which is $\tilde{O}(n^{1.4854})$ time for $\omega < 2.373$. Williams and Xu actually show how to compute the *frequency* of the mode for each query. We can adapt this method to find the element that is mode using the following binary search. For query $[l, r]$, we ask the frequency of the mode in range $[l, (l + r)/2]$. If it is the same, we repeat the search with right endpoint in range $[l, (l + r)/2]$; if it is not, we repeat the search with right endpoint in range $[(l + r)/2, r]$. Using this method, we can binary search until we determine when the frequency of the mode changes, thus finding the element that is mode in an additional $O(\log n)$ queries. The algorithm of Williams and Xu can also be used to speed up the preprocessing time of the $O(n)$ space, $O(\sqrt{n})$ query time static range mode data structure to $\tilde{O}(n^{1.4854})$ time.

Both static and dynamic range mode have been studied in approximate settings [37, 123, 79].

5.3 Preliminaries

We formally define the Min-Plus product problem and the dynamic range mode problem.

Problem 60 (Min-Plus product)**.** *The Min-Plus product of an $m \times n$ matrix A and an $n \times p$ matrix B is the $m \times p$ matrix $C = A \star B$ such that $C_{i,j} = \min_k\{A[i,k] + B[k,j]\}$.*

Problem 61 (Dynamic Range Mode)**.** *In the dynamic range mode problem, we are given an initially empty sequence and must support the following operations:*

- *Insert an element at a given position of the sequence.*

- *Delete one element of the sequence.*
- *Query the most frequent element of any contiguous subsequence. If there are multiple answers, output any.*

It is guaranteed that the size of the array does not exceed N at any point in time.

We use ω to denote the square matrix multiplication exponent, i.e. the smallest real number such that two $n \times n$ matrices can be multiplied in $n^{\omega+o(1)}$ time. The current bound on ω is $2 \le \omega < 2.373$ [106, 231]. In this work, we will use fast rectangular matrix multiplication. Analogous to the square case, we use $\omega(k)$ to denote the exponent of rectangular matrix multiplication, i.e., the smallest real number such that an $n \times n^k$ matrix and an $n^k \times n$ matrix can be multiplied in $n^{\omega(k)+o(1)}$ time. Le Gall and Urrutia [107] computed smallest upper bounds to date for various values of k. In this work, we are mostly interested in values of $\omega(k)$ listed in Figure 5.1.

k	Upper Bound on $\omega(k)$
1.75	3.021591
2	3.251640

Figure 5.1: Upper bounds for the exponent of multiplying an $n \times n^k$ matrix and an $n^k \times n$ matrix [107].

It is known that the function $\omega(k)$ is convex for $k > 0$ (see e.g. [168], [173]), so we can use values of $\omega(p)$ and $\omega(q)$ to give upper bounds for $\omega(k)$ as long as $p \le k \le q$.

Fact 62. *When $0 < p \le k \le q$, $\omega(k) \le \frac{k-p}{q-p}\omega(q) + \frac{q-k}{q-p}\omega(p)$.*

Combining Figure 5.1 and Fact 62, we obtain the following bound on $\omega(k)$ when $k \in [1.75, 2]$.

Corollary 63. *When $1.75 \le k \le 2$, $\omega(k) \le 0.920196k + 1.41125$.*

5.4 Main Algorithm

A main technical component for our dynamic range mode algorithm is the use of the *Min-Plus-Query* problem, which is formally defined in Section 5.1. We are given two matrices A, B. For each query, we are given three parameters i, j, S, and we need to compute $\min_{k \notin S}\{A_{i,k} + B_{k,j}\}$.

If we just use the Min-Plus-Query problem, we can only compute the frequency of the range mode. Although we can binary search for the most frequent element as described in Section 5.2, we are also able to return the witness from the Min-Plus-Query problem organically. This construction may be of independent interest.

Problem 64 (*Min-Plus-Query-Witness* problem)**.** *During initialization, we are given two matrices A, B. For each query, we are given three parameters i, j, S, where i, j are two integers, and S is a set of integers. We must output an index $k^* \notin S$ such that $A_{i,k^*} + B_{k^*,j} = \min_{k \notin S}\{A_{i,k} + B_{k,j}\}$.*

If A is an $n \times n^s$ matrix and B is an $n^s \times n$ matrix, then the naive algorithm for Min-Plus-Query just enumerates all possible indices k for each query, which takes $O(n^s)$ time per query. In order to get a faster algorithm for dynamic range mode, we need to achieve $\tilde{O}(n^{2+s-\epsilon})$ preprocessing time and $\tilde{O}(n^{s-\epsilon} + |S|)$ query time, for some $\epsilon > 0$, where A, B are some special matrices generated by the range mode instance. Specifically, matrix B meets the following two properties:

1. Each row of B is non-increasing;
2. The difference between the sum of elements in the j-th column and the sum of elements in the $(j+1)$-th column is at most n^s, for any j.

Williams and Xu [238] give a faster algorithm for multiplying an arbitrary matrix A with such matrix B, which leads to a faster algorithm for static range mode. We will show that nontrivial data structures exist for the Min-Plus-Query problem for such input matrices A and B. Such a data structure will lead to a faster algorithm for dynamic range mode.

In the following lemma, we show a data structure for the Min-Plus-Query problem when both input matrices have integer weights small in absolute value.

Lemma 65. *Let $s \geq 1$ be a constant. Let A and B be two integer matrices of dimension $n \times n^s$ and $n^s \times n$, respectively, with entries in $\{-W, \dots, W\} \cup \{\infty\}$ for some $W \geq 1$. Then we can solve the Min-Plus-Query problem of A and B in $\tilde{O}(Wn^{\omega(s)})$ preprocessing time and $\tilde{O}(|S|)$ query time. The space complexity is $\tilde{O}(Wn^2 + n^{1+s})$.*

Proof. The algorithm uses the idea by Alon, Galil and Margalit in [9], which computes the Min-Plus product of A, B in $\tilde{O}(Wn^{\omega(s)})$ time.

In their algorithm, they first construct matrix A' defined by

$$A'_{i,k} = \begin{cases} (n^s+1)^{A_{i,k}+W} & \text{if } A_{i,k} \neq \infty, \\ 0 & \text{otherwise.} \end{cases}$$

We can define B' similarly. Then the product $A'B'$ captures some useful information about the Min-Plus product of A and B. Namely, for each entry $(A'B')_{i,j}$, we can uniquely write it as $\sum_{t\geq 0} r_t^{i,j}(n^s+1)^t$ for integers $0 \leq r_t^{i,j} \leq n^s$. Note that $r_t^{i,j}$ exactly equals the number of k such that $A_{i,k} + B_{k,j} = t - 2W$. Thus, we can use $A'B'$ to compute the Min-Plus Product of A and B.

In our algorithm, we use a range tree to maintain the sequence $r_t^{i,j}$ for each pair of i, j. The preprocessing takes $\tilde{O}(Wn^{\omega(s)})$ time, which is the time to compute $A'B'$ and the sequences $r_t^{i,j}$.

During each query, we are given i, j, S. We enumerate each $k \in S$, and decrement $r^{i,j}_{A_{i,k}+B_{k,j}+2W}$ in the range tree if $A_{i,j} + B_{k,j} < \infty$. After we do this for every $k \in S$, we query the range tree for the smallest t such that $r_t^{i,j} \neq 0$, so $t - 2W$ is the answer to the Min-Plus-Query query. After each query, we need to restore the values of $r^{i,j}$, which can also be done efficiently. The query time is $\tilde{O}(|S|)$, since each update and each query of range tree takes $\tilde{O}(1)$ time. The space complexity should be clear from the algorithm. □

In the previous lemma, the data structure only answers the Min-Plus-Query problem. In all subsequent lemmas, the data structure will be able to handle the Min-Plus-Query-Witness problem.

In the next lemma, we use Lemma 65 as a subroutine to show a data structure for the Min-Plus-Query-Witness problem when only matrix A has small integer weights in absolute value.

Lemma 66. *Let $s \geq 1$ be a constant. Let A and B be two integer matrices of dimension $n \times n^s$ and $n^s \times n$, respectively, where A has entries in $\{-W, \dots, W\} \cup \{\infty\}$ for some $W \geq 1$, and B has arbitrary integer entries represented by* polylog n *bit numbers. Then for every integer $1 \leq P \leq n^s$, we can solve the Min-Plus-Query-Witness problem of A and B in $O(\frac{n^s}{P}Wn^{\omega(s)})$ preprocessing time and $O(|S| + P)$ query time. The space complexity is $\tilde{O}(\frac{Wn^{2+s}}{P} + \frac{n^{1+2s}}{P})$.*

Proof. For simplicity, assume P is a factor of n^s. We sort each column of matrix B and put entries whose rank is between $(\ell-1)P+1$ and ℓP into the ℓ-th bucket. We use $K_{j,\ell}$ to denote the set of row indices of entries in the ℓ-th bucket of the column j. We use $L_{j,\ell}$ to denote the smallest entry value of the bucket $K_{j,\ell}$, and use $H_{j,\ell}$ to denote the largest entry value. Formally,

$$L_{j,\ell} = \min_{k\in K_{j,\ell}} B_{k,j} \quad \text{and} \quad H_{j,\ell} = \max_{k\in K_{j,\ell}} B_{k,j}.$$

For each $\ell \in [n^s/P]$, we do the following[2]. We create an $n^s \times n$ matrix B^ℓ and initialize

[2]We use $[n]$, with n integer, to denote the set $\{1, 2, \dots, n\}$.

all its entries to ∞. Then for each column j, if $H_{j,\ell} - L_{j,\ell} \le 2W$ (we will call it a small bucket), we set $B^{\ell}_{k,j} := B_{k,j} - L_{j,\ell} - W$ for all $k \in K_{j,\ell}$. We will handle the case $H_{j,\ell} - L_{j,\ell} > 2W$ (large bucket) later. Clearly, all entries in B^{ℓ} have values in $\{-W, \dots, W\} \cup \{\infty\}$, so we can use the algorithm in Lemma 65 to preprocess A and B^{ℓ} and store the data structure in D^{ℓ}. Also, for each pair (i,j), we create a range tree $\mathcal{T}^{i,j}_{\text{small}}$ on the sequence $(A \star B^1)_{i,j}, (A \star B^2)_{i,j}, (A \star B^3)_{i,j}, \dots, (A \star B^{n^s/P})_{i,j}$, which stores the optimal Min-Plus values when k is from a specific small bucket. This part takes $\tilde{O}(\frac{n^s}{P} W n^{\omega(s)})$ time. The space complexity is $\frac{n^s}{P}$ times more than the space complexity of Lemma 65, so space complexity of this part is $\tilde{O}(\frac{Wn^{2+s}}{P} + \frac{n^{1+2s}}{P})$.

We also do the following preprocessing for buckets where $H_{j,\ell} - L_{j,\ell} > 2W$. We first create a 0/1 matrix $\bar{A}$ where $\bar{A}_{i,k} = 1$ if and only if $A_{i,k} \ne \infty$. Then for each $\ell \in [n^s/P]$, we create a 0/1 matrix $\bar{B}^{\ell}$ such that $\bar{B}^{\ell}_{k,j} = 1$ if and only if $k \in K_{j,\ell}$ and $H_{j,\ell} - L_{j,\ell} > 2W$. Then we use fast matrix multiplication to compute the product $\bar{A}\bar{B}^{\ell}$. If $K_{j,\ell}$ is a large bucket, the (i,j)-th entry of $\bar{A}\bar{B}^{\ell}$ is the number of $k \in K_{j,\ell}$ such that $A_{i,k} < \infty$; if $K_{j,\ell}$ is a small bucket, the (i,j)-th entry is 0. For each pair (i,j), we create a range tree $\mathcal{T}^{i,j}_{\text{large}}$ on the sequence $(\bar{A}\bar{B}^1)_{i,j}, (\bar{A}\bar{B}^2)_{i,j}, (\bar{A}\bar{B}^3)_{i,j}, \dots, (\bar{A}\bar{B}^{n^s/P})_{i,j}$. This part takes $\tilde{O}(\frac{n^s}{P} n^{\omega(s)})$ time, which is dominated by the time for small buckets. The space complexity is also dominated by the data structures for small buckets.

Now we describe how to handle a query (i,j,S). First consider small buckets. In $O(|S|)$ time, we can compute the set of small buckets $K_{j,\ell}$ that intersect with S. For each such $K_{j,\ell}$, we can query the data structure D^{ℓ} with input $(i, j, S \cap K_{j,\ell})$ to get the optimum value when $k \in K_{j,\ell}$. For each small bucket that intersects with S, we can set its corresponding value in the range tree $\mathcal{T}^{i,j}_{\text{small}}$ to ∞, then we can compute the optimum value of all small buckets that do not intersect with S by querying the minimum value of the range tree $\mathcal{T}^{i,j}_{\text{small}}$. After this query, we need to restore all values in the range tree. It takes $\tilde{O}(|S|)$ time to handle small buckets on query.

Now consider large buckets. Intuitively, we want to enumerate indices in all large buckets $K_{j,\ell}$ such that there exists an index $k \in K_{j,\ell} \cap ([n^s] \setminus S)$ where $A_{i,k} < \infty$. However, doing so would be prohibitively expensive. We will show that we only need two such buckets. Consider three large buckets $l_1 < l_2 < l_3$. Pick any $k_1 \in T_{j,l_1}, k_3 \in T_{j,l_3}$ such that $A_{i,k_1} < \infty$. Since

$$A_{i,k_1} + B_{k_1,j} \le W + L_{j,l_2} < W + H_{j,l_2} - 2W < A_{i,k_3} + B_{k_3,j},$$

k_3 can never be the optimum. Thus, it suffices to find the smallest two buckets such that there exists an index $k \in K_{j,\ell} \cap ([n^s] \setminus S)$ where $A_{i,k} < \infty$, and then enumerate all indices in these two buckets. To find such two buckets, we can enumerate over all indices $k \in S$, and if $A_{i,k} < \infty$ we can decrement the corresponding value in the range tree $\mathcal{T}^{i,j}_{\text{large}}$. Thus, we can compute the two smallest buckets by querying the two earliest nonzero values in the range tree. We also need to restore the range tree after the query. The range tree part takes $\tilde{O}(|S|)$ time and scanning the two large buckets requires $O(P)$ time. Thus, this step takes $\tilde{O}(|S| + P)$ time.

At this point, we will know the bucket that contains the optimum index k^*. Thus, we can iterate all indices in this bucket to actually get the witness for the Min-Plus-Query-Witness query. It takes $O(P)$ time to do so.

In summary, the preprocessing time, query time, and space complexity meet the promise in the lemma statement.

□

In the following lemma, we show a data structure for the Min-Plus-Query-Witness problem when the matrix B has the *bounded difference* property, which means that nearby entries in each row have close values. The proof adapts the strategy of [238].

Lemma 67. *Let $s \geq 1$ be a constant. Let A be an $n \times n^s$ integer matrix, and let B be an $n^s \times n$ integer matrix. It is guaranteed that there exists $1 \leq \Delta \leq \min\{n, W\}$, such that for every k, $|B_{k,j_1} - B_{k,j_2}| \leq W$ as long as $\lceil j_1/\Delta \rceil = \lceil j_2/\Delta \rceil$. Then for every $L = \Omega(\Delta)$, we can solve the Min-Plus-Query-Witness problem of A and B in $\tilde{O}(\Delta^2 \frac{n^s}{L} W n^{\omega(s)} + \frac{n^{2+s}}{\Delta})$ preprocessing time and $\tilde{O}(L)$ query time, when $|S| < L$. The space complexity is $\tilde{O}(\frac{\Delta^2 W n^{2+s}}{L} + \frac{\Delta^2 n^{1+2s}}{L} + \frac{n^{2+s}}{\Delta})$.*

Proof. **Preprocessing Step 1: Create an Estimation Matrix**

First, we create a matrix $\hat{B}$, where $\hat{B}_{k,j} = B_{k,\lceil j/\Delta \rceil \Delta}$. By the property of matrix B, $|\hat{B}_{k,j} - B_{k,j}| \leq W$ for every k, j. For each pair (i, j), we compute the L-th smallest value of $A_{i,k} + \hat{B}_{k,j}$ among all $1 \leq k \leq n^s$, and denote this value by $\hat{C}^L_{i,j}$. Notice that $\hat{C}^L_{i,j} = \hat{C}^L_{i,\lceil j/\Delta \rceil \Delta}$, so it suffices to compute $\hat{C}^L_{i,j}$ when j is a multiple of Δ, and we can infer other values correctly. It takes $O(n^s)$ time to compute each $\hat{C}^L_{i,j}$, so this step takes $O(n^{2+s}/\Delta)$ time.

If we similarly define $C^L_{i,j}$ as the L-th smallest value of $A_{i,k} + B_{k,j}$ among all $1 \leq k \leq n^s$, then $|C^L_{i,j} - \hat{C}^L_{i,j}| \leq W$ by the following claim, whose proof is omitted for space constraint.

Claim. *Given two sequences $(a_k)_{k=1}^m$ and $(b_k)_{k=1}^m$ such that $|a_k - b_k| \leq W$, then the L-th smallest element of a and the the L-th smallest element of b differ by at most W.*

Also, in $\tilde{O}(n^{2+s}/\Delta)$ time, we can compute a sorted list $\mathcal{L}^{i,j}_{\text{small}}$ of indices k sorted by the value $A_{i,k} + \hat{B}_{k,j} - \hat{C}^L_{k,j}$, for every i, and every j that is a multiple of Δ.

The space complexity in this step is not dominating.

Preprocessing Step 2: Perform Calls to Lemma 66

For some integer $\rho \geq 1$, we will perform ρ rounds of the following algorithm. At the r-th round for some $1 \leq r \leq \rho$, we randomly sample $j^r \in [n]$, and let $A^r_{i,k} := A_{i,k} + B_{k,j^r} - \hat{C}^L_{i,j^r}$ and $B^r_{k,j} := B_{k,j} - B_{k,j^r}$. Clearly, $A^r_{i,k} + B^r_{k,j} = A_{i,k} + B_{k,j} - \hat{C}^L_{i,j^r}$. For each pair (i,k), we find the smallest r such that $|A^r_{i,k}| \leq 3W$. We keep these entries as they are and replace all other entries by ∞. For every (i,k), there exists at most one r such that $A^r_{i,k} \neq \infty$. Then we use Lemma 66 to preprocess A^r and B^r for every $1 \leq r \leq \rho$. Thus, this part takes $O(\rho \frac{n^s}{P} W n^{\omega(s)})$ time, for some integer P to be determined later. Note that this parameter also affects the query time. This step stores ρ copies of the data structure from Lemma 66, so the space complexity is $\tilde{O}(\rho \frac{Wn^{2+s}}{P} + \rho \frac{n^{1+2s}}{P})$.

Note that this step is the only step that uses randomization. We can use the method of [238], Appendix A, to derandomize it. We omit the details for simplicity.

Preprocessing Step 3: Handling Uncovered Pairs

For a pair (i,k), if $A^r_{i,k} \neq \infty$ for any r, we call (i,k) *covered*; otherwise, we call the pair (i,k) *uncovered*. For each pair (i,j), we enumerate all k such that $|A_{i,k} + \hat{B}_{k,j} - \hat{C}^L_{i,j}| \leq 2W$ and (i,k) is uncovered. Notice that since $A_{i,k} + \hat{B}_{k,j} - \hat{C}^L_{i,j} = A_{i,k} + \hat{B}_{k,\lceil j/\Delta \rceil \Delta} - \hat{C}^L_{i,\lceil j/\Delta \rceil \Delta}$, we only need to exhaustively enumerate all $k \in [n^s]$ when j is a multiple of Δ. Thus, if the total number of (i,k,j) where $|A_{i,k} + \hat{B}_{k,j} - \hat{C}^L_{i,j}| \leq 2W$ and (i,k) is uncovered is X, then we can enumerate all such triples (i,k,j) in $O(X + n^{2+s}/\Delta)$ time.

It remains to bound the total number of triples that satisfy the condition. Fix an arbitrary pair (i,k), and suppose the number of j such that $|A_{i,k} + \hat{B}_{k,j} - \hat{C}^L_{i,j}| \leq 2W$ is at least $(10+s)n \ln n/\rho$. Then with probability at least $1 - (1 - \frac{(10+s)\ln n}{\rho})^\rho \geq 1 - \frac{1}{n^{10+s}}$, we pick a j^r where $|A_{i,k} + \hat{B}_{k,j^r} - \hat{C}^L_{i,j^r}| \leq 2W$. Therefore,

$$|A^r_{i,k}| = \left|A_{i,k} + B_{k,j^r} - \hat{C}^L_{i,j^r}\right| \leq \left|A_{i,k} + \hat{B}_{k,j^r} - \hat{C}^L_{i,j^r}\right| + \left|\hat{B}_{k,j^r} - B_{k,j^r}\right| \leq 3W,$$

which means (i,k) is covered. Therefore, with high probability, all pairs of (i,k) where the number of j such that $|A_{i,k} + \hat{B}_{k,j} - \hat{C}^L_{i,j}| \leq 2W$ is at least $(10+s)n \ln n/\rho$ will be covered. In other words, $X = O(n^{1+s} \cdot n \ln n/\rho) = \tilde{O}(n^{2+s}/\rho)$.

For each pair (i,j), if we enumerate more than L indices k, we only keep the L values of k that give the smallest values of $A_{i,k} + B_{k,j}$. We call this list $\mathcal{L}^{i,j}_{\text{triple}}$. From previous discussion, the time cost in this step is $\tilde{O}(n^{2+s}/\rho + n^{2+s}/\Delta)$. Since we need to store all the triples, the space complexity is $O(n^{2+s}/\rho)$.

Handling Queries

Now we discuss how to handle queries. For each query (S,i,j), let $k^* = \arg\min_{k \notin S} A_{i,k} + B_{k,j}$ be the optimum index. Consider two cases:

- (i, k^*) is covered. By definition of being covered, there exists a round r such that $A^r_{i,k^*} = A_{i,k^*} + B_{k^*,j^r} - \hat{C}^L_{i,j^r}$, so $A^r_{i,k^*} + B^r_{k^*,j} = A_{i,k^*} + B_{k^*,j} - \hat{C}^L_{i,j^r}$. Therefore, we can query the data structure in Lemma 66 for every A^r and B^r and denote b^r as the result. The answer is given by the smallest value of $b^r + \hat{C}^L_{i,j^r}$ over all r. The witness is given by the data structure of Lemma 66.

  Note that when querying A^r and B^r, we only need to pass the set $\{k \in S : A^r_{i,k} \neq \infty\}$. For every $k \in S$, there is at most one r such that $A^r_{i,k} \neq \infty$, so the total size of the sets passing to the data structure of Lemma 66 is $|S|$. Thus, this case takes $O(|S| + \rho P)$ time.

- (i, k^*) is uncovered. There are still two possibilities to consider in this case.

  - **Possibility I**: $A_{i,k^*} + \hat{B}_{k^*,j} - \hat{C}^L_{i,j} < -2W$. In this case,

  $$A_{i,k^*} + B_{k^*,j} \leq A_{i,k^*} + \hat{B}_{k^*,j} + W < \hat{C}^L_{i,j} - W,$$

    so the optimum value is smaller than $\hat{C}^L_{i,j}$. By reading the list $\mathcal{L}^{i,\lceil j/\Delta \rceil \Delta}_{\text{small}}$, we can effectively find all such k where $A_{i,k} + \hat{B}_{k,j} - \hat{C}^L_{i,j} < -2W$ in time linear to the number of such k. The number of such k is at most L, by the definition of $\hat{C}^L_{i,j}$. Thus, this part takes $O(L)$ time.

  - **Possibility II**: $A_{i,k^*} + \hat{B}_{k^*,j} - \hat{C}^L_{i,j} \geq -2W$. In fact, in this case, we further have

  $$A_{i,k^*} + \hat{B}_{k^*,j} - \hat{C}^L_{i,j} \leq A_{i,k^*} + B_{k^*,j} - C^L_{i,j} + 2W \leq 2W,$$

    where $A_{i,k^*} + B_{k^*,j} - C^L_{i,j} \leq 0$ because $|S| < L$. Therefore, in this case, we have $|A_{i,k^*} + \hat{B}_{k^*,j} - \hat{C}^L_{i,j}| \leq 2W$, so we can enumerate all indices in $\mathcal{L}^{i,j}_{\text{triple}}$ and take the best choice. This takes $O(L)$ time.

Time and Space Complexity

In summary, the preprocessing time is

$$\tilde{O}\left(\rho \frac{n^s}{P} W n^{\omega(s)} + n^{2+s}/\Delta + n^{2+s}/\rho\right),$$

and the query time is $\tilde{O}(L + \rho P)$. To balance the terms, we can set $\rho = \Delta$ and $P = \frac{L}{\Delta}$ to achieve a $\tilde{O}(\Delta^2 \frac{n^s}{L} W n^{\omega(s)} + \frac{n^{2+s}}{\Delta})$ preprocess time and a $\tilde{O}(L)$ query time. Note that since we need $P \geq 1$, we must have $L = \Omega(\Delta)$.

From the preprocessing steps, the space complexity is $\tilde{O}(\rho\frac{Wn^{2+s}}{P} + \rho\frac{n^{1+2s}}{P} + n^{2+s}/\rho)$. Plugging in $\rho = \Delta$ and $P = \frac{L}{\Delta}$ reduces this to

$$\tilde{O}\left(\frac{\Delta^2 W n^{2+s}}{L} + \frac{\Delta^2 n^{1+2s}}{L} + \frac{n^{2+s}}{\Delta}\right),$$

as given in the statement of the lemma. □

The next lemma is our last data structure for Min-Plus-Query-Witness problems.

Lemma 68. *Let $s \geq 1$ be a constant. Let A be an $n \times n^s$ integer matrix and B be an $n^s \times n$ integer matrix. Suppose matrix B satisfies*

1. *Each row of B is non-increasing;*
2. *The difference between the sum of elements in the j-th column and the sum of elements in the $(j+1)$-th column is at most n^s, for any j.*

Then for every positive integer $L = \Omega(n^{\omega(s)-2})$, we can solve the Min-Plus-Query-Witness problem of A and B in $\tilde{O}(n^{\frac{8}{5}+s+\frac{1}{5}\omega(s)}L^{-\frac{1}{5}})$ preprocessing time and $\tilde{O}(L)$ query time, when $|S| < L$. The space complexity is $\tilde{O}(L^{-\frac{1}{5}}n^{\frac{18}{5}+s-\frac{4}{5}\omega(s)} + L^{-\frac{3}{5}}n^{\frac{9}{5}+2s-\frac{2}{5}\omega(s)} + L^{-\frac{1}{5}}n^{\frac{8}{5}+s+\frac{1}{5}\omega(s)})$.

Proof. Let $\Delta, W \geq 1$ be small polynomials in n to be fixed later. Define $I(j)$ to be the interval $[j - \Delta + 1, j]$.

Let j' be any multiple of Δ. By property **2** of matrix B, $\sum_{k=1}^{n^s} B_{k,j} - \sum_{k=1}^{n^s} B_{k,j+1} \leq n^s$ for any $j \in I(j')$. Thus, we have

$$\sum_{k=1}^{n^s} B_{k,j'-\Delta+1} - \sum_{k=1}^{n^s} B_{k,j'} \leq \Delta n^s.$$

By averaging, there are at most $\Delta n^s/W$ indices $k \in [n^s]$ such that $B_{k,j'-\Delta+1} - B_{k,j'} > W$. We create a new matrix $\hat{B}$, initially the same as matrix B. For each k such that $B_{k,j'-\Delta+1} - B_{k,j'} > W$, and for each $j \in I(j')$, we set $\hat{B}_{k,j}$ as M, where M is some large enough integer. After this replacement, $\hat{B}_{k,j'-\Delta+1} - \hat{B}_{k,j'} \leq W$ for any k and any j' multiple of Δ. Also, since $\hat{B}_{k,j'-\Delta+1} \geq \hat{B}_{k,j} \geq \hat{B}_{k,j'}$ for any $j \in I(j')$, we have that $|\hat{B}_{k,j_1} - \hat{B}_{k,j_2}| \leq W$ as long as $\lceil j_1/\Delta \rceil = \lceil j_2/\Delta \rceil$. Therefore, we can use Lemma 67 to preprocess A and $\hat{B}$ in $O(\Delta^2 \frac{n^s}{L} W n^{\omega(s)} + \frac{n^{2+s}}{\Delta})$ time. The space complexity is $\tilde{O}(\frac{\Delta^2 W n^{2+s}}{L} + \frac{\Delta^2 n^{1+2s}}{L} + \frac{n^{2+s}}{\Delta})$.

On the other hand, note that $\hat{B}$ differs with B on at most $n^{1+s}\Delta/W$ entries, so we need to do some extra preprocessing to handle those entries. For each pair (i, j), we initialize a range tree $\mathcal{T}^{(i,j)}$ whose elements are all ∞ (it takes $\tilde{O}(1)$ time to initialize each range tree if we implement it carefully). Then for every k such that $B_{k,j} \neq \hat{B}_{k,j}$, we set the k-th element in $\mathcal{T}^{(i,j)}$ as $A_{i,k} + B_{k,j}$. The total number of operations we perform in all the range trees are $O(n^{2+s}\Delta/W)$, so this part takes $\tilde{O}(n^{2+s}\Delta/W)$ time. The space complexity is also $\tilde{O}(n^{2+s}\Delta/W)$.

During a query (S, i, j), we first query the data structure in Lemma 67 on matrix A and $\hat{B}$ with parameters (S, i, j). Then we query the minimum value from the range tree $\mathcal{T}^{(i,j)}$ after setting all $A_{i,k} + B_{k,j}$ as ∞ for $k \in S$. Taking the minimum of these two queries gives the answer. The optimum index k^* is either given by the data structure of Lemma 67 or can be obtained from the range tree.

Thus, the preprocessing time of the algorithm is

$$\tilde{O}(\Delta^2 \frac{n^s}{L} W n^{\omega(s)} + \frac{n^{2+s}}{\Delta} + n^{2+s}\Delta/W),$$

and the query time is $\tilde{O}(L)$. We get the desired preprocessing time by setting $\Delta = L^{1/5} n^{\frac{2}{5} - \frac{1}{5}\omega(s)}$ and $W = \Delta^2$. Since we need $\Delta \geq 1$, we require that $L = \Omega(n^{\omega(s)-2})$. In Lemma 67, we also requires that $L = \Omega(\Delta)$, but this is always true when $L = \Omega(n^{\omega(s)-2})$.

By previous discussion, the space complexity is $\tilde{O}(\frac{\Delta^2 W n^{2+s}}{L} + \frac{\Delta^2 n^{1+2s}}{L} + \frac{n^{2+s}}{\Delta} + n^{2+s}\Delta/W)$. Plugging in the value for Δ and W simplifies the complexity to

$$\tilde{O}(L^{-1/5} n^{18/5+s-4\omega(s)/5} + L^{-3/5} n^{9/5+2s-2\omega(s)/5} + L^{-1/5} n^{8/5+s+\omega(s)/5}).$$

□

Finally, we can apply the data structure of Lemma 68 to prove Theorem 59.

Proof of Theorem 59. For clarity, we will use *element* to refer to a specific item a_i of the sequence and use *value* to refer to all elements of a given type. Given a pointer to an element of the sequence a_i, we assume the ability to look up its index i in the sequence in $\tilde{O}(1)$ time by storing all elements of the sequence in a balanced binary search tree with worst-case time guarantees (e.g. a red-black tree). Thus we can go from index i to element a_i and back via appropriate rank and select queries on the balanced binary search tree. We may also add or remove an element a_i from the sequence, and thus the binary search tree, in $\tilde{O}(1)$ time.

Let T_1, T_2, T_3 be three parameters of the algorithm. Parameter T_1 is a threshold that controls the number of "frequent" colors, T_2 controls how frequently the data structure is rebuilt, and T_3 represents the size of blocks in the algorithm.

We call values that appear more than N/T_1 times *frequent* and all other values *infrequent*. Thus, there are at most T_1 frequent values at any point in time. Note that a fixed value can change from frequent to infrequent, or from infrequent to frequent, via a deletion or insertion.

Infrequent Values

First, we discuss how to handle infrequent values. We maintain $\frac{N}{T_1}$ balanced search trees $\mathcal{BST}_1, \ldots, \mathcal{BST}_{\frac{N}{T_1}}$. For balanced search tree $\mathcal{BST}_k$, we prepare the key/value pairs in the following way. Fix a given value of the sequence. Say all its occurrences are at indices $i_1, i_2, \ldots, i_t$. Then we insert the key/value pairs (i_x, i_{x+k-1}) to $\mathcal{BST}_k$ for every $1 \le x \le t-k+1$. However, the indices themselves would need updating when sequence a is updated. Instead of inserting the indices themselves, we insert corresponding pointers to the nodes of the binary search tree that holds sequence a. That way we can perform all comparisons using binary search tree operations in $\tilde{O}(1)$ time, without needing to update indices when sequence a changes. We also augment each balanced search tree $\mathcal{BST}_i$ so that every subtree stores the smallest value y of any pair (x, y) in the subtree. After an insertion or deletion, we need to update a total of $O((\frac{N}{T_1})^2)$ pairs. Thus, we can maintain these balanced search trees in $\tilde{O}((\frac{N}{T_1})^2)$ time per operation.

During a query $[l, r]$, we iterate through all the balanced search trees $\mathcal{BST}_1, \ldots, \mathcal{BST}_{\frac{N}{T_1}}$. If there exists a pair $(i_1, i_2) \in \mathcal{BST}_k$ such that $l \le i_1 \le i_2 \le r$, then the range mode is at least k. Thus, if the range mode is an infrequent value, we can find its frequency and corresponding value by querying the balanced search trees. The query time is $\tilde{O}(\frac{N}{T_1})$, which is not the dominating term.

Newly Modified Values

We now consider how to handle frequent values. We handle newly modified values and unmodified values differently. We will rebuild our data structure after every T_2 operations, and call values that are inserted or deleted after the last rebuild *newly modified values*.

For every value, we maintain a balanced search tree of occurrences of this value in the sequence. It takes $\tilde{O}(1)$ time per operation to maintain such balanced search trees. Thus, given an interval $[l, r]$, it takes $\tilde{O}(1)$ time to query the number of occurrences of a particular value in the interval. We use this method to query the number of occurrences of each newly modified value. Since there can be at most T_2 such values, this part takes $\tilde{O}(T_2)$ time per operation.

Data Structure Rebuild

It remains to handle the frequent, not newly modified values during each rebuild. In this case, we will assume we can split the whole array roughly equally into a left half and right half. We can recursively build the data structure on these two halves so that we may assume a range mode query interval has left endpoint in the left half and right endpoint in the right half. The recursive construction adds only a poly-logarithmic factor to the complexity.

We split the left half and the right half into consecutive segments of length at most T_3, so that there are $O(N/T_3)$ segments. We call the segments $P_1, P_2, \dots, P_m$ in the left half and $Q_1, Q_2, \dots, Q_m$ in the right half, where segments with a smaller index are closer to the middle of the sequence.

Let $v_1, v_2, \dots, v_l$ be the frequent values during the rebuild. We create a matrix A such that $A_{i,k}$ equals the negation of the number of occurrences of v_k in segments $P_1, \dots, P_i$; similarly, we create a matrix B such that $B_{k,j}$ equals the negation of the number of occurrences of v_k in segments $Q_1, \dots, Q_j$. Note that the negation of the value $A_{i,k} + B_{k,j}$ is the frequency of value v_k in the interval from P_i to Q_j. It is not hard to verify that matrix B satisfies the requirement of Lemma 68. We take the negation here since Lemma 68 handles $(\min, +)$-product instead of $(\max, +)$-product. Then we use the preprocessing part of Lemma 68 with matrices A, B, and $L = T_2$. If we let $T_1 = N^{t_1}, T_2 = N^{t_2}, T_3 = N^{t_3}$, then in the notation of Lemma 68, $n = m = O(N/T_3) = O(N^{1-t_3})$ and $n^s = O(T_1) = O(N^{t_1})$, so $s = \frac{t_1}{1-t_3}$ and $L = N^{t_2}$. Thus, by Lemma 68 the rebuild takes

$$\tilde{O}(N^{(1-t_3)(\frac{8}{5}+\frac{t_1}{1-t_3}+\frac{1}{5}\omega(\frac{t_1}{1-t_3}))-\frac{1}{5}t_2})$$

time. Since we perform the rebuild every T_2 operations, the amortized cost of rebuild is

$$\tilde{O}(N^{(1-t_3)(\frac{8}{5}+\frac{t_1}{1-t_3}+\frac{1}{5}\omega(\frac{t_1}{1-t_3}))-\frac{6}{5}t_2})$$

per operation.

Now we discuss how to handle queries for frequent, unmodified elements. For a query interval $[l, r]$, we find all the segments inside the interval $[l, r]$. The set of such segments must have the form $P_1, \cup \dots \cup P_i \cup Q_1 \cup \dots \cup Q_j$ for some i, j. We scan through all elements in $[l, r] \setminus (P_1 \cup \dots \cup P_i \cup Q_1 \cup \dots \cup Q_j)$, and use their frequency to update the answer. Since the size of segments is $O(T_3)$, the time complexity to do so is $\tilde{O}(T_3)$.

For the segments $P_1, \dots, P_i, Q_1, \dots, Q_j$, we query the data structure in Lemma 68 with S being the set of newly modified elements. The answer will be the most frequent element in the interval from P_i to Q_j that is not newly modified. By Lemma 68 this takes $O(L) = O(N^{t2})$ time per operation.

Time and Space Complexity

In summary, the amortized cost per operation is

$$\tilde{O}(N^{2-2t_1} + N^{t_2} + N^{t_3} + N^{(1-t_3)(\frac{8}{5}+\frac{t_1}{1-t_3}+\frac{1}{5}\omega(\frac{t_1}{1-t_3}))-\frac{6}{5}t_2}).$$

To balance the terms, we set $t_1 = 1 - \frac{1}{2}t_2$, and $t_3 = t_2$. The time complexity thus becomes

$$\tilde{O}(N^{t_2} + N^{(1-t_2)(\frac{8}{5}+\frac{1-0.5t_2}{1-t_2}+\frac{1}{5}\omega(\frac{1-0.5t_2}{1-t_2}))-\frac{6}{5}t_2}).$$

By observation, we can note that the optimum value of $\frac{1-0.5t_2}{1-t_2}$ lies in $[1.75, 2]$. Thus, we can plug in Corollary 63 and use $t_2 = 0.655994$ to balance the two terms. This gives an $\tilde{O}(N^{0.655994})$ *amortized* time per operation algorithm.

The space usage has two potential bottlenecks. The first is the space to store $\mathcal{BST}_1, \ldots, \mathcal{BST}_{\frac{N}{T_1}}$ for handling infrequent elements, which is $\tilde{O}(\frac{N^2}{T_1})$. The second is the space used by Lemma 68, which is

$$\tilde{O}(N^{-t_2/5}N^{(1-t_3)(18/5+s-4\omega(s)/5)} + N^{-3t_2/5}N^{(1-t_3)(9/5+2s-2\omega(s)/5)} + N^{-t_2/5}N^{(1-t_3)(8/5+s+\omega(s)/5)}).$$

By plugging in the values for t_2, t_3 and s, the space complexity becomes $\tilde{O}(N^{1.327997})$, with the $\tilde{O}(\frac{N^2}{T_1})$ term being the dominating term.

Worst-Case Time Complexity

By applying the *global rebuilding* of Overmars [194], we can achieve a worst-case time bound. The basic idea is that after T_2 operations, we don't immediately rebuild the Min-Plus-Query-Witness data structure. Instead, we rebuild the data structure during the next T_2 operations, spreading the work evenly over each operation. To answer queries during these T_2 operations, we use the previous build of the Min-Plus-Query-Witness data structure. By this technique, the per-operation runtime can be made worst-case.

□

Chapter 6

Approximate Succinct Range Mode and Range Selection

6.1 Introduction

The mode and median of a data set are important statistics, widely used across many disciplines. Thus, they are frequently computed in applications for data mining, information retrieval and data analytics. The range mode and median problems further aim at speeding up the computation of the mode and median in an arbitrary subrange of the given sequence of elements, and thus have been studied extensively [167, 37, 203, 204, 44, 123, 144, 44, 131, 55, 111, 58, 124, 80]. In these problems, we preprocess a sequence of elements $c_1, c_2, \ldots, c_n$ to answer queries. Given two indices a and b with $1 \le a \le b \le n$, a *range mode query* asks for a position of the most frequent element in $c_{a..b}$ ($c_{a..b}$ denotes $c_a, \ldots, c_b$), while a *range median query* asks for the position of the median element in $c_{a..b}$. A generalization of range median is the *range selection query*, which asks for the position of the k^{th} smallest element in $c_{a..b}$ for any given k. Thus a range selection query becomes range median if $k = \lceil (b-a+1)/2 \rceil$.

Due to the massive amounts of electronic data available, linear space data structures are often preferred by modern applications. The following are the best solutions to these query problems that use $O(n)$ words of space. In static settings, Chan et al. [55] showed how to answer a range mode query in $O(\sqrt{n/\lg n})$ time. By proving a conditional lower bound, they also gave strong evidence that, if linear space is required, this query time cannot be improved significantly using purely combinatorial methods with current knowledge. When updates to elements are allowed, El-Zein et al. [80] showed how to support both range mode queries and updates in $O(n^{2/3})$ time. For range selection, the solution of Chan and Wilkinson [58] answers queries in $O(\lg k/\lg\lg n + 1)$ time, matching the lower bound of Jørgensen and Larsen [144] under the cell probe model. He et al. [131] showed how to support range selection in $O((\lg n/\lg\lg n)^2)$ worst-case time and updates in $O((\lg n/\lg\lg n)^2)$ amortized time.

The query times for range mode in both linear space data structure solutions and conditional lower bounds are much larger than that for many other query problems, including range median. To provide faster support for queries, researchers have studied *approximate range mode* [123]. To define this query, let $F_x(c_{a..b})$ denote the frequency of an element x in $c_{a..b}$ and $F(c_{a..b})$ denote the frequency of the mode of $c_{a..b}$ ($F(c_{a..b}) = \max_x F_x(c_{a..b})$). Then a $(1+\varepsilon)$-approximate range mode query asks for the position of an element x in $c_{a..b}$ such that $(1+\varepsilon)\cdot F_x(c_{a..b}) \geq F(c_{a..b})$ for some positive ε. This element is called a $(1+\varepsilon)$-approximate mode of $c_{a..b}$. Previously, the best result on this problem is that of Greve et al. [123], which uses $O(n/\varepsilon)$ words of space to support queries in $O(\lg(1/\varepsilon))$ time, for any $\varepsilon \in (0,1)$.

Approximate range median can be defined similarly. We say that the ith smallest element in the query range $c_{a..b}$ has rank i. Then, for an approximation ratio $\alpha \in (0, 1/2)$, an α-approximate range median query asks for the position of an element x whose rank in $c_{a..b}$ is between $\lceil s/2 \rceil - \alpha s$ and $\lceil s/2 \rceil + \alpha s$, where $s = b - a + 1$. Bose et al. [37] studied this problem, for which they proposed a data structure occupying $O(n/\alpha)$ words of space that answers queries in constant time. An α-approximate range selection query can also be defined, which, for any given k, asks for the position of an element x whose rank in $c_{a..b}$ is between $k - \alpha s$ and $k + \alpha s$. However, this problem has not been formally studied previously.

To further improve the space efficiency of data structures, researchers have recently studied various query problems in the encoding model [92, 124]. Under this model, a data structure is not allowed to store or assume access to the original data set. Instead, it should occupy as little space as possible while providing support for queries. For example, in this model, Fischer and Heun [92] studied the range minimum query problem, which asks for the position of the smallest element in $c_{a..b}$. They proposed a data structure occupying only $2n + o(n)$ bits with constant query time. The range selection problem has also been considered in this model: Grossi et al. [124] proposed an encoding data structure occupying $O(n \lg \kappa)$ bits for any fixed positive integer κ, using which a range selection query can be answered in $O(\lg k/\lg\lg n + 1)$ time for any k given in the query with $1 \leq k \leq \kappa$.

Naturally, encoding data structures are only relevant when their space occupancy is asymptotically less than the input data, at least for certain choices of parameters. The space costs of previous results on approximate range mode or median, however, match the size of the input sequence asymptotically when ε or α is a constant and become superlinear when ε or α is in $o(1)$. Thus, we study the problem of designing encoding data structures

Query Type	Query Time	Update Time	Space in Bits	Source
Exact	$O(n^{\delta}\log n)$	-	$O(n^{2-2\delta}\lg n)$	[167]
	$O(1)$	-	$O(n^2\log\log n/\lg n)$	[204]
	$O(\sqrt{n/\log n})$	-	$O(n\lg n)$	[55]
	$O(n^{3/4}\log n/\log\log n)$	$O(n^{3/4}\log\log n)$	$O(n\lg n)$	[55]
	$O(n^{2/3}\log n/\log\log n)$	$O(n^{2/3}\log n/\log\log n)$	$O(n^{4/3}\lg n)$	[55]
	$O(n^{2/3})$	$O(n^{2/3})$	$O(n\lg n)$	[80]
$(1+\varepsilon)$– Approximation	$O(\lg\lg n + \lg(1/\varepsilon))$	-	$O(n\lg n/\varepsilon)$	[37]
	$O(\lg(1/\varepsilon))$	-	$O(n\lg n/\varepsilon)$	[123]
	$O(\lg(1/\varepsilon))$	-	$O(n/\varepsilon)$	new
	$O(\lg m/\lg\lg m)$	$O(\lg n/\varepsilon^2)$	$O(m\lg m)$	new

Table 6.1: Static and Dynamic Range Mode Query History. In this table, δ is an arbitrary constant in $(0, 1/2)$ and $m = \min(n\lg n/\varepsilon, n/\varepsilon^2)$.

of approximate range mode, median and selection queries, to improve the space efficiency of previous solutions. Furthermore, previously no research has been done on dynamic approximate range mode, while the dynamic exact data structures for range mode require polynomial query and update times. Therefore, we also study approximate range mode queries under dynamic settings, to provide substantially faster support for queries and updates.

Our Results. For $(1+\varepsilon)$-approximate range mode, where $0 < \varepsilon < 1$, we design an encoding data structure using $O(n/\varepsilon)$ bits that can answer a query in $O(\lg(1/\varepsilon))$ time. This is an improvement upon the previous best result of Greve et al. [123], since we match their query time while saving the space cost by a factor of $\lg n$; we assume a word RAM model in which each word has $\Theta(\lg n)$ bits. We also prove a lower bound to show that any data structure supporting $(1+\varepsilon)$-approximate range mode must use $\Omega(n/(1+\varepsilon))$ bits for any positive ε. This means that our space cost is asymptotically optimal for constant ε. When ε is not necessarily a constant, as long as $\varepsilon = \omega(1/\lg n)$, our data structure uses $o(n\lg n)$ bits, i.e., $o(n)$ words, which is asymptotically less than the space needed to encode the original sequence itself.

For α-approximate range selection, where $0 < \alpha < 1/2$, we design encoding data structures for two variants of this problem. If k is fixed and given in advance, either as a constant or as a function of the size, s, of the query range satisfying certain reasonable constraints (e.g., $k = \lceil s/2 \rceil$ for range median), we have a solution occupying $O(n/\alpha^2)$ bits that can

Query Type	Query Time	Update Time	Space in Bits	Source
Exact	$O(1)$	-	$O((n\lg\lg n)^2/\lg n)$	[204]
	$O(\lg n/\lg\lg n)$	-	$O(n\log n)$	[44]
	$O(\lg k/\lg\lg n+1)$	-	$O(n\lg n)$	[58]
	$O(\lg^2 n)$	$O(\lg^2 n)$	$O(n\lg^2 n)$	[44]
	$O((\lg n/\lg\lg n)^2)$	$O((\lg n/\lg\lg n)^2)$	$O(n\lg^2 n/\lg\lg n)$	[44]
	$O((\lg n/\lg\lg n)^2)$	$O((\lg n/\lg\lg n)^2)$	$O(n\lg n)$	[131]
α – Approximation (with fixed k)	$O(1)$	-	$O(n\lg n/\alpha)$	[37]
	$O(1)$	-	$O(n/\alpha^2)$	new
α – Approximation	$O(1)$	-	$O(n/\alpha^3)$	new

Table 6.2: Static and Dynamic Range Median and Selection Query History.

answer a query in constant time. If k is not known beforehand and different values of k could be given with each query, we have another encoding structure in $O(n/\alpha^3)$ bits with constant query time. Our query time matches that of the previous best data structure of Bose et al. [37] which supports range median only, while we decrease the space cost by a factor of $\lg n$ when α is a constant. As we also show that any approximate range selection data structure must use at least $\Omega(n)$ bits, our data structures are asymptotically optimal for constant α.

In dynamic settings, for any $\varepsilon \in (0,1)$, we present an $O(m\lg m)$-bit structure where $m = \min(n\lg n/\varepsilon, n/\varepsilon^2)$. It supports $(1+\varepsilon)$-approximate range mode in $O(\lg m/\lg\lg m)$ time and insertions/deletions in $O(\lg n/\varepsilon^2)$ time. When ε is an arbitrary constant in $(0,1)$, this data structure uses $O(n)$ words, answers queries in $O(\lg n/\lg\lg n)$ time, and supports updates in $O(\lg n)$ time. As the best result on dynamic exact range mode [80] requires $O(n^{2/3})$ time for both queries and updates, this approximate solution is much faster for constant ε. It is also the first result on dynamic approximate range mode. Finally, we apply the technique to solve static $(1+\varepsilon)$-approximate three-sided range mode in two dimensions, achieving $O(\lg m)$ time query and occupying $O(m\lg m)$ words of space, where again $m = \min(n\lg n/\varepsilon, n/\varepsilon^2)$. This is another new approximate query problem.

Tables 6.1 and 6.2 compare our results to previous work to be surveyed in Section 6.2.

6.2 Previous Work

Range Mode. Krizanc et al. [167] first studied the static range mode problem and showed that, for any $\delta \in (0,1/2)$, there is an $O(n^{2-2\delta})$-word solution that answers queries in $O(n^\delta \log n)$ time. Setting $\delta = 1/2$ yields an $O(n)$-word data structure supporting range mode in $O(\sqrt{n} \log n)$ time. They also presented a data structure using $O(n^2 \log\log n / \log n)$ words, or $O(n^2 \log\log n)$ bits, to support queries in constant time. Chan et al. [55] further provided a better linear word solution with $O(\sqrt{n/\log n})$ query time. They also proved a conditional lower bound to show that, with current knowledge, either the query time must be polynomial, or the construction time must be polynomially larger than n. Later, Greve et al. [123] gave an (unconditional) lower bound in the cell probe model, showing that any structure using S memory cells of w-bit words requires $\Omega(\frac{\log n}{\log(Sw/n)})$ time to answer a range mode query. On the other end of the spectrum, there has been work [203, 204] on improving the constant-time query structure of Krizanc et al., and the best solution uses $O(n^2 \lg\lg n / \lg^2 n)$ words, or $O(n^2 \lg\lg n / \lg n)$ bits [204].

In dynamic settings, Chan et al. [55] provided a tradeoff among space cost, query time and update time. This tradeoff implies two important results: using linear space in words, range mode can be supported in $O(n^{3/4} \log n / \log\log n)$ worst-case time while updates can be performed in $O(n^{3/4} \log\log n)$ amortized expected time. Alternatively, they can use $O(n^{4/3})$ words to improve the query and update efficiency to $O(n^{2/3} \log n / \log\log n)$ worst-case time and amortized expected time, respectively. They also proved a conditional lower bound to show that, with current knowledge, either queries or updates must require polynomial time. Very recently, El-Zein et al. [80] further improved these solutions by designing an $O(n)$-word structure supporting both queries and updates in $O(n^{2/3})$ time.

Bose et al. [37] were the first to study approximate range mode. They showed how to provide constant-time support for 4-approximate mode, 3-approximate mode and 2-approximate mode using data structures occupying $O(n)$, $O(n \lg\lg n)$ and $O(n \lg n)$ words, respectively. For $(1+\varepsilon)$-approximation, they designed an $O(n/\varepsilon)$-word solution that can answer a query in $O(\lg\lg_{1+\varepsilon} n) = O(\lg\lg n + \lg(1/\varepsilon))$ time. Greve et al. [123] further improved these results by using $O(n/\varepsilon)$ words of space to support queries in $O(\lg(1/\varepsilon))$ time.

Range Median and Selection. The study of range median also has a rich history. It was also Krizanc et al. [167] who initially proposed this problem. There have been several solutions with near-quadratic space and constant query time [167, 203, 204], the best of which uses $O((n \lg\lg n / \lg n)^2)$ words [204]. For linear-space solutions, following a series of earlier work [167, 105, 44, 45], Brodal et al. [44] first achieved an $O(n)$-word solution that answers range median and selection queries in $O(\lg n / \lg\lg n)$ time. Jørgensen and Larsen [144] further improved the query time of range selection to $O(\lg\lg n + \lg k / \lg\lg n)$, where k is the specified query rank. They also proved that, under the cell probe model, $\Omega(\lg k / \lg\lg n + 1)$ time is necessary for any range selection data structure using $O(n \lg^{O(1)} n)$

space. Chan and Wilkinson [58] were then the first who designed a linear word solution with $O(\lg k/\lg\lg n + 1)$ optimal query time for range selection. More recently, Grossi et al. [124] proposed an encoding data structure occupying $O(n\lg\kappa)$ bits for any fixed positive integer κ, using which a range selection query can be answered in $O(\lg k/\lg\lg n + 1)$ time for any k given in the query with $1 \le k \le \kappa$. Gawrychowski and Nicholson [111] presented a space-optimal encoding of range selection which uses even less space, and proved its space cost is optimal within an $o(n)$ additive term in bits, though no support for queries is provided. All of the above results for range selection assume the selection rank k is specified at query time.

In the dynamic case, Gfeller and Sanders [44] proposed a data structure that uses $O(n\lg n)$ words of space to support range median in $O(\lg^2 n)$ time and insertions and deletions in $O(\lg^2 n)$ amortized time. The structure of Brodal et al. [44] occupies $O(n\lg n/\lg\lg n)$ words of space, answers queries in $O((\lg n/\lg\lg n)^2)$ worst-case time and supports insertions and deletions in $O((\lg n/\lg\lg n)^2)$ amortized time. Later He et al. [131] improved the space cost to $O(n)$ words while providing the same support for queries and updates. The work of Bose et al. [37] is the only work on α-approximate range median, for which they proposed a data structure occupying $O(n/\alpha)$ words of space that answers queries in constant time.

6.3 Approximate Range Mode

Before we proceed, we give a few preliminaries. We will at times refer to elements (of $c_{1..n}$ or otherwise) as *colors*. This is because their data type has no significance in frequency applications and thus the term color standardizes the data type. Furthermore, at times we create indexing such as a value r_i for when the mode in some range $c_{s_i..r_i}$ exceeds a given threshold. It is possible the mode never exceeds such a threshold. To avoid dealing with such corner cases in the rest of this exposition, we make the assumption that our list of elements $c_{1..n}$ is padded at the beginning and end with a sufficient number of one arbitrary color.

We allow non-constant ε. However, in our upper bounds, we make the restriction $\epsilon \le 1$, to allow simplification in the runtime and space analyses.

Theorem 69. *Any one-dimensional $(1+\varepsilon)$-approximate range mode data structure requires $\Omega(n/(1+\varepsilon))$ bits.*

Proof. Using a simple proof we show that $\Omega(n/(1+\varepsilon))$ bits are required for any data structure that answers one-dimensional approximate range mode queries. Here we allow arbitrary ϵ.

Given an approximation factor $1+\varepsilon$, divide the sequence S of size n into $\lfloor n/(2k) \rfloor$ *full blocks* each of size $2k$, where $k = \lceil 1+\varepsilon \rceil + 1$, and, if n is not a multiple of $2k$, a non-full block of size $n \bmod 2k$. Denote by $t_1, \ldots, t_{k+1}$ $k+1$ arbitrary, distinct colors. We say that S satisfies property $(\ast)$ if for each full block b in S one of the following two conditions hold:

- either b consists of t_1 repeated k times followed by $t_2, \ldots, t_{k+1}$,
- or b consists of $t_2, \ldots, t_{k+1}$ followed by t_1 repeated k times.

Clearly, the number of sequences that satisfy $(\ast)$ is at least $2^{\lfloor n/(2k) \rfloor}$, since there exist $\lfloor n/(2k) \rfloor$ full blocks in a sequence of size n and each of them can have one of two different values. Moreover, for any two distinct sequences S_1 and S_2 satisfying $(\ast)$ differing at full block b, there exists at least one approximate range mode query, namely the query that asks for an approximate mode of b, that will return different values (either a value from the first k position in the block or from the last k positions of the block). Thus, the information theoretic lower bound for storing an approximate range mode data structure is $\Omega(\lg 2^{\lfloor n/(2k) \rfloor}) = \Omega(\lfloor n/(2k) \rfloor) = \Omega(n)$ bits. □

We now proceed with our new upper bound. Our data structure consists of two parts. The first part answers *low* frequency queries $c_{a..b}$ with $F(c_{a..b}) \le \lceil 1/\varepsilon \rceil$, and is exact. The second part answers *high* frequency queries $c_{a..b}$ with $F(c_{a..b}) > \lceil 1/\varepsilon \rceil$, and makes use of the approximation factor.

Low Frequencies: $O(n/\varepsilon)$-Bits $O(\lg(1/\varepsilon))$ Query Time. Similar to the data structure of Greve et al. [123], for $k = 0, \ldots, \lceil 1/\varepsilon \rceil$ let Q_k be an increasing sequence of size n such that $Q_k[i]$ is the largest integer $j \ge i$ satisfying $F(c_{i..j}) = k$. Since Q_k is an increasing sequence whose largest element is n, we store it in $2n + O(n/\lg^2 n)$ bits [198] while still accessing its i^{th} element in constant time[1]. The total space used is $O(n/\varepsilon)$ bits. Given a query range $c_{a..b}$, $F(c_{a..b}) > k$ iff $b > Q_k[a]$. Thus, using binary search, we can determine if $F(c_{a..b}) < 1/\varepsilon$ and $K = F(c_{a..b})$ in that case. If $F(c_{a..b}) < 1/\varepsilon$ we return index $Q_{(K-1)}[a] + 1$; otherwise we query the high frequency structure. The total time is $O(\lg(1/\varepsilon))$.

High Frequencies: $O(n/\varepsilon)$-Bits $O(\lg\lg n + \lg(1/\varepsilon))$ Query Time. We first present an $O(n/\varepsilon)$-bit structure that answers high frequency $(1+\varepsilon)$-approximate range mode queries in $O(\lg\lg n + \lg(1/\varepsilon))$ time. We start by developing a tool to binary search the frequency of the mode, with the goal of locating a $(1+\varepsilon)$-approximate mode.

[1] We store $Q_k[1]$ and $(Q_k[i] - Q_k[i-1])$ in unary with a 0 separator between each two consecutive values in a $2n$-bit vector ψ with rank and select structures. To access $Q_k[i]$ we count the number of 1s before the i^{th} 0 in ψ.

Lemma 70. *There exists a data structure using $O(k \cdot \varepsilon \cdot n/(1+\varepsilon)^k + n/\lg^2 n)$ bits that can find in constant time, for any query range $c_{a..b}$, one of the following that holds:*

1. *$F(c_{a..b}) < (1+\varepsilon)^k/\varepsilon$,*
2. *$F(c_{a..b}) > (1+\varepsilon)^k/\varepsilon$, or*
3. *$((1+\varepsilon)^{k-1/2})/\varepsilon < F(c_{a..b}) < ((1+\varepsilon)^{k+1/2})/\varepsilon$.*

In case 2, we find an element with frequency greater than $(1+\varepsilon)^k/\varepsilon$ in range $c_{a..b}$. In case 3, we find an element with frequency greater than $((1+\varepsilon)^{k-1/2})/\varepsilon$ in range $c_{a..b}$.

When this structure is present for all k in range $0, \ldots, \lfloor \lg_{1+\varepsilon} \varepsilon n \rfloor$, the above trichotomy is sufficient to binary search for an approximate mode of frequency at least $1/\varepsilon$. If we ever land in case 3, the encoding gives an approximate mode, and otherwise, we find the k satisfying $(1+\varepsilon)^k/\varepsilon < F(c_{a..b}) < (1+\varepsilon)^{k+1}/\varepsilon$, which represents case 2 for value k and case 1 for value $k+1$. Since case 2 provides an element with frequency greater than $(1+\varepsilon)^k/\varepsilon$, this element is an approximate mode.

Proof. Let $1+\Delta = \sqrt{1+\varepsilon}$ and $f_j = (\Delta/\varepsilon)\cdot(1+\Delta)^j$. For each integer i in $[0, n/\lceil f_{2k-1}\rceil]$ let $s_i = i \cdot \lceil f_{2k-1} \rceil + 1$ and denote by r_i the smallest value such that $F(c_{s_i..r_i}) \geq (1+\Delta)^{2k}/\varepsilon$. Notice that c_{r_i} is the unique mode of $c_{s_i..r_i}$. Similarly, for each integer i in $[0, n/\lceil f_{2k}\rceil]$, let $s'_i = i \cdot \lceil f_{2k} \rceil + 1$ and denote by r'_i the smallest value such that $F(c_{s'_i..r'_i}) \geq (1+\Delta)^{2k+1}/\varepsilon$.

Given a query range $c_{a..b}$, we find the biggest indices s_i, s'_j preceding or equal to a. We proceed as follows.

1. If $b < r_i$, then $F(c_{a..b}) \leq F(c_{s_i..r_i-1}) < ((1+\Delta)^{2k}/\varepsilon) = ((1+\varepsilon)^k/\varepsilon)$.
2. If $b \geq r'_j$, then $F_{r'_j}(c_{a..b}) > F_{r'_j}(c_{s'_j..r'_j}) - f_{2k}$, since there are at most $\lceil f_{2k} \rceil - 1 < f_{2k}$ elements between s'_j and a. Then:
$$\begin{aligned} F_{r'_j}(c_{a..b}) > F_{r'_j}(c_{s'_j..r'_j}) - f_{2k} &\geq ((1+\Delta)^{2k+1}/\varepsilon) - (\Delta/\varepsilon)\cdot(1+\Delta)^{2k} = (1+\Delta)^{2k}/\varepsilon \\ &= (1+\varepsilon)^k/\varepsilon. \end{aligned}$$
3. Suppose $b \geq r_i$ and $b < r'_j$. Since there are at most $\lceil f_{2k-1} \rceil - 1 < f_{2k-1}$ elements between s_i and a and since $b \geq r_i$, we have that
$$\begin{aligned} F_{r_i}(c_{a..b}) > F_{r_i}(c_{s_i..r_i}) - f_{2k-1} &\geq ((1+\Delta)^{2k}/\varepsilon) - (\Delta/\varepsilon)\cdot(1+\Delta)^{2k-1} = ((1+\Delta)^{2k-1})/\varepsilon \\ &= ((1+\varepsilon)^{k-1/2})/\varepsilon. \end{aligned}$$
Finally, since $b < r'_j$, then $F(c_{a..b}) \leq F(c_{s'_j..r'_j-1}) < ((1+\Delta)^{2k+1})/\varepsilon = ((1+\varepsilon)^{k+1/2})/\varepsilon$.

To store the values $\{r_i\}$, we construct a bit vector of length $O(n)$ as follows. In the bit vector, there are n 0s. For each r_i, we insert a 1 bit after the r_ith 0 bit in the bit vector. Thus r_i is equal to the number of 0s before the ith 1 bit in the bit vector. A second bit vector of length $O(n)$ is used to encode the values $\{r'_i\}$ in a similar way. We then represent these two bit vectors in the succinct data structure of Patrascu [198]. This data structure provides constant time rank and select, which allow us to locate r_i and r'_j, and thus determine whether case 1, 2, or 3 applies, in constant time.

For a bit vector of size n with m ones, the space cost can be made $O(m \lg(n/m) + n/\lg^2 n)$ bits [198]. For vector r, $m \lg(n/m) = O((n/f_{2k-1}) \lg f_{2k-1})$, and for vector r', $m \lg(n/m) = O((n/f_{2k}) \lg f_{2k})$. The cost is dominated by vector r. Let us first consider the $O(m \lg(n/m))$ term. We have

$$n/f_{2k-1} \lg f_{2k-1} = \frac{\varepsilon n}{\Delta(1+\Delta)^{2k-1}} \lg((\Delta/\varepsilon) \cdot (1+\Delta)^{2k-1}). \tag{6.1}$$

Rationalizing the denominator, we can show $\frac{1}{\Delta} = \frac{1}{\sqrt{1+\varepsilon}-1} = \frac{1+\sqrt{1+\varepsilon}}{\varepsilon}$ and so $\frac{1}{\Delta} = \Theta(\frac{1}{\varepsilon})$ and $\Delta/\varepsilon = O(1)$. Thus, with $\epsilon \leq 1$, we can bound (6.1) with $O\left(\frac{(k-1/2)n}{(1+\varepsilon)^{k-1/2}} \lg(1+\varepsilon)\right)$. Finally, since we restrict $\varepsilon \leq 1$, we can do a Taylor series expansion to give $\lg(1+\varepsilon) = O(\varepsilon)$. Thus our final space bound is $O((n/f_{2k-1}) \lg f_{2k-1} + n/\lg^2 n) = O(k \cdot \varepsilon \cdot n/(1+\varepsilon)^k + n/\lg^2 n)$. $\square$

To make the above lemma useful, we must apply it to all k in range $0, \dots, \lfloor \lg_{1+\varepsilon} \varepsilon n \rfloor$. We first analyze the total space cost of all the $O(k \cdot \varepsilon \cdot n/(1+\varepsilon)^k)$ terms. Summing up these terms, we have $O\left(\sum_{k=1}^{\lfloor \lg_{1+\varepsilon} n \rfloor} (k \cdot \varepsilon \cdot n/(1+\varepsilon)^k)\right) = O\left(n \cdot \varepsilon \sum_{k=1}^{\infty} (k/(1+\varepsilon)^k)\right) = O(n/\varepsilon)$ bits. The other term comes out to $O(\lg_{1+\varepsilon}(\varepsilon \cdot n) \cdot n/\lg^2 n) \subseteq O\left(\frac{n}{\lg n \lg(1+\varepsilon)}\right)$ bits. Again applying Taylor series for $1/\lg(1+\varepsilon) = O(1/\varepsilon)$ gives $O(n/(\varepsilon \lg n))$ bits. Thus the total space cost is $O(n/\varepsilon)$ bits.

The time complexity of the binary search is different from a typical binary search. The number of values of k in the entire range is $O(\lg_{1+\varepsilon} n)$, so the complexity of the binary search is $O(\lg(\lg_{1+\varepsilon} n)) = O(\lg\left(\frac{\lg n}{\lg(1+\varepsilon)}\right)) = O(\lg\left(\frac{\lg n}{\varepsilon}\right)) = O(\lg \lg n + \lg(1/\varepsilon))$.

Lemma 71. *There exists an $O(n/\varepsilon)$-bit data structure that supports one-dimensional $(1+\varepsilon)$-approximate range mode queries in $O(\lg \lg n + \lg(1/\varepsilon))$ time.*

High Frequencies: $O(n/\varepsilon)$-Bits $O(\lg(1/\varepsilon))$ Query Time. The bottleneck of the approach described in the previous section is the binary search on k. To speed up queries, we store an additional data structure that uses $O(n)$ bits but returns a 4-approximate range mode.

Lemma 72. *There exists an $O(n)$-bit data structure that supports one-dimensional approximate range mode queries in constant time with approximation factor* 4.

Proof. We assume n is a power of 2. We construct a network of fusion trees [98]. At the top level, we store two fusion trees $\mathcal{F}_{n/2,l}$ and $\mathcal{F}_{n/2,r}$. The tree $\mathcal{F}_{n/2,l}$ contains the values $e_1, \ldots, e_{\lg n}$, where e_j is the largest index satisfying $F(c_{e_j..n/2}) = 2^j$. $\mathcal{F}_{n/2,r}$ contains the values $e_1, \ldots, e_{\lg n}$, where e_j is the smallest index satisfying $F(c_{n/2..e_j}) = 2^j$. If a query crosses the middle index $n/2$, we query $\mathcal{F}_{n/2,l}$ to get p_1, the smallest value greater than or equal to a, and we query $\mathcal{F}_{k,r}$ to get p_2, the largest value less than or equal to b. We return p_1 if $F(c_{p_1..n/2}) > F(c_{n/2..p_2})$ and p_2 otherwise. Clearly, p_1 is a 2-approximate mode for $c_{a..n/2}$ and p_2 is a 2-approximate mode for $c_{n/2..b}$. The true mode has at least half its occurrences in one of these regions, so the value we return is a 4-approximate mode for $c_{a..b}$.

If the query does not cross the middle, it falls entirely in one of the two sides. We may therefore repeat our fusion tree scheme in a divide and conquer fashion, recursing on the two halves. Eventually, there will be a level of the recursion that intersects the query.

To analyze the total space used, we use the recurrence $S(n) = 2S(n/2) + O(\lg^2 n)$, which solves to $S(n) = O(n)$ bits.

To analyze the time complexity of the query, observe that the fusion trees on $O(\lg n)$ elements with word size $O(\lg n)$ support the necessary predecessor/ successor queries in constant time. However, we must know which fusion trees to query. This involves finding the level of recursion in which the query range intersects the midpoint. This is equivalent to the highest set bit in the XOR of a and b, which can be determined in constant time in the word RAM model. With this information, we can do the necessary arithmetic to find the appropriate fusion trees to query, and thus query takes constant time. □

We now return to the $(1+\varepsilon)$-approximation. To answer a query $c_{a..b}$, we first query the 4-approximation structure of Lemma 72, which returns a corresponding frequency x. We now know $x \leq F(c_a, \ldots, c_b) \leq 4x$. We have thus shrunk the number of values of k from Lemma 70 that need be tested for the $(1+\varepsilon)$-approximation from $\lceil \lg_{1+\varepsilon} n \rceil$ to $\lceil \lg_{1+\varepsilon}(4x/x) \rceil = \lceil \lg_{1+\varepsilon} 4 \rceil$. Thus our binary search now takes time $O(\lg \left(\frac{2}{\lg(1+\varepsilon)}\right)) = O(\lg(1/\varepsilon))$.

Theorem 73. *There exists an $O(n/\varepsilon)$-bit data structure that supports one-dimensional approximate range mode queries in $O(\lg(1/\varepsilon))$ time with approximation factor $1+\varepsilon$.*

6.4 Dynamic Approximate Range Mode

In this section we consider the dynamic variant of the approximate range mode problem. We maintain our sequence $c_{a..b}$ under insertions and deletions, so that for an arbitrary query range $c_{a..b}$ an approximate range mode can be found efficiently.

The high-level approach is as follows. Similar to Section 6.3, for each $j \leq \lg_{1+\varepsilon} n$, our goal is to maintain a set of intervals $\mathcal{I}_j$ such that the mode of a query range $c_{a..b}$ occurs more than $(1+\varepsilon)^j$ times if and only if $c_{a..b}$ contains an interval in $\mathcal{I}_j$. Then, for all j and each interval $c_{l..r}$ in $\mathcal{I}_j$ we maintain the points (l, r, j) in a data structure $\mathcal{D}$ that supports the following range queries: given a query point (a, b), return the highest j such that $a \leq l$ and $r \leq b$ for at least one point (l, r, j) in $\mathcal{D}$.

However, unlike the sets of intervals maintained in Section 6.3, our construction in this section satisfies the property that a single update affects only a *small* number of intervals in the sets $\mathcal{I}_j$ for all j. We now proceed with the technical argument.

Let S_x denote the set of positions of the element x in the sequence $c_{1..n}$. We will denote by $S_x[i]$ the position of the i^{th} occurrence of element x. Let $I_x(l, r)$ denote the interval $c_{S_x[l]..S_x[r]}$.

Now let $\delta = 1 + \varepsilon' = (1+\varepsilon)^{1/3}$ and fix x. There are $f = F_x(c_{1..n})$ occurrences of element x in the full range $c_{1..n}$. We will maintain a subset of the $\binom{f}{2}$ possible intervals $I_x(l, r)$ in sets $\mathcal{I}_{j,x}$, $1 \leq j \leq \lceil \lg_\delta n \rceil$. We will not have need for nested intervals in $\mathcal{I}_{j,x}$; therefore, we can number each interval of $\mathcal{I}_{j,x}$ from left to right with s_k the start of interval k and e_k the end of interval k, satisfying $s_k \leq s_{k+1}$, $e_k \leq e_{k+1}$. We maintain the following two invariants on the intervals of $\mathcal{I}_{j,x}$: (1) $\delta^j \leq e_k - s_k \leq \delta^{j+1}$, and (2) $(\varepsilon'/2)\delta^j \leq s_{k+1} - s_k \leq \varepsilon'\delta^j$, and the number of positions of S_x not covered by an interval of $\mathcal{I}_{j,x}$ at either end is at most $(\varepsilon'/2)\delta^j$ (so $s_1 \leq (\varepsilon'/2)\delta^j$ and $f - r_{|\mathcal{I}_{j,x}|} \leq (\varepsilon'/2)\delta^j$). From our invariants we get the following proposition.

Proposition 74. *An interval $I_x(l, r)$ intersects at most $2(r-l+1)/(\varepsilon'\delta^j) + O(1/\varepsilon')$ intervals of $\mathcal{I}_{j,x}$.*

Proof. By Invariant (2), we have a gap size of between $(\varepsilon'/2)\delta^j$ and $\varepsilon'\delta^j$ elements between consecutive starting points of intervals of $\mathcal{I}_{j,x}$. Since each interval has size at most δ^{j+1}, the total number of intervals intersecting $I_x(l, r)$ is at most $2(r - l + 1 + 2\delta^{j+1})/(\varepsilon'\delta^j)$. □

For each interval $I_x(s_k, e_k)$ of $\mathcal{I}_{j,x}$, let $\texttt{pot}(I_x(s_k, e_k)) = e_k - s_k + 1$ denote the number of elements of S_x (and thus positions in the original sequence $c_{1..n}$) that fall between s_k and e_k. When we insert or delete an element x, by Proposition 74, we must update the pot values of $O(1/\varepsilon')$ intervals of $\mathcal{I}_{j,x}$. Across all j, $1 \leq j \leq \lceil \log_\delta n \rceil$, $O((1/\varepsilon') \lg_\delta n)$ intervals are affected.

During the updates, each affected $\texttt{pot}(I_x(s_k, e_k))$ value is incremented or decremented by one. If, for an interval $I_x(s_k, e_k)$ in $\mathcal{I}_{j,x}$, Invariant (1) is violated by the update, then we rearrange the intervals in the neighborhood of $I_x(s_k, e_k)$ as follows. Consider all intervals of $\mathcal{I}_{j,x}$ that intersect with $I_x(s_k, e_{k+1})$. By Proposition 74, there are $O(1/\varepsilon')$ such intervals. We remove said intervals and create new intervals in their place with exactly $\lceil (1+\varepsilon'/2)\delta^j \rceil$ positions of x that fall in each interval. Furthermore, we space them so that Invariant (2) holds when the new intervals are inserted into $\mathcal{I}_{j,x}$.

To build these intervals, we must be able to efficiently search for elements by rank in S_x. As this will not dominate the update cost, we can use a typical order statistic tree, with $O(\lg f) = O(\lg n)$ query and update time. We may construct the new intervals satisfying invariants (1) and (2) with a constant number of queries on S_x per interval, thus in $O((1/\varepsilon') \lg n)$ time overall.

We can analyze the total cost of rebuilds as follows. On each update, we affect $O(1/\varepsilon')$ intervals at each level. However, the affect on $\texttt{pot}$ is the same for each interval, and when we rebuild, we rebuild a superset of the intervals affected on update. It follows that the total amortized cost of rebuilds is $\sum_{j=1}^{\lceil \log_\delta n \rceil} (1/\delta^j) \cdot O((1/\varepsilon') \lg n) = O((\lg n)/\varepsilon'^2)$ per update.

Further, in each update we must update S_x and update the $\texttt{pot}$ values. These take time $O(\lg n)$ and $O((1/\varepsilon') \lg_\delta n) = O(\lg n/\varepsilon'^2)$, respectively. So far we pay $O(\lg n/\varepsilon'^2)$ per update, but we have yet to describe the data structure that holds each $\mathcal{I}_{j,x}$, which will also need to be updated during rebuilds.

Consider each interval $I_x(s_k, e_k)$ of $\mathcal{I}_{j,x}$ as a point (s_k, e_k, j). We store each interval of $\mathcal{I}_{j,x}$, across all $1 \le j \le \lceil \lg_\delta n \rceil$ and all x, in a data structure $\mathcal{D}$ that supports the following range queries: given a query point (a, b), return the highest j such that $a \le l$ and $r \le b$ for at least one point (l, r, j) in $\mathcal{D}$. Associated with each point, we keep the element x from which it originated.

We first must consider the number of intervals (and thus points) stored in $\mathcal{D}$. As before, we assume element x occurs $f = F_x(c_{1..n})$ times in $c_{1..n}$. Then $|\mathcal{I}_{j,x}| = O(f/\lceil \varepsilon'\delta^j \rceil)$. Across all levels, we can bound the total number of intervals at $O(f \lg_\delta n) = O(f \lg n/\varepsilon')$ or $O(f/\varepsilon'^2)$. Accounting for all x, the number of intervals in $\mathcal{D}$ will be $O(m) = O(\min(n \lg n/\varepsilon', n/\varepsilon'^2))$.

Lemma 75. *Data structure $\mathcal{D}$ can be stored in $O(n \lg n)$ bits, where n is the number of elements in $\mathcal{D}$. Queries and updates can be supported in $O(\lg n/\lg\lg n)$ time.*

Proof. Let P denote the set of points to be stored in our data structure. Here we use $\varepsilon > 0$ independently of the rest of the section. We start by considering the special case when the second coordinate is bounded by $\lg^\varepsilon n$, i.e., $r \le \lg^\varepsilon n$ for all $(l, r, j) \in P$. In this case it is sufficient to store $\lg n$ points for every possible value of b: let $\max_{r,j}$ denote the biggest first coordinate of a point (l', r', j') in P with $r' = r$, $j' = j$ ($\max_{r,j} = \max\{ l' \,|\, (l', r', j') \in$

P and $r' = r, j' = j\}$. The answer to a query (a, b) is the largest j that satisfies $a \le \max_{r,j}$ for some $r \le b$. We keep all values $\max_{r,j}$ such that P contains at least one point (l, r, j) for some l, and store them in increasing order. We group them in blocks of size $\Theta(\lg^{1-\varepsilon} n)$ and we keep a global lookup table of size $o(n)$ bits that allows answering queries within any possible block.

Also, in a local lookup table of size $O(\lg^{3\varepsilon} n)$ bits we store for each block and every possible value of r the index of the block preceding it which maximizes the value of j given r. We also store a fusion tree on the values $\max_{r,j}$ so that we can compute the rank of a within these values in constant time. Given a query, we compute in constant time the block which the predecessor of a belongs to and use table lookup on that block and one other block preceding it to get the answer. Updates also take constant time since the size of individual blocks and the local lookup table fit in a single word.

A general query can be reduced to the above described special case by using a range tree with node degree $\lg^{\varepsilon} n$ that splits the points on the value of their second coordinate. Although every point is stored in $O(\lg n / \lg\lg n)$ nodes, our data structure uses linear space. Let $P(u)$ denote the set of points stored in a node u. We replace the second coordinate of each point $p \in P(u)$ with the index i of the child node u_i such that $p \in P(u_i)$. We keep the above described special case data structure in every node $P(u)$, but we do not store the set $P(u)$ itself. A query interval can be fully covered by $O(\lg n / \lg\lg n)$ tree nodes. We query the data structure in each one of them and return the maximum value j in $O(\lg n / \lg\lg n)$ time. Similarly, an update affects the special case data structure in $O(\lg n / \lg\lg n)$ nodes and requires $O(\lg n / \lg\lg n)$ time.

The total space usage is $O(n \log n)$ bits because we spend $O(\min(\log^{2+\varepsilon} n, |P(u)| \lg n)$ bits in each node u of the range tree. To prove this bound, we classify nodes into low and high nodes. Low nodes are the nodes in the lowest $(1 + 2/\varepsilon)$ levels of the tree and the rest of the nodes are high nodes. We also store the set of points $P(u)$ in every low node u. Thus we spend $O(|P(u)| \lg n)$ bits in every low node, so the total space consumed by all low nodes is $O((1/\varepsilon) n \lg n)$ bits. We spend $O(\lg^{2+\varepsilon} n) = O(|P(u)|)$ bits in every high node because $|P(u)| \ge \lg^{2+\varepsilon} n$. Since the total number of points in all $P(u)$ is $O(n(\lg n / \lg\lg n))$, the total space consumed by high nodes is $O(n(\lg n / \lg\lg n))$ bits. □

Now suppose we are given a query range $c_{a..b}$. We find the largest j such that some interval from $\mathcal{I}_{j,x}$ for some x is contained in $c_{a..b}$. Using data structure $\mathcal{D}$ from Lemma 75, we can compute the index j in $O(\lg n / \lg\lg n)$ time. We return the element x associated with j.

Lemma 76. *The element x returned is a $(1 + \varepsilon)$-approximate mode of query range $c_{a..b}$.*

Proof. If $c_{a..b}$ contains an interval from $\mathcal{I}_{j,x}$, then x occurs at least δ^j times in $c_{a..b}$. On the other hand, we can show that if some y occurs δ^{j+3} times in $c_{a..b}$, then $c_{a..b}$ contains an interval from $\mathcal{I}_{j+1,y}$. Recall $\delta^{j+3} = (1+\varepsilon')\delta^{j+2}$. Each interval of $\mathcal{I}_{j+1,y}$ has size at most δ^{j+2} and there is a gap of at most $\varepsilon'\delta^j$ elements of y between the start of every interval in $\mathcal{I}_{j+1,y}$. Then since $\varepsilon'\delta^{j+1} + \delta^{j+2} < (1+\varepsilon')\delta^{j+2}$, it must be that an interval of $\mathcal{I}_{j+1,y}$ falls in the query range $c_{a..b}$. We therefore know $\delta^j \le F(c_{a..b}) < \delta^{j+3} = (1+\epsilon)\delta^j$. It follows that x is a $(1+\varepsilon)$-approximate mode of query range $c_{a..b}$. □

This gives us the main theorem for the section.

Theorem 77. *There exists an $O(m \lg m)$-bit data structure, where $m = \min(n \lg n/\varepsilon, n/\varepsilon^2)$ that answers $(1+\varepsilon)$-approximate range mode queries in $O(\lg m/\lg\lg m)$ time. Insertions and deletions are supported in $O(\lg n/\varepsilon^2)$ time.*

Proof. We have $(1+\varepsilon')^3 = (1+\varepsilon)$ and $(1+\varepsilon)^3 = \varepsilon^3 + 3\varepsilon^2 + 3\varepsilon + 1$. The smallest exponent dominates $O(1/\varepsilon')$ since $\varepsilon \le 1$ and thus $\varepsilon' < \varepsilon \le 1$. Thus we have $1/\varepsilon' = O(1/\varepsilon)$. As previously stated, the number of intervals in $\mathcal{D}$ is $O(m)$, where $m = \min(n \lg n/\varepsilon, n/\varepsilon^2)$. The space bound for $\mathcal{D}$ is thus $O(m \lg m) = \Omega(n \lg n)$ bits, which dominates the total space cost. The query time is $O(\lg m/\lg\lg m)$.

The update cost has four components: Updating $\mathcal{D}$, updating S_α, and updating `pot` values for all affected intervals. As previously mentioned, the latter three are dominated by $O(\lg n/\varepsilon'^2) = O(\lg n/\varepsilon^2)$. Via Lemma 75, the cost of updating $\mathcal{D}$ is $O(\lg m/\lg\lg m)$. Since m is no more than n/ε^2, $\lg m/\lg\lg m$ is dominated by $O(\lg n/\varepsilon^2)$. In total, the cost of updates is $O(\lg n/\varepsilon^2)$. □

We can use our dynamic data structure to obtain a result for approximate range mode queries on two-dimensional points. Our data structure can find approximate mode in the case when the query range is bounded on three sides.

Corollary 78. *There exists a data structure that supports three-sided two-dimensional approximate range mode queries in $O(\log m)$ time and uses $O(m \log m)$ words of $\log n$ bits, where $m = \min(n \lg n/\varepsilon, n/\varepsilon^2)$.*

Proof. Using the technique introduced by Dietz in [70], we can transform a data structure that supports updates in $u(n)$ time and queries in $q(n)$ time into an offline partially persistent data structure that answers queries in $O(q(n) \cdot \log\log n)$ time and uses $O(n \cdot u(n))$ words of space. Using sweep line technique, we can transform an offline partially persistent data structure for one-dimensional queries into a static data structure for three-sided queries with the same time and space bounds. □

6.5 Approximate Range Median and Range Selection

In this section we present solutions to approximate range selection queries. As discussed previously, a range selection query takes two indices a, b of a sequence $c_1, \ldots, c_n$ and must return the index of an element x whose rank in $c_{a..b}$ is between $k - \alpha(b - a + 1)$ and $k + \alpha(b - a + 1)$. We study two variants. In the first variant, the rank k is supplied prior to construction of the data structure. In the second variant, we allow k to be specified at query time. Here rank is defined so the ith smallest element in the range has rank i. We also support a specific k depending on the size of the range, i.e. $f(b-a+1) = \lceil (b-a+1)/2 \rceil$, which is the range median problem. We make the restrictions $f(x) \leq f(x+1) \leq f(x) + 1$ and $1 \leq f(x) \leq x$.

Theorem 79. *Any one-dimensional approximate range median data structure requires $\Omega(n)$ bits.*

Proof. Assume n is even. Divide the sequence S of size n to $n/2$ blocks each of size 2. We say that S satisfies property $(*)$ if for each block b in S one of the following two conditions hold:

- either b consists of $\{1, 2\}$,
- or b consists of $\{2, 1\}$.

Clearly, the number of sequences that satisfy $(*)$ is $2^{(n/2)}$ since there exists $n/2$ blocks in a sequence of size n and each block can have one of two different values. Moreover for any two distinct sequences S_1 and S_2 satisfying $(*)$ differing at block b, the approximate range selection query must be exact on block b, and therefore must return different values. Thus, the information theoretic lower bound for storing an approximate range median data structure is $\Omega(\lg 2^{(n/2)}) = \Omega(n/2) = \Omega(n)$ bits. □

Fixed Rank $f(b - a + 1) = k$ Selection. We first address the range selection variant with a fixed rank $f(b - a + 1) = k$. We use a similar approach to the one in Lemma 72. We again assume n is a power of 2. At the top level, we store values $m_{n/2,i,j}$ $(1 \leq i, j \leq \lceil \lg_{1+\alpha} n \rceil)$. Let $r_i = n/2 - (1 + \alpha)^i$ and $s_j = n/2 + 1 + (1 + \alpha)^j$. Then $m_{n/2,i,j}$ is the element of rank $f(s_j - r_i + 1)$ in the range $c_{\lfloor r_i \rfloor..\lceil s_j \rceil}$. We then build the structure recursively on the left and right halves of the full range.

Given a query range $c_{a..b}$, we find the appropriate element $m_{t,i,j}$ where $a \leq t$, $t + 1 \leq b$, and i and j are largest possible satisfying $a \leq r_i$ and $s_j \leq b$. We return $m_{t,i,j}$.

Lemma 80. *The above data structure returns an α-approximate fixed-rank k element of any query range $c_{a..b}$.*

Proof. Let $x = r_i - a$ and $y = b - s_j$. Consider the size of x. If we let $z = (1+\alpha)^i$, then $x + z < (1+\alpha)z$. It follows $x < \alpha z$. Since $z \leq (t - a + 1)$, and applying similarly for y, we can show $x < \alpha(t - a + 1)$ and $y < \alpha(b - t)$. The elements in the ranges represented by x and y shift the true rank k element of $c_{a..b}$ at most $x + y < \alpha(b - a + 1)$ ranks from $m_{t,i,j}$. It follows that $m_{t,i,j}$ is an α-approximate rank k element for range $s_a, \ldots, s_b$. □

As for Theorem 72, to find the level to query, we find the highest set bit of a XOR b, then find the appropriate index $m_{t,i,j}$ via arithmetic. In total, the query takes constant time.

We now analyze the space required. At the top level, we use $O(\lg^2_{1+\alpha}(n) \cdot \lg n)$ bits, which is equal to $O(\frac{\lg^3 n}{\lg^2(1+\alpha)}) = O(\lg^3 n/\alpha^2)$ bits. Therefore our recurrence is $S(n) = 2S(n/2) + O(\lg^3 n/\alpha^2)$. The recursion tree is leaf-heavy, with total space amounting to $O(n/\alpha^2)$ bits.

Theorem 81. *There exists an $O(n/\alpha^2)$-bit data structure that supports one-dimensional α-approximate fixed-rank $f(b - a + 1) = k$ selection queries in constant time.*

Online Rank k Selection. Our data structure from the previous section can be adapted to support queries that specify the rank k at query time. We again assume n is a power of 2. Let $\delta = 1 + \alpha/2$. At the top level we now store values $m_{n/2,i,j,l}$ $(1 \leq i, j, \leq \lceil \lg_\delta n \rceil$, $0 \leq l \leq \lfloor 1/\alpha \rfloor)$. Again, we let $r_i = n/2 - \delta^i$ and $s_j = n/2 + 1 + \delta^j$. However, this time, $m_{n/2,i,j,l}$ represents the element of rank $q_l = l\alpha \cdot (s_j - r_i + 1) + 1$ in $c_{r_i..s_j}$. As q_l may be fractional, for simplicity we just store both rank $\lfloor q_l \rfloor$ and $\lceil q_l \rceil$ elements. We build this structure recursively on both halves of the full range.

Given a query $s_{a..b}$, we again find the appropriate element $m_{t,i,j,l}$ where $a \leq t$, $t + 1 \leq b$, i and j are largest possible satisfying $a \leq r_i$ and $s_j \leq b$, and l is chosen so q_l is as close to k as possible. We return $m_{t,i,j,l}$.

Lemma 82. *The above data structure returns an α-approximate rank k element of any query range $c_{a..b}$ and specified rank k.*

Proof. Again let $x = r_i - a$ and $y = b - s_j$. For the same reasons as in the proof of Lemma 80, we have $x + y < \alpha(b - a + 1)/2$.

There are no more than $\alpha \cdot (s_j - r_i + 1) \leq \alpha \cdot (b - a + 1)$ ranks between each consecutive q_l and q_{l+1}. Thus our chosen q_l satisfies $|q_l - k| < \lfloor \alpha(b - a + 1)/2 \rfloor$. It follows that $m_{t,i,j,l}$ is no more than $\alpha \cdot (b - a + 1)$ ranks away from the true rank k element in range $c_{a..b}$. □

The query time follows as in the previous section. However, we must account for the additional space usage. Our recurrence is now $T(n) = 2T(n/2) + O(\lg^3 n/\alpha^3)$, from the additional $1/\alpha$ factor in the space cost at each level. This totals to $O(n/\alpha^3)$ bits.

Theorem 83. *There exists an $O(n/\alpha^3)$-bit data structure that supports one dimensional α-approximate online rank k selection queries in constant time.*

Part III

Graph and Dynamic Graph Algorithms

Chapter 7

Offline Dynamic Higher Connectivity

7.1 Introduction

Dynamic graph data structures seek to answer queries on a graph as it undergoes edge insertions and deletions. Perhaps the simplest and most fundamental query to consider is connectivity. A connectivity query asks for the existence of a path connecting two vertices u and v in the current graph. As the insertion or deletion of a single edge may have large consequences to connectivity across the entire graph, constructing an efficient dynamic data structure to answer connectivity queries has been a challenge to the data structure community. A number of solutions have been developed, achieving a wide variety of runtime tradeoffs in a number of different models [84, 85, 86, 148, 137, 138, 141, 149, 172, 226, 241]

The models typically addressed are *online*: each query must be answered before the next query or update is given. A less demanding variant is the *offline* setting, where the entire sequence of updates and queries is provided as input to the algorithm. While an online data structure is more general, there are many scenarios in which the entire sequence of operations is known in advance. This is often the case when a data structure is used in a subroutine of an algorithm [170, 40], one specific example being the use of dynamic trees in the near-linear time minimum cut algorithm of Karger [153].

In exchange for the loss of flexibility, one can hope to obtain faster and simpler algorithms in the offline setting. This has been shown to be the case in the dynamic minimum spanning tree problem. While an online fully-dynamic minimum spanning tree data structure requires about $O(\log^4 n)$ time per update [139], the offline algorithm of Eppstein requires only $O(\log n)$ time per update [84].

In this chapter, we show similar, but stronger, performance gains for higher versions of connectivity. In particular, we consider the problems of $2, 3$-edge/vertex connectivity on a fully-dynamic undirected graph. An extension of connectivity, c-edge connectivity asks for the existence of c edge-disjoint paths between two vertices u and v in the current graph. Vertex connectivity requires vertex-disjoint paths instead of edge-disjoint paths. Current online fully-dynamic 2-edge/vertex connectivity data structures require update time $\tilde{O}(\log^2 n)$[1] [138] and $\tilde{O}(\log^3 n)$[2] [226], respectively, and current online fully-dynamic 3-edge/vertex connectivity data structures require update time $O(n^{2/3})$ and $O(n)$, respectively [85]. In contrast, our offline algorithms for $2, 3$-edge/vertex connectivity require only $O(\log n)$ time per operation. As the lower bound on dynamic connectivity [197], as well as most lower bounds in general [3, 2, 1, 65], also apply in the offline model, our algorithms are optimal up to a constant factor. Our results further show that any lower bound attempting to show hardness stronger than $\Omega(\log n)$ time per operation for online fully-dynamic $2, 3$-edge/vertex connectivity must make use of the online model.

As a straightforward application of our work, one can consider the use of our algorithms when data regarding a dynamic network is collected, but not analyzed, until a later point in time. For example, to diagnose an issue of network latency across key routing hubs, or determine viability of a dynamically-changing network of roads, our algorithms can answer a batch of queries in time $O(t \log n)$, where t is the total number of updates and queries. Since online fully-dynamic algorithms for higher versions of connectivity are significantly slower, namely, $O(n^{2/3})$ and $O(n)$ time update for 3-edge and 3-vertex connectivity, respectively, our offline data structure makes these computations practical for large data sets when they would otherwise be prohibitively expensive.

Related to our work are papers by Łącki and Sankowski [172] and Karczmarz and Łącki [149], which also apply to the above applications but for lower versions of connectivity. Their work considers a fixed sequence of graph updates, given in advance, and is then able to answer connectivity queries regarding intervals of this sequence, online. This is more general than the model we consider because the queries need not be supplied in advance and data regarding an interval of time is richer than information from a specific point in time. For connectivity and 2-edge connectivity, Karczmarz and Łącki achieve $O(\log n)$ time per operation [149]. Both 2-vertex connectivity and 3-edge/vertex connectivity queries are not supported.

In competitive programming, the idea of using divide and conquer as an offline algorithm for connectivity is known. Several contest problems[3] have appeared that are solved

[1]The $\tilde{O}(\cdot)$ notation hides $\log\log n$ factors.

[2]This complexity is claimed in Thorup's STOC 2000 [226] result. As noted by Huang et al. [141], the paper provides few details, deferring to a journal version that has since not appeared. The best complexity for online fully-dynamic biconnectivity prior to this claim was $O(\log^5 n)$ by Holm and Thorup [137].

[3] See `https://codeforces.com/blog/entry/15296` and `https://codeforces.com/gym/100551/problem/A`, for example.

with similar techniques to Eppstein's minimum spanning tree algorithm [84], the approach we adopt here. The master's thesis of Sergey Kopeliovich, a member of the competitive programming community, describes such an offline algorithm for fully-dynamic 2-edge connectivity, also achieving about $O(\log n)$ time per operation [163]. Unfortunately, the thesis only appears in Russian, but we speculate that the ideas used are similar.

The techniques developed in this chapter may be of independent interest. Our work has close connections with recent developments in vertex sparsification, particularly vertex sparsification-based dynamic graph data structures [4, 76, 118, 119, 75, 86, 120, 166, 14, 89, 88]. In particular, the equivalent graphs at the core of our algorithms are akin to vertex sparsifiers, with the main difference that 2- and 3-connectivity require preserving far less information than the more general definitions of vertex sparsifiers [118, 166]. A promising step in this direction is very recent work of Goranci et al. [120], which suggests the notion of a *local sparsifier*. This is a generalization of the sparsifier that we consider here, and leads to efficient incremental algorithms in the *online* setting.

Indeed, offline algorithms haven proven useful for the development of online counterparts in the past. One such example is recent development in the maintenance of dynamic effective resistance. Recent work in fully-dynamic data structures for maintaining effective resistances online [75] relied heavily upon ideas from earlier data structures for maintaining effective resistances in offline [76, 171] or offline-online hybrid [76, 170] settings.

This work was first published online in the open access journal arXiv [202] and has recently been extended to offline 4- and 5-edge connectivity [179]. This new work achieves about $O(\sqrt{n})$ time complexity per operation.

The rest of this chapter will be dedicated to proving the following theorem:

Theorem 84. *Given a sequence of t updates/queries on a graph of the form:*

- *Insert edge (u, v),*
- *Delete edge (u, v),*
- *Query if a pair of vertices u and v are 2-edge connected/3-edge connected/bi-connected/triconnected in the current graph,*

there exists an algorithm that answers all queries in $O(t \log n)$ time.

For simplicity, we will assume the graph is empty at the start and end of the sequence, but the results discussed are easily modified to start with an initial graph G, at the cost of an additive $O(m)$ term in the running time, where m is the number of edges of G. Further, we assume a fixed vertex set of size n. Any update sequence with arbitrary vertex endpoints can be modified to one on a fixed set of vertices, where the size of the fixed set is equal to the largest number of non-isolated vertices in any graph achieved in the given update sequence. Finally, we consider the graph G to be a multigraph, since at times during our constructions and definitions, we will need to work with multigraphs.

The chapter is organized as follows. We describe our offline framework for reducing graphs to smaller equivalents in Section 7.2. We show how simple techniques can be used to create such equivalents for 2-edge connectivity and bi-connectivity in Section 7.3. In Section 7.4, we extend these constructions to 3-edge connectivity. Our most technical section is Section 7.5, where constructing equivalent graphs for 3-vertex connectivity requires careful manipulation of SPQR trees.

7.2 Offline Framework

The main idea of our offline framework is to perform divide and conquer on the input sequence, similar to what is done in Eppstein's offline minimum spanning tree algorithm [84]. Consider the full sequence of updates and queries $x_1, \ldots, x_t$, where each x_i is either an edge insertion, edge deletion, or query. Call each x_i an event.

Assume each inserted edge has a unique identity. Then for each inserted edge e, we may associate an interval $[I(e), D(e)]$, indicating that edge e was inserted at time $I(e)$ and removed at time $D(e)$. Plotting time along the x-axis and edges on the y-axis as in Figure 7.1 gives a convenient way to view the sequence of events.

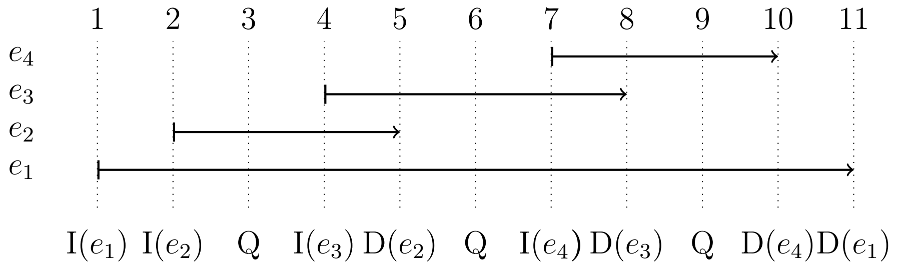

Figure 7.1: A timeline diagram of four edge insertions(I)/deletions(D) and three queries(Q), with time on the x-axis and edges on the y-axis.

Fix some subinterval $[l, r]$ of the sequence of events. Let us classify all edges present at any point of time in the sequence $x_l, \ldots, x_r$ as one of two types.

1. Edges present throughout the duration ($i_e \leq l \leq r \leq d_e$), we call *permanent* edges.

2. Edges affected by an event in this range (one or both of $I(e), D(e)$ is in (l, r)), we call *non-permanent* edges.

While there may be a large number of permanent edges, the number of non-permanent edges is limited by the number of time steps, $r - l + 1$. Therefore, the graph can be viewed as a large static graph on which a smaller number of events take place.

Our goal will be to reduce this graph of permanent edges to one whose size is a small function of the number of events in the subinterval. If we may do so without affecting the answers to the queries, we can recursively apply the technique to achieve an efficient divide and conquer algorithm for the original dynamic c-connectivity sequence.

We will work in the dual-view, considering cuts instead of edge-disjoint or vertex-disjoint paths. Two vertices u and v are c-edge connected if there does not exist a cut of $c-1$ edges separating them; further, u and v are c-vertex connected if there does not exist a cut of $c - 1$ vertices that separates them.

We need the following definition.

Definition 85. *Given a graph $G = (V_G, E_G)$ with vertex subset $W \subseteq V_G$ and a graph $H = (V_H, E_H)$ with $W \subseteq V_H$, we say that H and G are c-edge equivalent if, for any partition (A, B) of W, the size of a minimum cut separating A and B is the same in G and H whenever either of these sizes is less than c. Similarly, we say H and G are c-vertex equivalent if, for any partition (A, B, C) of W with $|C| < c$, the size of a minimum vertex cut D separating A and B such that $C \subseteq D$ and $D \cap A = \varnothing$, $D \cap B = \varnothing$, is the same in G and H whenever either of these sizes is less than c.*

This gives the following.

Lemma 86. *Suppose $G = (V_G, E_G)$ and $H = (V_H, E_H)$ are c-edge/c-vertex equivalent on vertex set W. Let E_W denote any set of edges between vertices of W. Then $H' = (V_H, E_H \cup E_W)$[4] and $G' = (V_G, E_G \cup E_W)$ are c-edge/c-vertex equivalent.*

Proof. We first show c-edge equivalence. Let (A, B) be any partition of W and consider the minimum cuts separating A and B in G' and H'. Since the edges in E_W are between vertices of W, they must cross the separation (A, B) in the same way. Therefore, if the minimum cut separating A and B had size less than c in either G or H, the minimum cuts separating A and B will have equivalent size in G' and H'. Further, if the minimum cuts separating A and B had size larger or equal to c in both G and H, the minimum cuts separating A and B will also have size larger or equal to c in G' and H', since we only add edges to G' and H'. Thus G' and H' are c-edge equivalent.

[4]We take $\cup$ here to be in the multigraph sense; an edge $uv \in E_W$ is added regardless if there is already a uv edge in E_H or E_G

We now consider c-vertex equivalence. Consider a partition (A, B, C) of W. As with edge connectivity, if no vertex subset D exists satisfying the conditions of Definition 85, the introduction of additional edges between any vertices of W will not change the existence of such a set D in G' or H'. Furthermore, if an edge of E_W connects a vertex of A to a vertex of B, no vertex cut separates A and B in G' and H'. Now suppose none of these cases is true, and there exists a vertex set D satisfying the conditions of Definition 85 such that the removal of D disconnects A and B in G and H and further that no edge of E_W connects a vertex of A to a vertex of B. Then the removal of vertex set D still disconnects A and B in G' and H'. Thus c-vertex equivalence of G' and H' follows from c-vertex equivalence of G and H. □

Now consider the graph G of permanent edges for the subinterval $x_l, \ldots, x_r$ of events. Let W be the set of vertices involved in any event in the subinterval (that is, W is the set of endpoints of all non-permanent edges in the subinterval, as well as vertices involved in a query). We will refer to these vertices as *active* vertices, and all other vertices of G not in W as *inactive* vertices. Lemma 86 says that if we reduce G to a c-edge/ c-vertex equivalent graph H on set W, the result of all queries in $x_l, \ldots, x_r$ on H will be the same as on G. This is because all cuts in H and G that affect the queries (therefore of size less than c) are of equivalent size, even after the addition of non-permanent edges in H and G.

This idea can lead to a divide and conquer algorithm if we can produce such equivalent graphs H of small size efficiently. Specifically:

Lemma 87. *Given a graph G with m edges and vertex set W of size k, if there is an $O(m)$ time algorithm that produces a graph H of size $O(k)$ that is c-edge/c-vertex equivalent to G on W, then there is an algorithm that can answer all c-edge/c-vertex connectivity queries in a sequence of events $x_1, \ldots, x_t$ in $O(t \log n)$ time.*

Proof. We perform divide and conquer on the sequence of events. We take the sequence of events $x_1, \ldots, x_t$ and divide it in half. Over each half, we will take the graph of permanent edges, which we denote G, and reduce it to a c-edge/ c-vertex equivalent graph H. We repeat the scheme recursively. As the subintervals get smaller, non-permanent edges become permanent and are absorbed by the production of equivalent graphs. Eventually, we reduce to subintervals with a constant number of events, which can be answered by any algorithm of our choice on a graph of constant size.

Consider the sizes of the graphs in each step of recursion. The graph G is the graph produced in the previous level plus the edges that became permanent in this interval. The graph produced at the previous level has size linear in the number of events at the current level, and since we reduce the number of events by a factor 2 in each step of recursion, the number of edges that become permanent is also linear. It follows that the divide and conquer satisfies the recurrence $T(t) = 2T(t/2) + O(t)$, which solves to $T(t) = O(t \log t)$. If t is polynomially-bounded by n, $T(t) = O(t \log n)$. If not, we may first break the sequence of events $x_1, \dots x_t$ into blocks of size, say, n^2. Since the size of the graph G cannot be more than $O(n^2)$ in any subinterval, we can therefore handle each block separately and answer all queries in $O(t \log n)$ time. This proves the lemma. □

The remainder of the chapter will show the construction of 2-edge, 2-vertex, 3-edge, and 3-vertex equivalent graphs.

7.3 Equivalent Graphs for 2-Edge Connectivity and Biconnectivity

We now show offline algorithms for dynamic 2-edge connectivity and bi-connectivity by constructing 2-edge and 2-vertex equivalents needed by Lemma 87. These two properties ask for the existence of a single edge/vertex whose removal separates query vertices u and v. Since these cuts can affect at most one connected component, it suffices to handle each component separately.

The underlying structure for 2-edge connectivity and bi-connectivity is tree-like. This is perhaps more evident for 2-edge connectivity, where vertices on the same cycle belong to the same 2-edge connected component. We will first describe the reductions that we will make to this tree in Section 7.3.1, and adapt them to bi-connectivity in Section 7.3.2.

At times we will make use of the term "equivalent cut". By this we mean that a cut C' is equivalent to C if it has the same size and separates the vertices of W in the same way.

7.3.1 2-Edge Connectivity

Using depth-first search [140], we can identify all cut-edges in the graph and the 2-edge connected components that they partition the graph into. The case of edge cuts is slightly simpler conceptually, since we can combine vertices without introducing new cuts. Specifically, we show that each 2-edge connected component can be shrunk to single vertex.

Lemma 88. *Let S be a 2-edge connected component in G. Then contracting all vertices in S to a single vertex s in H*[5]*, and endpoints of edges correspondingly, creates a 2-edge equivalent graph.*

[5]Here we slightly abuse our requirement $W \subseteq V_H$, where V_H are the vertices of H. A map of W onto V_H that preserves the cuts needed by c-edge/c-vertex equivalence suffices.

Proof. The only cuts that we need to consider are ones that remove cut edges in G or H. Since we only contracted vertices in a component, there is a one-to-one mapping of these edges from G to H. Since S is 2-edge connected, all vertices in it will be on the same side of one of these cuts. Furthermore, removing the same edge in H leads to a cut with s instead. Therefore, all active vertices in S are mapped to s, and are therefore on the same side of the cut. □

This allows us to reduce G to a tree H, but the size of this tree can be much larger than k. Therefore we need to prune the tree by removing inactive leaves and length 2 paths whose middle vertex is inactive.

Lemma 89. *If G is a tree, the following two operations lead to 2-edge equivalent graphs H.*

- *Removing an inactive leaf.*
- *Removing an inactive vertex with degree 2 and adding an edge between its two neighbors.*

Proof. In the first case, the only cut in G that no longer exist in H is the one that removes the cut edge connecting the leaf with its unique neighbor. However, this places all active vertices in one component and thus does not separate W and need not be represented in H.

In the second case, if a cut removes either of the edges incident to the degree 2 vertex, removing the new edge creates an equivalent cut since the middle vertex is inactive. Also, for a cut that removes the new edge in H, removing either of the two original edges in G leads to an equivalent cut. □

This allows us to bound the size of the tree by the number of active vertices, and therefore finish the construction.

Lemma 90. *Given a graph G with m edges and k active vertices W, a 2-edge equivalent of G of size $O(k)$, H, can be constructed in $O(m)$ time.*

Proof. We can find all the cut edges and 2-edge connected components in $O(m)$ time using depth-first search [140], and reduce the resulting structure to a tree H using Lemma 88. On H, we repeatedly apply Lemma 89 to obtain H'.

In H', all leaves are active, and any inactive internal vertex has degree at least 3. Therefore the number of such vertices can be bounded by $O(k)$, giving a total size of $O(k)$. □

7.3.2 Bi-connectivity

All cut-vertices (articulation points) can also be identified using DFS, leading to a structure known as the block-tree. However, several modifications are needed to adapt the ideas from Section 7.3.1. The main difference is that we can no longer replace each bi-connected component with a single vertex in H, since cutting such vertices corresponds to cutting a much larger set in G. Instead, we will need to replace the bi-connected components with simpler bi-connected graphs such as cycles.

Lemma 91. *Replacing a bi-connected component with a cycle containing all its cut-vertices and active vertices gives a 2-vertex equivalent graph.*

Proof. As this mapping maintains the bi-connectivity of the component, it does not introduce any new cut-vertices. Therefore, G and H have the same set of cut vertices and the same block-tree structure. Note that the actual order the active vertices appear in does not matter, since they will never be separated. The claim follows similarly to Lemma 88. □

The block-tree also needs to be shrunk in a similar manner. Note that the fact that blocks are connected by shared vertices along with Lemma 91 implies the removal of inactive leaves. Any leaf component with no active vertices aside from its cut vertex can be reduced to the cut vertex, and therefore be removed. The following is an equivalent of the degree two removal part of Lemma 89.

Lemma 92. *Two bi-connected components C_1 and C_2 with no active vertices that share cut vertex w and are only incident to one other cut vertex each, u and v respectively, can be replaced by an edge connecting u and v to create a 2-vertex equivalent graph.*

Proof. As we have removed only w, any cut vertex in H is also a cut vertex in G. As C_1 and C_2 contain no active vertices, this cut would induce the same partition of active vertices.

For the cut given by removing w in G, removing u in H gives the same cut since C_1 has no active vertices (which in turn implies that u is not active). Note that the removal of u may break the graph into more pieces, but our definition of cuts allows us to place these pieces on two sides of the cut arbitrarily. □

Note that Lemma 91 may need to be applied iteratively with Lemma 92 since some of the cut vertices may no longer be cut vertices due to the removal of components attached to them.

Lemma 93. *Given a graph G with m edges and k active vertices W, a 2-vertex equivalent of G of size $O(k)$, H, can be constructed in $O(m)$ time.*

Proof. We can find all the initial block-trees using depth-first search [140]. Then we can apply Lemmas 91 and 92 repeatedly until no more reductions are possible. Several additional observations are needed to run these reduction steps in $O(m)$ time. As each cut vertex is removed at most once, we can keep a counter in each component about the number of cut vertices on it. Also, the second time we run Lemma 91 on a component, it's already a cycle, so the reductions can be done without examining the entire cycle by tracking it in a doubly linked list and removing vertices from it.

It remains to bound the size of the final block-tree. Each leaf in the block-tree has at least one active vertex that's not its cut vertex. Therefore, the block-tree contains at most $O(k)$ leaves and therefore at most $O(k)$ internal components with 3 or more cut vertices, as well as $O(k)$ components containing active vertices. If these components are connected by paths with 4 or more blocks in the block tree, then the two middle blocks on this path meet the condition of Lemma 92 and should have been removed by the above procedure. This gives a bound of $O(k)$ on the number of blocks, which in turn implies an $O(k)$ bound on the number of cut vertices. The edge count then follows from the fact that Lemma 91 replaces each component with a cycle, whose number of edges is linear in the number of vertices, and that the bi-connected components themselves are arranged in a tree. □

7.4 3-Edge Connectivity

We now extend our algorithms to 3-edge connectivity. Our starting point is a statement similar to Lemma 88, namely that we can contract all 3-edge connected components. Though of no consequence to our algorithms, we note that unlike 2-edge or biconnected components, 3-edge connected components need not be connected.

Lemma 94. *Let S be a 3-edge connected component in G. Then contracting all vertices in S to a single vertex s in H, and endpoints of edges correspondingly, creates a 3-edge equivalent graph.*

Proof. A two-edge cut will not separate a 3-edge connected component. Therefore all active vertices in S fall on one side of the cut, to which vertex s may also fall. The proof follows analogously to Lemma 88. □

Such components can also be identified in $O(m)$ time using depth-first search [229], so the preprocessing part of this algorithm is the same as with the 2-connectivity cases. However, the graph after this shrinking step is no longer a tree. Instead, it is a cactus, which in its simplest terms can be defined as:

Definition 95. *A cactus is an undirected graph where each edge belongs to at most one cycle.*

On the other hand, cactuses can also be viewed as a tree with some of the vertices turned into cycles[6]. Such a structure essentially allows us to repeat the same operations as in Section 7.3 after applying the initial contractions.

Lemma 96. *A connected undirected graph with no nontrivial* 3*-edge connected component is a cactus.*

Proof. We prove by contradiction. Let G be a graph with no nontrivial 3-edge connected component. Suppose there exists two simple cycles a and b in G with more than one vertex, and thus at least one edge, in common.

Call the vertices in the first simple cycle $a_1, \dots, a_n$ and the second simple cycle $b_1, \dots, b_m$, in order along the cycle.

Since these cycles are not the same, there must be some vertex not common to both cycles. Without loss of generality, assume (by flipping a and b) that b is not a subset of a, and (by shifting b cyclically) that b_1 is only in b and not a.

Now let b_{first} be the first vertex after b_1 in b that is common to both cycles, so

$$first \overset{\text{def}}{=} \min_i b_i \in a. \tag{7.1}$$

and let b_{last} be the last vertex in b common to both cycles

$$last \overset{\text{def}}{=} \max_i b_i \in a. \tag{7.2}$$

The assumption that these two cycles have more than 1 vertex in common means that

$$first < last. \tag{7.3}$$

We claim b_{first} and b_{last} are 3-edge connected.

We show this by constructing three edge-disjoint paths connecting b_{first} and b_{last}. Since both b_{first} and b_{last} occur in a, we may take the two paths formed by cycle a connecting b_{first} and b_{last}, which are clearly edge-disjoint.

By construction, vertices

$$b_{last+1}, \dots, b_m, b_1, \dots, b_{first-1} \tag{7.4}$$

[6]Some 'virtual' edges are needed in this construction, because a vertex can still belong to multiple cycles.

are not shared with a. Thus they form a third edge-disjoint path connecting b_{first} and b_{last}, and so the claim follows. Therefore, a graph with no 3-edge connected vertices, and thus no nontrivial 3-edge connected component has the property that two simple cycles have at most one vertex in common. □

With this structural statement, we can then repeat the reductions from the 2-edge equivalent algorithm from Section 7.3.1 to produce the 3-edge equivalent graph.

Lemma 97. *Given a graph G with m edges and k active vertices W, a 3-edge equivalent of G of size $O(k)$, H, can be constructed in $O(m)$ time.*

Proof. Lemma 96 means that we can reduce the graph to a cactus after $O(m)$ time preprocessing.

First consider the tree where the cycles are viewed as vertices. Note that in this view, a vertex that's not on any cycle is also viewed as a cycle of size 1. This can be pruned in a manner analogous to Lemma 89:

1. Cycles containing no active vertices and incident to 1 or 2 other cycles can be contracted to a single vertex.
2. Inactive single-vertex cycles incident to 1 other cycle can be removed.

This procedure takes $O(m)$ time and produces a graph with at most $O(k)$ leaves. Correctness of the first rule follows by replacing a cut of the two edges within an inactive cycle by a cut of the single contracted vertex with one of its neighbors. The second rule does not affect any cuts separating W. It remains to reduce the length of degree 2 paths and the sizes of the cycles themselves.

As in Lemma 89, all inactive vertices of degree 2 can be replaced by an edge between its two neighbors. This bounds the length of degree 2 paths and reduces the size of each cycle to at most twice its number of incidences with other cycles. This latter number is in turn bounded by the number of leaves of the tree of cycles. Hence, this contraction procedure reduces the total size to $O(k)$. □

We remark that this is not identical to iteratively removing inactive vertices of degrees at most 2. With that rule, a cycle can lead to a duplicate edge between pairs of vertices, and a chain of such cycles needs to be reduced in length.

7.5 Tri-connectivity

Tri-connectivity queries involving s and t ask for the existence of a separation pair $\{u, v\}$ whose removal disconnects s from t. In this section we will extend our techniques to offline tri-connectivity. To do so, we rely on a tree-like structure for the set of separation pairs in a bi-connected graph, the SPQR tree [140, 69]. We review these structures in Section 7.5.1 and show how to trim them in Section 7.5.2. As these trees require that the graphs are bi-connected, we extend this subroutine to our full construction in Section 7.5.3.

7.5.1 SPQR Trees

We now review the definition of SPQR Trees. We will follow the model given in Chapter 2 of [233]. For a more thorough description of SPQR Trees, please refer to [233].

SPQR trees are based on the definition of a split pair, which generalizes separation pairs by allowing the extra case of $\{u, v\}$ being an edge of G. A split component of the split pair $\{u, v\}$ is either an edge connecting them, or a maximal connected subgraph G' of G such that removing $\{u, v\}$ does not disconnect G'.

The SPQR tree is defined recursively on a graph G with a special split pair $\{s, t\}$. This process can be started by picking an arbitrary edge as the root. Each node μ in the tree $\mathcal{T}$ has an associated graph denoted as its skeleton, $\mathit{skeleton}(\mu)$, and is associated with an edge in the skeleton of its parent ν, called the virtual edge of μ in $\mathit{skeleton}(\nu)$. In this way, each virtual edge of a node ν corresponds to a child of ν. For contrast, we will also use real edges to denote edges that are present in G.

- Trivial Case: if G is a single edge from s to t, then $\mathcal{T}$ contains a single Q-node whose skeleton is G itself.

- Series Case: If the removal of the (virtual) edge st creates cut-vertices, then these cut vertices partition G into blocks $G_1 \ldots G_k$ and the block-tree has a cycle-like structure. The root of $\mathcal{T}$ is then an S-node, and $\mathit{skeleton}(\mu)$ is the cycle containing these cut vertices with virtual edges corresponding to the blocks.

- Parallel Case: If the split pair $\{s, t\}$ creates split components $G_1 \ldots G_k$ with $k \geq 2$, the root of $\mathcal{T}$ is a P-node and the skeleton contains k parallel virtual edges from s to t corresponding to the split components.

- Rigid Case: If none of the above cases apply, then the root of $\mathcal{T}$ is an R-node μ and $\mathit{skeleton}(\mu)$ is a tri-connected graph where each edge corresponds to a split pair $\{s_i, t_i\}$, and the corresponding child contains the union of all split components generated by this pair.

Note that by this construction, each edge in the SPQR tree corresponds to a split-pair. An additional detail that we omitted is that the construction of R-nodes picks only the split pairs that are maximal w.r.t. the edge st. This detail is unimportant to our trimming routine, but is discussed in [233]. Our algorithm acts directly upon the SPQR tree, and we make use of several important properties of this tree in our reduction routines.

When viewed as an unrooted tree, the SPQR tree is unique with all leaves as Q-nodes. For simplicity, we will refer to this tree with all Q-nodes removed as the simplified SPQR tree, or $\mathcal{T}_{simple}$. Also, it suffices to work on the skeletons of nodes, and the only candidates for separation pairs that we need to consider are:

1. Two cut vertices in an S-node.

2. The split pair corresponding to a P-node.

3. Endpoints of an edge in an R-node.

7.5.2 Trimming SPQR trees

We now show how to convert a bi-connected graph with k active vertices to a 3-vertex equivalent with $O(k)$ vertices and edges. Our algorithm makes a sequence of modifications on the SPQR trees similar to the trimming from Section 7.3. We will call a vertex u *internal* to a node μ of the SPQR tree if u is contained in the split graph of μ, and u is not part of the split pair associated with μ. We call a vertex u *exact* to a node μ if it is internal to μ but not to any descendants of μ. Exact vertices provide a way to count nodes according to active vertices without reusing active vertices for multiple nodes.

We first give a way to remove split components with no internal active vertices, which we will refer to as inactive split components.

Lemma 98. *Consider an inactive non-Q split component G' produced by the split pair $\{u, v\}$. We may create a 3-vertex equivalent graph H by the following replacement rule: if u and v are ≥ 3-vertex connected in G, we may replace G' with the edge uv; otherwise, we may replace G' with a vertex x and edges ux, vx.*

Proof. Consider a cut in G. If u and v are on the same side of the cut, an equivalent cut exists in H by placing u, v, and possibly x on the same side of the cut. Similarly, a cut in G that removes either u or v can be made in H by removing the same vertex. If a cut in G separates u and v, u and v are not ≥ 3-vertex connected in G and a vertex w in G' must have been removed since G' connects u and v. Removing x instead of w creates an equivalent cut in H.

In the other direction, if u and v were $\geq$ 3-vertex connected in G, also no cut in H separates u and v. If u and v were not $\geq$ 3-vertex connected in G, there exists a vertex w in G' whose removal separates u and v in G'. Therefore the cut in H produced by vertices $\{x, z\}$ for some $z \notin G'$ that separates u and v can be made in G by removing $\{w, z\}$ instead. All other cuts in H can be formed in G by placing G' on the same side as any remaining vertex in $\{u, v\}$. □

An important consequence of Lemma 98 is that an inactive split component is a leaf in $\mathcal{T}_{simple}$. Because of this, it will be easier to think of inactive split components in the same way we think of Q-nodes. We define the tree $\mathcal{T}_{simple'}$ as the tree $\mathcal{T}_{simple}$ with inactive split component leaves removed. Every leaf μ in $\mathcal{T}_{simple'}$ must have an internal active vertex u. Furthermore, any vertex internal to a node η is only internal to ancestors and possibly descendants of η. Since μ is a leaf in $\mathcal{T}_{simple'}$, it has no descendants. It follows that u is exact for μ and the tree $\mathcal{T}_{simple'}$ has $O(k)$ leaves.

We will use this to bound the number of leaves in $\mathcal{T}$. However, if G contains an R-node whose skeleton contains a complete graph between active vertices, the number of leaves in $\mathcal{T}$ can still be $\Omega(k^2)$. Therefore, we need another rule to process each skeleton of $\mathcal{T}_{simple'}$ so that we can bound its size by the number of its exact active vertices and split components.

Lemma 99. *There exists a constant c_0 such that any node μ in $\mathcal{T}_{simple'}$ whose skeleton contains a total of k exact active vertices and virtual edges corresponding to active split components can be replaced by a node whose skeleton contains $c_0 \cdot k$ vertices/edges to give a 3-vertex equivalent graph.*

In other words, the number of children of μ in T is at most $c_0 \cdot k$.

Proof. We consider the cases where the node is of type S, P, R separately.

If the node is a P-node, it suffices to consider the case where there are 3 or more inactive split components incident to the cut pair. Removing at most 2 vertices from these components will leave at least one of them intact, and therefore not change the connectivity between the separation pair, and therefore the other components. Therefore, all except 3 of these components can be discarded.

If the node is an S-node, any virtual edge in the cycle corresponding to an inactive split component that's not incident to two active vertices is replaced by actual edges via Lemma 98. Furthermore, two consecutive real edges in the cycle connecting three inactive vertices can be reduced to a single edge by removing the middle vertex in a manner analogous to Lemma 89.

Therefore the size of the cycle is proportional to the number of exact active vertices and active split components.

For an R-node, we can identify all vertices that are either exact and active, or are incident to virtual edges corresponding to active split components. Then we can simply connect these vertices together using a tri-connected graph of linear size (e.g. a wheel graph) and add the active split components back between their respective separation pairs. Every cut in the new skeleton with a child in $\mathcal{T}_{simple}$ can be produced by the original graph, and the only cuts in the original graph that can't be produced in the new one are cuts isolating an inactive component. Therefore the two graphs are 3-vertex equivalent. $\square$

Before we can bound the number of nodes in $\mathcal{T}_{simple'}$, we must eliminate long paths where a node has only one active child, which in turn has only one active child, etc. This is only possible if there exists an active vertex (vertices) internal to each split component, otherwise by Lemma 98, a node with no internal active vertices becomes a leaf. Since the internal active vertex (vertices) are shared amongst all ancestors, there is no way to associate the active vertex with a constant number of SPQR nodes. The replacement rule in this Lemma provides a way to get around this.

Lemma 100. *Let μ_1, μ_2, and μ_3 be a parent-child sequence of SPQR nodes where μ_2 and μ_3 are associated with split pairs $\{u,v\}$ and $\{x,y\}$ respectively, μ_3 is the single active child of μ_2, μ_2 is the single active child of μ_1, and either none of u, v, x, and y are active or $u = x$ and is active. We may replace μ_2 with μ_3, effectively replacing vertex u with x and v with y.*

Proof. In G, cuts induced by the removal of $\{u,v\}$ and $\{x,y\}$ both give the same partition of active vertices. Therefore, we only need one in H, which is given by the cut $\{x,y\}$. All other cuts in G not present in H only separate inactive split components, which need not be represented in H.

In the other direction, every cut in H is still present in G. Thus the new graph H preserves 3-vertex equivalence. $\square$

We can now bound the number of nodes in $\mathcal{T}_{simple'}$ after no reductions via Lemmas 98, 99, and 100 are possible.

Lemma 101. *Consider a graph G where no reductions via Lemmas 98, 99, and 100 are possible. There is a constant c_1 such that the number of nodes in G's SPQR tree $\mathcal{T}$ is no more than $c_1 \cdot k$, where k is the total number of active vertices in G.*

Proof. As explained earlier, the tree $\mathcal{T}_{simple'}$ has $O(k)$ leaves. To bound the total number of nodes, we must consider the length of a path of degree 2 nodes in $\mathcal{T}_{simple'}$. If a node μ on this path has a split pair $\{u,v\}$ with an active vertex, say u, internal to its parent η (implying η does not have u in its split pair), then u is exact for η and we may associate

u with η. Otherwise, consider two adjacent degree 2 nodes to which this does not apply. The parent may have an active vertex in its split pair not shared with its child, however then its child cannot have an active vertex in its split pair or else it fits the above situation (the active vertex in the child's split pair is internal to the parent). Therefore this can only happen once until the parent and child have at most one active vertex shared in their split pairs, and the procedure of Lemma 100 may be used to replace the parent node with the child. It follows that we may associate each active vertex along this path to a constant number of SPQR nodes.

Since the number of nodes of degree ≥ 3 in a tree is bounded by the number of leaves, the above shows $\mathcal{T}_{simple'}$ has $O(k)$ nodes. We now consider the full tree $\mathcal{T}$. This tree adds inactive split components with a constant number of Q-node leaves as well as Q-node leaves themselves to active split components. By Lemma 99, for any node μ in $\mathcal{T}_{simple'}$, we add at most $c_0 \cdot k$ extra children to μ in $\mathcal{T}$, where k is the sum of active vertices exact to μ and the number of active split components, which are children of μ in $\mathcal{T}_{simple'}$. The sum of the degrees of all the vertices in $\mathcal{T}_{simple'}$ is $O(k)$, therefore the number of extra nodes in $\mathcal{T}$ is also $O(k)$. Therefore $\mathcal{T}$ has $O(k)$ nodes.

□

We now consider applying Lemmas 98, 99, and 100 on G to produce a 3-vertex equivalent graph H in linear time.

Lemma 102. *Given a bi-connected graph G and k active vertices W, we may find the SPQR tree associated with G and apply the reduction rules given by Lemmas 98, 99, and 100 until exhaustion in linear time. From this can be constructed a graph H 3-vertex equivalent to G with $O(k)$ vertices and edges.*

Proof. The SPQR tree can be found in linear time [30]. Lemma 101 shows that continued application of Lemmas 98, 99, and 100 produces an SPQR tree $\mathcal{T}$ with $O(k)$ nodes. As each leaf of $\mathcal{T}$ is an edge of the graph it represents, this shows the resulting graph represented has $O(k)$ vertices and edges.

The rules given by Lemmas 98, 99, and 100 can be applied until exhaustion in $O(m)$ time. First, we may apply Lemma 98 by traversing each split component of $\mathcal{T}$ and applying the lemma when possible. Next, we may apply Lemma 99 by a similar traversal. Finally, the rule of Lemma 100 can be applied by keeping a stack of the SPQR nodes of a depth-first traversal of the SPQR tree $\mathcal{T}$. All three rules require a single traversal of $\mathcal{T}$ each and thus can be done in $O(m)$ total time. □

7.5.3 Constructing the Full Equivalent

We now extend this trimming routine for bi-connected graphs to arbitrary graphs. By an argument similar to that at the start of Section 7.3, we may assume that G is connected. Therefore we need to work on the block-tree in a manner similar to Section 7.3.2. We first show that we may invoke the trimming routine from Section 7.5.2 on each bi-connected component.

Lemma 103. *Let B be a bi-connected component in G and B' be a 3-vertex equivalent graph for B where the active vertices consist of all cut vertices and active vertices in B. Replacing B with B' gives H that's 3-vertex equivalent to G.*

Proof. By symmetry of the setup it suffices to show that any cut in G has an equivalent cut in H. Since B is two-connected, any cut in B that does not remove at least 2 vertices in B results in all vertices of B on the same side of the cut. This means any vertex in B that's not a cut-vertex in G can be added without changing the cut. This also means that removing these vertices has the same effect in G and H.

Therefore it suffices to consider cuts that remove 2 vertices in B. By assumption, there exists a cut in H that results in the same set of active vertices in C, and cut vertices incident to B on one side. The structure of the block tree gives that the rest of the graph is connected to B via one of the cut vertices, and the side that this cut vertex is on dictates the side of the cut that part is on. Therefore, the rest of the graph, and therefore the active vertices not in B will be partitioned the same way by this cut. □

It remains to reduce the block tree, which we do in a way analogous to Section 7.3.2. However, the proofs need to be modified for separation pairs instead of a single cut edge/vertex.

Lemma 104. *A bi-connected component containing no active vertices and incident to only one cut vertex can be removed (while keeping the cut-vertex) to create a 3-vertex equivalent graph.*

Proof. Any cut in G can be mapped over to H by removing the same set of vertices (minus the ones in this component). Given a cut in H, removing the same set of vertices in G results in a cut with this component added to one side. As there are no active vertices in this cut, the cut separates the active vertices in the same manner. □

Long paths of bi-connected components can be handled in a way similar to Lemma 92.

Lemma 105. *Two bi-connected components C_1 and C_2 with no active vertices that share cut vertex w and are only incident to one other cut vertex each, u and v respectively, can be replaced by an edge connecting u and v, preserving 3-vertex equivalence.*

Proof. As there is a path from u to v through these components and w, a minimum cut that separates W in G either removes one of u, v, or w, or has uv on the same side. In the first case, removing u or v leads to the same cut as none of them are active, while in the second case the edge uv does not cross the cut. Since uv is connected by an edge in H, in a cut in H they're either on the same side, or one of u or v is removed. In either case, an equivalent cut in G can be formed by assigning what remains of C_1 and C_2 to the same side as any remaining vertex. □

Our final construction can be obtained by applying these routines to the block tree first, then Lemma 103.

Lemma 106. *Given a graph G with m edges and k active vertices W, a 3-vertex equivalent of G of size $O(k)$, H, can be constructed in $O(m)$ time.*

Proof. By a proof similar to Lemma 93, applying Lemmas 104 and 105 repeatedly leads to a 3-vertex equivalent graph H whose block-tree contains $O(k)$ components and cut vertices.

The total number of cut vertices and active vertices summed over all bi-connected components is $O(k)$. Therefore, applying Lemma 103 on each component gives the size bound. □

Chapter 8

Simple Minimum Cuts in Near-Linear Time

8.1 Introduction

The minimum cut problem on an undirected (weighted) graph G asks for a vertex subset S such that the total number (weight) of edges from S to $V \setminus S$ is minimized. The minimum cut problem is a fundamental problem in graph optimization and has received vast attention by the research community across a number of different computation models [153, 103, 205, 150, 155, 121, 228, 145, 59, 133, 51, 152, 113, 158, 134, 64, 112, 187, 220, 114]. Its applications include network reliability [207, 151], cluster analysis [38], and a critical subroutine in cutting-plane algorithms for the traveling salesman problem [12].

A seminal result in weighted minimum cut algorithms is an algorithm by Karger [153] which produces a minimum cut on an m-edge, n-vertex graph in $O(m \log^3 n)$ time with high probability[1]. This algorithm stood as the fastest minimum cut algorithm for the past two decades, until very recently, work published on arXiv shaved a log factor in Karger's approach [182, 110]. The main component of Karger's algorithm is a subroutine that finds a minimum cut that 2-respects (cuts two edges of) a given spanning tree T of a graph G. In other words, the cut found is minimal amongst all cuts of G that cut exactly two edges of T. Despite the number of pairs of spanning tree edges totaling $\Omega(n^2)$, Karger shows this can be accomplished in $O(m \log^2 n)$ time. Unfortunately, the procedure developed is particularly complex, a detail Karger admits when comparing the algorithm to a simpler $O(n^2 \log n)$ algorithm he develops to find *all* minimum cuts [153]. Indeed, perhaps for this reason, implementation of the asymptotically fastest minimum cut algorithm has been

[1]With probability $1 - 1/n^c$ for some constant c, the size of the minimum cut produced is minimal.

avoided in practical performance analyses [59, 145].

In this chapter, we give a simple algorithm to find a minimum cut that 2-respects a spanning tree T of a graph G. Our procedure runs in $O(m \log^2 n)$ time, matching the performance of Karger's more-complicated subroutine. We achieve the simplification via a clever use of the heavy-light decomposition. Although our procedure requires the top tree data structure [10] to achieve optimal performance, at the cost of an extra $O(\log n)$ factor, heavy-light decomposition can be used a second time so that only augmented binary search trees are required. We also give a self-contained version of Karger's algorithm [153] with this new procedure and implement it, avoiding issues associated with previous implementations [153, 59].

Karger's algorithm [153], as well as the edge-sampling technique it is based on [152], has been extended and adapted to achieve results in a number of different settings [114, 64, 228, 113, 112, 187]. In particular, in the fully-dynamic setting, Thorup [228] uses the tree-packing technique developed by Karger [153], but maintains a larger set of trees so that the minimum cut 1-respects at least one of them. In the parallel setting, Geissmann and Gianinazzi [113] are able to parallelize both the dynamic tree data structure and the necessary computation required by Karger's algorithm [153]. This work is based off prior work in the cache-oblivious model [112], also based on Karger's algorithm [153]. In the distributed setting, Ghaffari and Kuhn [114] achieve a $(2+\epsilon)$-approximation to the minimum cut based on Karger's sampling technique [152]. This is improved to a $(1+\epsilon)$-approximation with similar runtime by Nanongkai and Su [187]. Nanongkai and Su develop their algorithm from Thorup's fully-dynamic min-cut algorithm [228], Karger's sampling technique [152], and Karger's dynamic program to find the minimum cut that 1-respects a tree [153]. Finally, Daga et al. [64] achieve a sublinear time distributed algorithm to compute the exact minimum cut in an unweighted undirected graph. This algorithm builds off a more recent development in minimum cut algorithms [158], combined again with the tree-packing technique introduced by Karger [153]. Specifically, a tree packing is found in an efficient number of distributed rounds, then Karger's more-complicated algorithm to find a minimum 2-respecting cut is applied in the distributed setting.

This vast amount of work based on Karger's original near-linear time algorithm suggests that simplifying it may yield additional techniques that can be applied both sequentially and in alternative settings. Indeed, the very recent improvements to Karger's algorithm [182, 110] were published on arXiv two months after the paper this chapter is based on was first made available online [174], one of which [110] cites our work as what drew the authors to the problem. Indeed, their procedure for "descendent edges", given in Section 3.1, is similar to our procedure given in Section 8.5. We have further found use of the approach given in this chapter to achieve new results in dynamic higher connectivity algorithms [179].

This chapter is organized as follows. In Section 8.2, we state the history of the minimum cut problem, in particular discussing other simple algorithms. In Section 8.3, we give an overview of Karger's algorithm to pack spanning trees, leaving the details of the approach to an optional Section 8.8, appearing at the end of the chapter. Our main contribution is given in Sections 8.4 and 8.5. In Section 8.4, we show how to find minimum cuts that 1-respect (cut one edge of) a tree using our new procedure. In Section 8.5, we extend the approach to find minimum cuts that 2-respect (cut two edges of) a tree. We discuss our implementation in Section 8.6 and give concluding remarks in Section 8.7.

8.2 Related Work

Before we begin, we give a brief history of the minimum cut problem. The minimum cut problem was originally perceived as a harder variant of the maximum s-t flow problem and was solved by $\binom{n}{2}$ flow computations. Gomory and Hu [116] showed how to compute all pairwise max flows in $n-1$ flow computations, thus reducing the complexity of the minimum cut problem by a $\Theta(n)$ factor. Hao and Orlin [130] further showed that the minimum cut in a directed graph can be reduced to a single flow computation.

Nagamochi and Ibaraki [186, 185] developed a deterministic algorithm that is not based on computing maximum s-t flows. They achieve $O(nm + n^2 \log n)$ time on a capacitated, undirected graph. This procedure was simplified by Stoer and Wagner [220], achieving the same runtime. The Stoer-Wagner algorithm gives a simple procedure to find an *arbitrary* minimum s-t cut. Vertices s and t are then merged, and the procedure repeats. Although the $O(nm+n^2 \log n)$ time complexity requires an efficient priority queue such as a Fibonacci heap [99], a binary heap can be used to achieve runtime $O(nm \log n)$.

Two algorithms based on *edge contraction* have been devised. The first is an algorithm of Karger [150] and is incredibly simple. The algorithm randomly contracts edges until only two vertices remain. Repeated $O(n^2 \log n)$ times, the algorithm finds all minimum cuts on an undirected, weighted graph in $O(n^2 m \log n)$ time with high probability. This technique was improved by Karger and Stein [155] by observing an edge of the minimum cut is more likely to be contracted later in the contraction procedure. Their improvement branches the contraction procedure after a certain threshold has been reached, spending more time to avoid contracting an edge of the minimum cut when fewer edges remain. The Karger-Stein algorithm achieves runtime $O(n^2 \log^3 n)$, finding the minimum cut with high probability.

In an unweighted graph, Gabow [103] showed how to compute the minimum cut in $O(cm \log(n^2/m))$ time, where c is the capacity of the minimum cut. Karger [152] improved Gabow's algorithm by applying random sampling, achieving runtime $\tilde{O}(m\sqrt{c})$ in expectation[2]. The sampling technique developed by Karger [152], combined with the tree-packing technique devised by Gabow [103], form the basis of Karger's near-linear time minimum cut algorithm [153]. As previously mentioned, this technique finds the minimum cut in an undirected, weighted graph in $O(m \log^3 n)$ time with high probability.

A recent development uses low-conductance cuts to find the minimum cut in an undirected unweighted graph. This technique was introduced by Kawarabayashi and Thorup [158], who achieve near-linear deterministic time (estimated to be $O(m \log^{12} n)$). This was improved by Henzinger, Rao, and Wang [134], who achieve deterministic runtime $O(m \log^2 n\, (\log \log n)^2)$. Although the algorithm of Henzinger et al. is more efficient than Karger's algorithm [153] on unweighted graphs, the procedure, as well as the one it was based on [158], are quite involved, thus making them largely impractical for implementation purposes.

Since an earlier version of the paper on which this chapter is based became available online [174], several important improvements in minimum cut algorithms have been discovered. Ghaffari et al. [115] devise a randomized unweighted minimum cut algorithm by using contraction based on sampling from each vertex, rather than standard uniform edge sampling. Their algorithm reduces unweighted minimum cuts to weighted minimum cuts on a graph with $O(n)$ edges, achieving $O(\min(m + n \log^3 n, m \log n))$ time complexity. Gawrychowski et al. [110] improve Karger's procedure for finding the minimum cut that 2-respects a tree to $O(m \log n)$ time. This improves the state-of-the-art for weighted minimum cuts to $O(m \log^2 n)$ time and, by Ghaffari et al. [115], improves the complexity of unweighted minimum cuts to $O(\min(m + n \log^2 n, m \log n))$ time. Mukhopadhyay and Nanongkai [182] also study Karger's procedure for finding the minimum cut that 2-respects a tree, arriving at an $O(m \frac{\log^2 n}{\log \log n} + n \log^6 n)$ time weighted minimum cut algorithm. Mukhopadhyay and Nanongkai further apply their new procedure to minimum cuts in the cut-query and streaming models.

8.3 Overview of Karger's Spanning Tree Packing

We first formalize the definition mentioned earlier in this chapter and originally given by Karger.

Definition 107 (Karger [153])**.** *Let T be a spanning tree of G. We say that a cut in G k-respects T if it cuts at most k edges of T. We also say that T k-constrains the cut in G.*

We also define weighted tree packings.

[2]The $\tilde{O}(f)$ notation hides polylog f factors.

Definition 108 (Karger [153]). *A weighted tree packing is a set of spanning trees, each with an assigned non-negative weight, such that the total weight of trees containing a given edge of G is no greater than the weight of that edge. The weight of the packing is the total weight of the trees in it.*

The first stage of Karger's algorithm is to sample edges independently and uniformly at random from graph G to form a graph H, and then pack spanning trees in H. If we sample a tree T from a packing with probability proportional to its weight, a minimum cut in G will cut at most two edges of T with constant probability. Thus, if we sample $O(\log n)$ trees from the weighted packing, a minimum cut in G 2-respects at least one of the sampled trees with high probability. The remainder of the algorithm is a procedure that, given a spanning tree T of a graph G, finds a minimal cut of G that 2-respects T. This procedure is applied to all $O(\log n)$ sampled spanning trees.

We leave the intuition behind Karger's approach and the relevant mathematics to Section 8.8, to be read at the reader's discretion. Here we also include versions of our algorithms with general constants.

We will use Algorithm 6 to pack spanning trees, credited to Thorup and Karger [226], Plotkin-Shmoys-Tardos [205], and Young [242]. The procedure appears in Gawrychowski et al. [110].

Algorithm 6 Obtain a Packing of Weight at least $.4c$ from a Graph G

```
Let $G$ be a graph with $m$ edges and $n$ vertices.
1. Initialize $\ell(e) \leftarrow 0$ for all edges $e$ of $G$. Initialize multiset $P \leftarrow \varnothing$. Initialize $W \leftarrow 0$.
2. Repeat the following:
    (a) Find a minimum spanning tree $T$ with respect to $\ell(\cdot)$.
    (b) Set $\ell(e) \leftarrow \ell(e) + 1/(75 \ln m)$ for all $e \in T$. If $\ell(e) > 1$, return $W, P$.
    (c) Set $W \leftarrow W + 1/(75 \ln m)$.
    (d) Add $T$ to $P$.
```

Lemma 109 ([205, 226, 242]). *Given an undirected unweighted graph G with m edges, n vertices, and minimum cut c, Algorithm 6 returns a weighted packing of weight at least $.4c$ in $O(mc \log n)$ time.*

Algorithm 6 and Lemma 109 are given in Section 8.8 with general epsilon and proven. To achieve $O(mc\log n)$ time in Algorithm 6, we may use a linear time minimum spanning tree routine [154] or the following implementation trick given by Gawrychowski et al. [110]. In the use of Algorithm 6 in Algorithm 7, the graph in Algorithm 6 has edges which may be duplicated $O(\log n)$ times, while the number of distinct edges can be bounded as a factor $\Theta(\log n)$ fewer. It suffices to invoke the minimum spanning tree algorithm of Algorithm 6 with only the minimum of each set of parallel edges. We can easily maintain the minimum of each set of parallel edges in $O(\log n)$ time per edge per iteration, which suffices to shave a log factor in the runtime of Algorithm 6. Note that if we chose to avoid these optimizations and/or avoid the use of top trees in Section 8.5, the final runtime becomes $O(m\log^4 n)$.

We use Algorithm 6 in Algorithm 7 to obtain $\Theta(\log n)$ trees for the 2-respect algorithm given in Sections 8.4 and 8.5.

Lemma 110. *Algorithm 7 returns a collection of $\Theta(\log n)$ spanning trees of G in time $O(m\log^3 n)$ such that the minimum cut of G 2-respects at least one tree in the collection with high probability.*

Algorithm 7 and Lemma 110 are given in Section 8.8 with general epsilon and proven.

8.4 Minimum Cuts that 1-Respect a Tree

We now give our algorithm for finding a minimum cut that 1-respects a spanning tree T of a graph G. We present it here only to build intuition for the idea used to find 2-respecting cuts in the following section, which also finds 1-respecting cuts.

We use the following lemma, a consequence of Sleator and Tarjan's heavy-light decomposition [218].

Lemma 111 (Sleator and Tarjan [218])**.** *Given a tree T, there is an ordering of the edges of T such that the edges of the path between any two vertices in T consist of the union of up to $2\log n$ contiguous subsequences of the order. The order can be found in $O(n)$ time.*

Proof. We use heavy-light decomposition, credited to Sleator and Tarjan [218]. Note that the algorithm assumes T is rooted. We can root T arbitrarily. We then take the heavy paths given from the usual construction and concatenate them in any order. □

Our algorithm begins by labeling the edges of T in heavy-light decomposition order $e_1, \dots, e_{n-1}$ as given by Lemma 111. Consider the cut of G induced by the vertex partition resulting from cutting a single edge of T. We iterate index i through heavy-light decom-

Algorithm 7 Obtain $\Theta(\log n)$ Spanning Trees for the 2-respect Algorithm

Let d denote the exponent in the probability of success $1 - 1/n^d$. Let $b = 3 \cdot 6^2(d+2)\ln n$.

1. Form graph G' from G by first normalizing the edge weights of G so the smallest non-zero edge weight has weight 1, then multiplying each edge weight by 100 and rounding to the nearest integer. Let U be an upper bound for the size of the minimum cut of G'.

2. Initialize $c' \leftarrow U$. Repeat the following:

   (a) Construct H in the following way: for each edge e of G', let e have weight in H drawn from the binomial distribution with probability $p = \min(b/c', 1)$ and number of trials the weight of e in G'. Cap the weight of any edge in H to at most $\lceil 7/6 \cdot 12b \rceil$.

   (b) Run Algorithm 6 on H, considering an edge of weight w as w parallel edges. There are three cases:

      i. If $p = 1$, set P to the packing returned and skip to step 3.

      ii. If the returned packing is of weight $24b/70$ or greater, set $c' \leftarrow c'/6$ and repeat steps 2a and 2b, setting P to the packing returned and then proceeding to step 3.

      iii. Otherwise, repeat steps 2a and 2b with $c' \leftarrow c'/2$.

3. Return $\lceil 36.53 d \ln n \rceil$ trees sampled uniformly at random proportional to their weights from P.

---

position order and keep up-to-date the total weight of all edges of G that cross the cut induced by e_i. The minimum weight found is then returned.

Call the edges of G in T *tree edges* and edges of G not in T *non-tree edges*. Critical to our approach is the following proposition.

Proposition 112. *For any cut of G that 2-respects T, the non-tree edge uv crosses the cut if and only if exactly one tree edge from the uv-path in T crosses the cut.*

Proof. Recall that for any edge of T crossing the cut, the components of each of its endpoints must fall on opposite sides of the cut. Therefore if the number of tree edges in the cut on the uv-path in T is odd, the non-tree edge uv crosses the cut. Since we are only considering cuts that cut at most 2 edges of T, the proposition follows. □

We now give our algorithm explicitly.

Algorithm 8 Minimum Cuts that 1-Respect T

1. Arrange the edges of T in the order of Lemma 111; label them $e_1, \ldots, e_{n-1}$.
2. For each non-tree edge uv, mark every i such that e_i is on the uv-path in T and e_{i+1} is not on the uv-path in T, or vice versa. Indicate whether edge e_1 is on the uv-path in T.
3. Iterate index i from 1 to $n-1$, in each iteration keeping track of the total weight of all non-tree edges uv such that e_i lies on the uv-path in T, added together with the weight of edge e_i.
4. Return the minimum total weight found in step 3.

Lemma 113. *Algorithm 8 finds the value of the minimum cut that* 1*-respects a spanning tree T of a graph G in $O(m \log n)$ time.*

Proof. Via Proposition 112, in a 1-respecting cut including only e_i from T, a non-tree edge uv is cut if and only if the edge e_i lies on the uv-path in T. Algorithm 8 keeps track of all such non-tree edges for each possible e_i that is cut, therefore it finds the minimum cut of G that cuts a single edge of T.

The time complexity can be determined as follows. Finding the heavy-light decomposition for step 1 takes $O(n)$ time. In doing so, we can label each edge and each heavy path so that every edge knows its index in the order as well as the heavy path to which it belongs. Each heavy path can store its starting and ending index in the order. With this information, step 2 can be completed by walking up from u and v in T towards the root of T. We spend $O(1)$ work per heavy path from root to vertex, which is bounded by $O(\log n)$ via the heavy-light decomposition. In total this step takes $O(m \log n)$ time.

In step 3, we spend $O(n)$ total work plus $O(1)$ work for each transition of the current edge e_i on or off the uv path for all non-tree edges uv. Each non-tree edge transitions on or off $O(\log n)$ times as guaranteed by Lemma 111, therefore the time complexity of this step is $O(m \log n)$. Overall, Algorithm 8 takes $O(m \log n)$ time. □

Note that if we wish to find the edges in the minimum cut, we can keep track of the minimum-achieving index i so we know the vertex separation of the minimum cut. With the vertex separation, it is easy to find in $O(m \log n)$ time which non-tree edges cross the cut.

Further note that we need not know the identity of the non-tree edge uv as e_i falls on or off the uv-path. Thus the space required for step 2 need only be $O(m)$, since at each transition point we can just keep track of the total weight added or subtracted from the minimum cut.

8.5 Minimum Cuts that 2-Respect a Tree

We now discuss an extension of Algorithm 8 to find a minimum cut that 2-respects a tree. We still iterate i through heavy-light decomposition order, but in addition to cutting e_i, we find the best j so that the cut resulting from cutting e_i and e_j is minimal. To find the best j efficiently we use a clever data structure.

Lemma 114 (Alstrup et al. [10]). *There is a data structure that supports the following operations on a weighted tree T in $O(\log n)$ time:*

- *`PathAdd(u, v, x)` := Add weight x to all edges on the unique uv-path in T.*
- *`NonPathAdd(u, v, x)` := Add weight x to all edges not on the unique uv-path in T.*
- *`QueryMinimum()` := Query for the minimum weight edge in T.*

Proof. Operations `PathAdd()` and `QueryMinimum()` are just Theorems 3 and 4 of [10]. Operation `NonPathAdd(u, v, x)` can be achieved by keeping a counter of global weight added to (subtracted from) T and executing `PathAdd(u, v, -x)` to undo this action on the uv-path. See also [225]. □

Note that the weight x can be positive or negative.

If we seek to avoid implementing any sophisticated data structures, we can instead use heavy-light decomposition again and support the above two operations in $O(\log^2 n)$ time. To see how, by Lemma 111 each path of T represents at most $O(\log n)$ contiguous segments of the total order of edges. Range add and a global minimum query can be supported in $O(\log n)$ time via an augmented binary search tree. Thus the total time complexity per operation is $O(\log^2 n)$.

We use the range operations as follows. As we iterate index i through the order of Lemma 111, we keep up to date the cost of the cut resulting from cutting any other edge e_j via the data structure of Lemma 114. Instead of querying each other edge e_j directly, however, we just use a global minimum query to find the best choice of j. The procedure is given in Algorithm 9. The first two steps are the same as Algorithm 8.

Algorithm 9 Minimum Cuts that 2-Respect T

1. Arrange the edges of T in the order of Lemma 111; label them $e_1, \ldots, e_{n-1}$.

2. For each non-tree edge uv, mark every i such that e_i is on the uv-path in T and e_{i+1} is not on the uv-path in T, or vice versa. Indicate whether edge e_1 is on the uv-path in T.

3. Initialize the data structure of Lemma 114 on T so that the weight of edge e_j is equal to its weight in T.

4. Iterate index i from 1 to $n-1$. Via the computation done in step 2, maintain the following invariants in the data structure of Lemma 114 as i is iterated.

   (a) When edge e_i is on the uv-path in T, add the weight of non-tree edge uv to all edges off the uv-path in T.

   (b) When edge e_i is off the uv-path in T, add the weight of non-tree edge uv to all edges on the uv-path in T.

   Each time i is incremented, after updating weights in Lemma 114 as per 4a and 4b, add ∞ to edge e_i, execute `QueryMinimum()`, then subtract ∞ from edge e_i. The value of the minimum cut found in each iteration is the result of `QueryMinimum()` plus the weight of e_i.

5. Return the minimum of the smallest cut found in step 4 with the result of `QueryMinimum()` when we consider edge e_i to be off the path of all non-tree edges uv in the data structure of Lemma 114.

Lemma 115. *Algorithm 9 finds the value of the minimum cut that 2-respects a spanning tree T of a graph G in $O(m \log^2 n)$ time.*

Proof. By Proposition 112, in a 2-respecting cut including e_i and e_j of T, a non-tree edge uv is cut if and only if exactly one of e_i or e_j lies on the uv-path in T. Observe that the invariants enforced in step 4 guarantee that in each iteration the total weight of edges from the cut resulting from cutting any other edge e_j along with e_i is kept up-to-date in the data structure of Lemma 114. Since the minimum such j is found for every i, it follows that step 4 finds the weight of the minimum cut of G that cuts exactly two edges of T. In step 5, we return the minimum of this weight with a single call to `QueryMinimum()` where we assume edge e_i to be off the path of all non-tree edges uv. Observe that this computes the minimum cut of G that cuts exactly one edge of T. Thus, the minimum cut of G that 2-respects T is returned in step 5.

The time complexity follows similarly to Algorithm 8. Steps 1 and 2 take $O(m \log n)$ total time. However, step 4 requires $O(\log n)$ time for non-tree edge uv whenever edge e_i falls on or off the uv-path in T, since the data structure of Lemma 114 takes $O(\log n)$ time per operation. For a given non-tree edge uv, edge e_i falls on or off the uv-path in T a total of $O(\log n)$ times by Lemma 111; thus step 4 takes $O(m \log^2 n)$ time. The final `QueryMinimum()` call in step 5 takes $O(m \log n)$ time. The total time taken is $O(m \log^2 n)$. □

We make a few further remarks about Algorithm 9. To determine the edges of the minimum cut, the data structure of Lemma 114 can be augmented to return the index j of the edge that achieves the minimum given in operation `QueryMinimum()`. With e_i and e_j, we can determine the vertex partition in G of the minimum cut and, as stated in Section 8.4, and from this we can find which non-tree edges cross the minimum cut easily in $O(m \log n)$ time.

The space complexity of Algorithm 8 was easily linear. In Algorithm 9, we must know the identity of each non-tree edge uv in every transition point where edge e_i falls on or off the uv-path. Naively this costs $O(m \log n)$ space. This can be improved to $O(m)$ space by performing step 2 incrementally while executing step 4. That is, we only need to know the next transition point where the non-tree edge uv falls on or off the uv-path, and from the current transition point this can be determined in constant time via the heavy-light decomposition.

Recall that while Algorithm 8 helped demonstrate the approach of Algorithm 9, we need only implement Algorithm 9, since Algorithm 9 finds the minimum cut of G that cuts either 1 or 2 edges of T.

From this we get our final theorem, equivalent to the result of Karger [153].

Theorem 116. *The minimum cut in a weighted undirected graph can be found in $O(m \log^3 n)$ time with high probability.*

Proof. We first find $\Theta(\log n)$ spanning trees by Algorithm 7. We then find the minimum cuts that 2-respect each of these trees by Algorithm 9. By Lemmas 110 and 115, this finds the minimum cut with high probability in $O(m \log^3 n)$ time. □

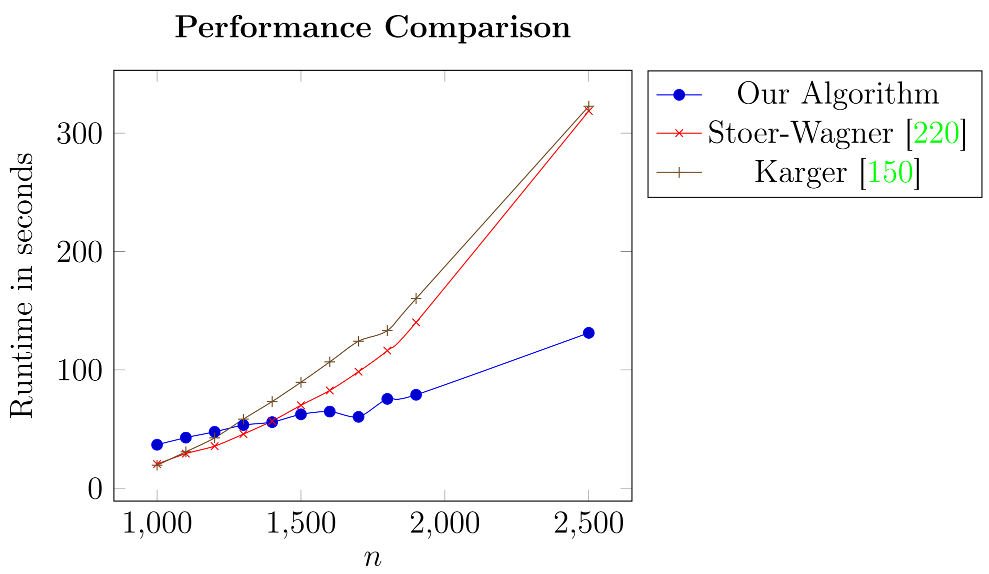

Figure 8.1: Performance comparison of an $O(m \log^4 n)$ implementation of our algorithm with an $O(n^3)$ Stoer-Wagner [220] and $O(n^3 \log n)$ Karger [150].

8.6 Implementation

We have implemented an $O(m \log^4 n)$ version of our algorithm in C++[3]. Algorithm 6 together with an $O(m \log n)$ minimum spanning tree routine take about 100 lines of code, Algorithm 7 takes about 200 lines, Algorithm 9 takes about 200 lines, and using an augmented binary search tree as the data structure for Lemma 114 takes about 200 lines. To the best of our knowledge, our implementation is the first to achieve near-linear time complexity. We have tested it against an $O(n^3)$ implementation of the Stoer-Wagner algorithm [220] and an $O(n^3 \log n)$ implementation of Karger's randomized contraction algorithm [150]. Under favorable inputs, the runtime compares as in Figure 8.1.

Figure 8.1 demonstrates the near-linear growth in the running time of our algorithm. Unfortunately, it does not appear our implementation is competitive compared to existing implementations [59]. The bottleneck is in obtaining the $O(\log n)$ spanning trees for Algorithm 9, even when Algorithm 7 runs in $O(m \log^3 n)$ time and Algorithm 9 runs in $O(m \log^4 n)$ time. The issue is the large constant factors due to the quadratic dependencies on epsilons, seen in Algorithms 10 and 11. We have calculated that the number of calls to the minimum spanning tree routine in our implementation can be as much as $8100 \ln n \ln m$,

[3]Our implementation is available at: `https://github.com/nalinbhardwaj/min-cut-paper`.

and that changing the choices of epsilons for Algorithms 6 and 7 does not yield significant improvement.

If we replace Algorithm 6 with the more-complicated Gabow's algorithm [103], we can likely improve our implementation's runtime. Further, a factor of about two can be saved by finding c' via an approximation algorithm [156]. However, a large constant factor will remain due to the sampling procedure in Lemma 120, discussed in Section 8.8. All known algorithms to compute weighted tree packings have dependence on c, the value of the minimum cut, and Lemma 120 reduces the value of the minimum cut to at least $3(d+2)(\ln n)/\epsilon^2$, which in our algorithms manifests as a factor of $108(d+2)\ln n$. It appears that for Karger's approach to be made practical, this large constant factor will likely need to be improved or heuristic approaches would need to be considered [59].

8.7 Conclusion

In this chapter, we have discussed a simplification to Karger's original near-linear time minimum cut algorithm [153]. In contrast to Karger's original algorithm [153], finding spanning trees that have a constant probability of 2-respecting the minimum cut is now the more-complicated part of the algorithm and finding minimum cuts that 2-respect a tree is relatively simpler. In actuality, both were complicated in Karger's original algorithm, however the work to find the tree packing was largely abstracted to previous publications. The same can be said for many statements of Karger's near-linear time algorithm [110, 182]. Our version, on the other hand, is self-contained: the only procedures outside of Algorithms 6, 7, and 9 required to implement the full algorithm are a minimum spanning tree subroutine and (optionally) a top tree data structure.

The main contribution of our algorithm is a new, simple procedure to find a minimum cut that 2-respects a tree T in $O(m\log^2 n)$ time. Karger advertises that the complexity of his near-linear time algorithm is $O(m\log^3 n)$ and thus his routine to find a minimum cut that 2-respects a tree also takes $O(m\log^2 n)$ time. However, he gives two small improvements to the algorithm to reduce the overall runtime to $O(m\log^2 n\log(n^2/m)/\log\log n + n\log^6 n)$. The first uses the fact that finding a 1-respecting cut can be done in linear time, and the other is an improvement which reduces an $O(\log n)$ factor to an $O(\log(n^2/m))$ factor in the 2-respect routine. For our algorithm, the first improvement can be applied by substituting our 1-respect algorithm with his. The second improvement can not be applied. Thus, when $m = \Theta(n^2)$, his algorithm is faster by an $O(\log n)$ factor. However, for this case, Karger gives a different, simpler algorithm [153] which finds the global minimum cut in $O(n^2\log n)$ time anyway.

There are three algorithms that are referred to as simple min-cut algorithms: the

Stoer-Wagner algorithm [220] which runs in $O(nm \log n)$ time or $O(nm + n^2 \log n)$ time with a Fibonacci heap [99], Karger's randomized contraction algorithm [150] which runs in $O(n^2 m \log n)$ time, and the improvement to Karger's algorithm by Karger and Stein [155] which runs in $O(n^2 \log^3 n)$ time. In comparison to these, our approach is the least simple. However, our $O(m \log^3 n)$ runtime is significantly better. While the large constant factors in our approach make this only relevant at large values of n, we hope the procedure developed in this chapter can be used in conjunction with an optimized version of Karger's sampling technique to produce an asymptotically fast, practical minimum cut algorithm.

8.8 Karger's Algorithm for Packing Spanning Trees

In this section we give the intuition and mathematics behind the spanning tree packing of Karger's algorithm.

8.8.1 Tree Packing

The basic idea of Karger's near-linear time algorithm [153] is to exploit the following combinatorial result. Recall that a tree packing of an undirected unweighted graph G is a set of spanning trees such that each edge of G is contained in at most one spanning tree. The weight of a tree packing is the number of trees in it.

Theorem 117 (Nash-Williams [188]). *Any undirected unweighted multigraph with minimum cut c contains a tree packing of weight at least $c/2$.*

Now consider a minimum cut and a tree packing given by Theorem 117. Each edge of the minimum cut can only be present in at most one spanning tree. As there are c edges of the minimum cut, this implies that the average spanning tree contains at most $c/(c/2) = 2$ edges of the minimum cut. In other words, a spanning tree chosen at random from a packing of Theorem 117 will 2-constrain the minimum cut with probability at least 1/2.

Suppose we are given a spanning tree T of G with each edge of T marked if it crosses the minimum cut. The endpoints of any marked edge must fall on opposite sides of the cut. Conversely, the endpoints of any unmarked edge must be on the same side of the cut. It follows that if we know the edges of T that cross the minimum cut, we can determine the vertex partition of the minimum cut and its total weight in G.

This gives the intuition behind Karger's algorithm [153]. We sample spanning trees from a tree packing of G and for each tree T, we find the minimum cut that 2-respects T. Unfortunately, several obstacles need be overcome before this can be made into an efficient algorithm. For one, all currently known approaches of determining a tree packing of Theorem 117 have runtime $\Omega(cm)$, which for large values of c is far more than the runtime we seek. Further, Theorem 117 must be generalized to weighted graphs.

We first address the latter concern. Recall the definition of weighted tree packings given in Section 8.3.

Lemma 118 (Karger [153]). *Any undirected weighted graph with minimum cut c contains a weighted tree packing of weight at least $c/2$.*

Proof. For contradiction, suppose some graph G with minimum cut c and $\epsilon > 0$ exist such that G does not contain a weighted packing of weight $(1-\epsilon)c/2$ or greater.

Take G and approximate each edge e_i of weight w_i by a rational number a_i/b_i such that $a_i/b_i < w_i$ and $w_i - a_i/b_i < \epsilon$. Multiply all edges by $d = \prod_i b_i$ and call the resulting graph G'. Then by Theorem 117, when viewed as an unweighted multigraph, G' has a tree packing of weight at least $(1-\epsilon)dc/2$. If we weight each tree of the packing by $1/d$, the packing becomes a weighted packing of G of weight at least $(1-\epsilon)c/2$, a contradiction. □

Note that for both Lemma 117 and Lemma 118, an upper bound of weight c also exists, because every spanning tree in the packing must cross the minimum cut at least once.

To effectively use Lemma 118, we formally state the relationship between weighted packings and trees that 2-constrain small cuts.

Lemma 119 (Karger [153]). *Consider a weighted graph G and a weighted tree packing of weight βc, where c is the weight of the minimum cut in G. Then given a cut of weight αc, a fraction at least $\frac{1}{2}(3-\alpha/\beta)$ of the trees (by weight) 2-constrain the cut.*

Proof. Note that every spanning tree must cross every cut. Let x denote the total weight of trees with at least three edges crossing the cut and y the total weight of trees with one or two edges crossing the cut. Then $x + y = \beta c$ and $3x + y \le \alpha c$. Rearranging, we get $y \ge \frac{1}{2}(3\beta c - \alpha c)$. □

8.8.2 Random Sampling

In order to avoid the $\Omega(cm)$ complexity of finding a packing of weight $c/2$, we first apply random sampling to G. Specifically, we use the following from Karger's earlier work.

Lemma 120 (Karger [152]). *Let $p = 3(d+2)(\ln n)/(\epsilon^2\gamma c) \le 1$, where c is the weight of the minimum cut of an unweighted multigraph G and $\gamma \le 1, \gamma = \Theta(1)$. Then if we sample each edge of G independently with probability p, the resulting graph H has the following properties with probability $1 - 1/n^d$.*

1. *The minimum cut in H is of size within a $(1+\epsilon)$ factor of $cp = 3(d+2)(\ln n)/(\gamma\epsilon^2)$, which is $O(\epsilon^{-2}\log n)$.*

2. *A cut in G takes value within a factor $(1+\epsilon)$ of its expected value in H. In particular, the minimum cut in G corresponds (under the same vertex partition) to a $(1+\epsilon)$-times minimum cut of H.*

By picking ϵ to be a constant such as $1/6$, Lemma 120 will allow us to reduce the size of the minimum cut in H to $O(\log n)$. We can then run existing algorithms [205, 103] to pack trees in H in $\tilde{O}(m)$ time. Further, since the minimum cut of G corresponds to a $(1+\epsilon)$-times minimum cut of H, we can still apply Lemma 119 on the sampled graph H so that a tree randomly sampled from the packing has a constant probability of 2-constraining the minimum cut in G.

There are still several issues to resolve. Lemma 120 applies to unweighted multigraphs G, but our graph G can have non-negative real weights. The other issue is that the value γ needs to be known ahead of time in order to apply the lemma. We first address the latter issue.

Lemma 120 requires knowing a constant-factor underestimate $c' = \gamma c$ for the minimum cut c. In particular, without $\gamma \le 1$, property 2 of Lemma 120 is not guaranteed with high probability, and if $\gamma = o(1)$, the minimum cut of H will be of size $\omega(\epsilon^{-2}\log n)$ with high probability. We may run a linear-time 3-approximation algorithm [175], with modifications to work on weighted graphs [156], to find this approximation. This is simple to state, but more difficult to implement.

A different approach is to start with a known upper bound U for c'. Karger states that we can then halve this upper bound until "our algorithms succeed" [152]. This approach is taken by the implementation of Chekuri et al. [59]. Unfortunately, it is not rigorous as stated. Lemma 120 indicates that with a constant-factor underestimate $c' = \gamma c$ for c, our algorithm can proceed. However, it does not give a process for rejecting a guess c' that is not a constant-factor underestimate for c. We could try all powers of 2 for c' within a known lower and upper bound of the value of the minimum cut, and run our algorithms for all possibilities. This is rigorous, but introduces an extra $O(\log n)$ factor in our runtimes, assuming the range of c' we try is polynomial in n. We instead show the following.

Lemma 121. *Let $p = 3(d+2)(\ln n)/(\epsilon^2\gamma c) \le 1$ as in Lemma 120, but with $\gamma \ge 6$ and $\epsilon \le 1/3$. Then if we sample each edge of the unweighted multigraph G uniformly at random with probability p, the resulting graph H has minimum cut of size less than $(d+2)(\ln n)/\epsilon^2$ with probability at least $1 - 1/n^{d+2}$.*

Proof. Consider the size of a minimum cut of G as a cut in H. Let X be a random variable denoting this size. Then $\mathbb{E}[X] = cp$. By a Chernoff bound, $\Pr[X \ge (1+\delta)cp] \le e^{-\frac{1}{3}(cp\delta)}$ for

$\delta \geq 1$. Let $(1+\delta) = \frac{\gamma}{3}$. Then

$$\begin{aligned}\Pr\left[X \geq (d+2)(\ln n)/\epsilon^2\right] &\leq e^{-\frac{1}{3}(cp(\frac{\gamma}{3}-1))} \\ &= e^{-(d+2)(\ln n)\gamma^{-1}\epsilon^{-2}(\frac{\gamma}{3}-1))} \\ &= n^{-\frac{1}{3}(d+2)\epsilon^{-2}+(d+2)\gamma^{-1}\epsilon^{-2}} \\ &\leq n^{-\frac{1}{6}(d+2)\epsilon^{-2}} \\ &< n^{-(d+2)}.\end{aligned}$$

Therefore, the minimum cut in H has size less than $(d+2)(\ln n)/\epsilon^{-2}$ with probability at least $1 - 1/n^{d+2}$. □

Lemma 121 states that if our estimate $c' = \gamma c$ satisfies $\gamma \geq 6$, the minimum cut will be at least a factor 3 smaller than $3(d+2)(\ln n)/\epsilon^2$ with high probability. Recall that with $\gamma = 1$ and therefore $c' = c$, we expect the minimum cut in H to be within a factor $(1+\epsilon)$ from $3(d+2)(\ln n)/\epsilon^2$ with high probability. Lemma 121 gives us the necessary tool to reject c' that are not a constant factor underestimate of c. We try a value for c', and if the size of the minimum cut in H is greater than $(1+\epsilon)^{-1}3(d+2)(\ln n)/\epsilon^2$, we know $c' < 6c$. Therefore we can decrease c' by a factor of 6 and rerun the tree packing algorithm. The resulting graph H must satisfy the conditions of Lemma 120, therefore the algorithm may proceed. Since our tree packing algorithms determine the minimum cut up to constant factors, this approach avoids the need of a different (or recursive!) minimum cut algorithm to run on H.

We briefly remark on the choice of known upper bound U. If the edge weights are polynomially bounded by the number of vertices, n, a simple upper bound of the sum of weights of edges attached to any single vertex will do. If we do not consider this guarantee, Karger shows [152] that the minimum weight edge w in a maximum spanning tree has the property that the minimum cut must have weight between w and $n^2 w$. Thus, setting $U = n^2 w$ gives only $O(\log n)$ values of c' to try regardless of edge weights. The choice of an upper bound U is further discussed in [59].

We now return to the issue of real-value weights in Lemma 120. This was described as a complication in [59], to which they substituted a heuristic method in order to achieve practicality. The approach we have described thus far is amenable to small constant-factor approximations. Suppose we replace G with a graph G' such that each edge weight is first normalized so the smallest weight edge has weight 1, then all edge weights are multiplied by 100 and rounded to the nearest integer. Normalizing has no effect on the relative sizes of cuts in G'. Rounding to the nearest integer when the smallest weight edge has weight at least 100 has the effect that a cut of weight x will take on a new weight in range

$[.995x, 1.005x]$. Then the original minimum cut of G corresponds to an at most 201/199-times minimum cut of G'. Now, G' can be represented as an unweighted multigraph and then sampled according to Lemma 120. In the resulting graph H, the minimum cut of G corresponds to an at most $201/199 \cdot 7/6$-times minimum cut of H with the choice $\epsilon = 1/6$. By adjusting constants throughout the rest of our approach, this shows we can treat real weighted graphs G correctly. The other issue is how to do so efficiently.

If we consider G' as an unweighted multigraph, the number of edges of G' is proportional to the weight of edges of G, which may be quite large. However, we may also consider G' as an integer-weighted graph, in which case we can sample each edge of G' by drawing from the binomial distribution with probability p and number of trials the weight of the respective edge. There are many methods to sample from the binomial distribution. One simple method that can be made efficient for our purposes is inverse transform sampling. Let X denote a random variable sampled from the binomial distribution as described. In inverse transform sampling, we draw a number u uniformly at random between 0 and 1, and then choose our sample x to be the largest such that $P(X < x) \leq u$. Instead of having to sample a number of times equal to the weight of an edge, we must only compute the probabilities of the cumulative distribution function for the binomial distribution for all possible values that may result in H. We can make this efficient with the following observation. Say the weight of the minimum cut in H is $\hat{c}$. Then a tree packing of H has value at most $\hat{c}$, and in particular for a given edge, any weight beyond $\hat{c}$ is excess capacity that cannot be used in the tree packing. It follows that capping the weight of any edge of H to the maximum size of the minimum cut in H, thus $O(\log n)$, will have no impact on the packing found. Thus, we must only compute $O(\log n)$ probabilities of the binomial distribution per edge, which can be done in total $O(\log n)$ time per edge.

The final choice is to pick a tree packing algorithm. Karger gives two options. The first is an algorithm by Gabow [103], which computes a $c/2$ packing. The second is a more general approach by Plotkin-Shmoys-Tardos [205], which can find a packing a factor $(1 + \epsilon')$ from the maximum packing, which has value in $[c/2, c]$. Karger describes the latter approach as simpler, using only minimum spanning tree computations. Although the paper [205] does not explicitly give a routine for packing spanning trees, such a procedure is explicitly given in Thorup and Karger [226], with credit given to Plotkin-Shmoys-Tardos [205] and Young [242]. This procedure also appears in Gawrychowski et al. [110]. We give the procedure in Algorithm 6 and state a version of Algorithm 6 with general epsilon in Algorithm 10.

We now give the general form of Lemma 109 with proof.

Lemma 122 ([205, 226, 242]). *Given $0 < \epsilon < 1$ and an undirected unweighted graph G with m edges, n vertices, and minimum cut c, Algorithm 10 returns a weighted packing of*

Algorithm 10 Obtain a Packing of Weight at least $(1-\epsilon)c/2$ from a Graph G

```
Let $G$ be a graph with $m$ edges and $n$ vertices.
  1. Initialize $\ell(e) \leftarrow 0$ for all edges $e$ of $G$. Initialize multiset $P \leftarrow \varnothing$. Initialize $W \leftarrow 0$.
  2. Repeat the following:
      (a) Find a minimum spanning tree $T$ with respect to $\ell(\cdot)$.
      (b) Set $\ell(e) \leftarrow \ell(e) + \epsilon^2/(3\ln m)$ for all $e \in T$. If $\ell(e) > 1$, return $W, P$.
      (c) Set $W \leftarrow W + \epsilon^2/(3\ln m)$.
      (d) Add $T$ to $P$.
```

weight at least $(1-\epsilon)c/2$ *in* $O(mc\log n)$ *time.*

Proof. On each iteration, the weight of some tree is increased by $\epsilon^2/(3\ln m)$. Since the weight of the resulting packing is bounded by c, there are at most $3c\ln m/\epsilon^2 = O(c\log n)$ iterations. The bottleneck in each iteration is the time to compute a minimum spanning tree in G. With an $O(m)$ time minimum spanning tree algorithm [154] our final time complexity is $O(mc\log n)$; an alternative way to achieve this runtime when Algorithm 10 is used in Algorithm 11 was shown in Section 8.3. Correctness is given via Thorup and Karger [226], Young [242], and Plotkin-Shmoys-Tardos [205]. $\square$

Our full procedure for obtaining $\Theta(\log n)$ spanning trees for the rest of the algorithm is given in Algorithm 7. We give a version of Algorithm 7 with general epsilons in Algorithm 11.

Algorithm 11 Obtain $\Theta(\log n)$ Spanning Trees for the 2-respect Algorithm

Let d denote the exponent in the probability of success $1 - 1/n^d$. Let $\epsilon_1, \epsilon_2, \epsilon_3 > 0$ be constants of approximation such that $f = 3/2 - (\frac{2+\epsilon_1}{2-\epsilon_1})(1+\epsilon_2)(1-\epsilon_3)^{-1} > 0$ and $(1+\epsilon_2)^{-1}(1-\epsilon_3) > 2/3$. Let $b = 3(d+2)\ln n/{\epsilon_2}^2$.

```
1. Form graph $G'$ from $G$ by first normalizing the edge weights of $G$ so the smallest
   non-zero edge weight has weight 1, then multiplying each edge weight by $\epsilon_1^{-1}$ and
   rounding to the nearest integer. Let $U$ be an upper bound for the size of the
   minimum cut of $G'$.
2. Initialize $c' \leftarrow U$. Repeat the following:
   (a) Construct $H$ in the following way: for each edge $e$ of $G'$, let $e$ have weight in
       $H$ drawn from the binomial distribution with probability $p = \min(b/c', 1)$ and
       number of trials the weight of $e$ in $G'$. Cap the weight of any edge in $H$ to at
       most $\lceil (1+\epsilon_2)12b \rceil$.
   (b) Run Algorithm 10 on $H$ with approximation $\epsilon_3$, considering an edge of weight
       $w$ as $w$ parallel edges. There are three cases:
       i. If $p = 1$, set $P$ to the packing returned and skip to step 3.
       ii. If the returned packing is of weight $\frac{1}{2}(1-\epsilon_3)(1+\epsilon_2)^{-1}b$ or greater, set
           $c' \leftarrow c'/6$ and repeat steps 2a and 2b, setting $P$ to the packing returned
           and then proceeding to step 3.
       iii. Otherwise, repeat steps 2a and 2b with $c' \leftarrow c'/2$.
3. Return $\lceil -d\ln n/\ln(1-f) \rceil$ trees sampled uniformly at random proportional to their
   weights from $P$.
```

We give the generalization of Lemma 110 for Algorithm 11 below.

Lemma 123. *Algorithm 11 returns a collection of $\Theta(\log n)$ spanning trees of G in time $O(m\log^3 n)$ such that the minimum cut of G 2-respects at least one tree in the collection with high probability.*

Proof. We first prove correctness. Consider general epsilons $\epsilon_1, \epsilon_2, \epsilon_3 > 0$, where in Algorithm 7, $\epsilon_1 = 1/100$ is the real-weight approximation, $\epsilon_2 = 1/6$ is the approximation for Lemmas 120 and 121, and $\epsilon_3 = 1/5$ is the approximation for Algorithm 6 to return a packing of size $(1-\epsilon_3)c/2$ or greater.

Suppose for a particular c' that $c' \geq 6c$, where c is the size of the minimum cut in G'. Then by Lemma 121, H will have minimum cut of size less than $b/3 = (d+2)\ln n/\epsilon_2{}^2$ with high probability. A maximum tree packing of H will have weight at most $\hat{c}$, the weight of the minimum cut in H, and thus the weight of the tree packing found by Algorithm 10 will be at most $b/3 < \frac{1}{2}(1-\epsilon_3)(1+\epsilon_2)^{-1}b$ because $(1+\epsilon_2)^{-1}(1-\epsilon_3) > 2/3$. Therefore Algorithm 11 will proceed to the next iteration with $c' \leftarrow c'/2$. Note that the overall probability of failure from any of the $O(\log n)$ iterations of this step is at most $O(\log n \cdot n^{-(d+2)}) \leq n^{-d}$ for sufficiently large n.

Now suppose Algorithm 10 returns a tree packing of weight $\frac{1}{2}(1-\epsilon_3)(1+\epsilon_2)^{-1}b$ or greater. By the above, $c' < 6c$ with high probability. If $c' \leq c$, Lemma 120 says that the weight of the minimum cut is at least $(1+\epsilon_2)^{-1}b$ with high probability, unless $p > 1$. In the latter case, this implies the weight of the minimum cut is $O(\log n)$ and there is no need to apply sampling to G'. Consider the former case. The tree packing is of weight at least $(1-\epsilon_3)$ times half the minimum cut. It follows that the tree packing will be of weight at least $\frac{1}{2}(1-\epsilon_3)(1+\epsilon_2)^{-1}b$. The consequence of this is that if a tree packing of this weight or greater is found in step 2b, in addition to the bound $c' < 6c$, we also know $c' > c/2$ with high probability, since whenever $c' \leq c$, Lemma 120 says the packing will have weight at least $\frac{1}{2}(1-\epsilon_3)(1+\epsilon_2)^{-1}b$, and we decrease c' by a factor of 2 in each iteration. Therefore, if we set $c' \leftarrow c'/6$, then in the next iteration we will have $c/12 < c' < c$.

Now consider the next iteration when the tree packing is returned. In sampling H, we only preserve weights in H up to $\lceil (1+\epsilon_2)\cdot 12b \rceil$. Since $c' > c/12$, the expected size of the minimum cut in H is at most $12b = 12 \cdot 3(d+2)\ln n/\epsilon_2{}^2$. Thus, with high probability, by Lemma 120, the size of the minimum cut in H is at most $(1+\epsilon_2)12b$, and as explained previously, we can afford to remove the capacity of any edge beyond $(1+\epsilon_2)12b$ without impacting the returned packing. Now by Lemma 119 with $\alpha \leq \frac{2+\epsilon_1}{2-\epsilon_1}(1+\epsilon_2)$ and $\beta \geq \frac{1}{2}(1-\epsilon_3)$, a fraction of at least $f = 3/2 - (\frac{2+\epsilon_1}{2-\epsilon_1})(1+\epsilon_2)(1-\epsilon_3)^{-1}$ of the trees in the packing found will 2-constrain the minimum cut of G. The probability that no tree in a sample of size t 2-constrains the minimum cut is $(1-f)^t$. Solving for t in $(1-f)^t = n^{-d}$ yields $t = -d\ln n/\ln(1-f)$. Therefore with probability at least $1 - 1/n^d$, at least one tree in the returned sample will 2-constrain the minimum cut.

Time complexity can be proven as follows. Sampling H can be done in $O(m\log n)$ time, as explained previously. Algorithm 10 runs in $O(m'\hat{c}\log^2 n)$ time using a textbook $O(m\log n)$ minimum spanning tree algorithm, where $\hat{c}$ is the value of the minimum cut in H and m' is the number of edges in H, where weighted edges are considered parallel unit weight edges. Due to the sampling procedure, $m' = O(m\log n)$. To reduce this complexity, we can either use a linear time minimum spanning tree algorithm [154] or the implementation trick given in Section 8.3. If we use the latter, we reduce the effective m'

needed in Algorithm 10 to $O(m)$. Further, in expectation, the value of the minimum cut $\hat{c}$ of H doubles in each iteration of Algorithm 11. A high probability statement can be made via an argument similar to Lemma 121. Therefore the cost of running Algorithm 10 doubles in each iteration, with the final cost being $O(m \log^3 n)$, since $\hat{c} = O(\log n)$ by Lemma 120. This is a geometric series, so the entire cost is $O(m \log^3 n)$, and so Algorithm 11 runs in $O(m \log^3 n)$ time with high probability. □

Since Algorithm 10 returns $O(\log n)$ trees, we could avoid sampling trees from the weighted packing and instead return all of them. We keep the sampling in Algorithm 11 because, depending on the constants, sampling may require less trees. Further, the above version of Algorithm 10 is more versatile in that the packing algorithm can be changed. Observe that the entire algorithm is still only correct with high probability, since we required sampling G' to construct graph H. Finally, returning all trees from Algorithm 10 does not actually allow us to relax ϵ_1, ϵ_2, or ϵ_3. The condition $f = 3/2 - (1+\epsilon_1)(1+\epsilon_2)(1-\epsilon_3)^{-1} > 0$ is satisfied for all values of α and β that guarantee at least one tree in a weighted packing of weight βc 2-constrains a cut of weight αc given by Lemma 119.

Algorithm 11 is slightly different than the approach taken by Karger [153]. In particular, Karger sparsifies edges of H to have $m' = O(n \log n)$ and replaces an $O(m \log n)$ time minimum spanning tree computation in the tree packing algorithm with an $O(m)$ one, avoiding the implementation trick of Gawrychowski et al. [110]. This gives complexity $O(n \log^3 n)$ for finding the $\Theta(\log n)$ spanning trees. However, since the remaining part of the algorithm also takes $O(m \log^3 n)$ time, we avoid these optimizations to simplify our procedures.

References

[1] Amir Abboud and Soren Dahlgaard. Popular conjectures as a barrier for dynamic planar graph algorithms. In *IEEE 57th Annual Symposium on Foundations of Computer Science*, pages 477–486, 10 2016.

[2] Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *Proceedings of the 2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, FOCS '14, pages 434–443, 2014.

[3] Amir Abboud, Virginia Vassilevska Williams, and Huacheng Yu. Matching triangles and basing hardness on an extremely popular conjecture. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015*, pages 41–50, 2015.

[4] I. Abraham, D. Durfee, I. Koutis, S. Krinninger, and R. Peng. On fully dynamic graph sparsifiers. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 335–344, Oct 2016.

[5] Georgy Adelson-Velsky and Evgenii Landis. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences*, 146:263–266, 1962.

[6] Alok Aggarwal and Prabhakar Raghavan. Deferred data structure for the nearest neighbor problem. *Information Processing Letters*, 40(3):119–122, November 1991.

[7] Hassan Aït-Kaci, Robert Boyer, Patrick Lincoln, and Roger Nasr. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(1):115–146, 1989.

[8] Brian Allen and Ian Munro. Self-organizing binary search trees. *Journal of the ACM*, 25(4):526–535, 1978.

[9] Noga Alon, Zvi Galil, and Oded Margalit. On the exponent of the all pairs shortest path problem. *J. Comput. Syst. Sci.*, 54(2):255–262, 1997.

[10] Stephen Alstrup, Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Trans. Algorithms*, 1(2):243–264, October 2005.

[11] Arne Andersson. Improving partial rebuilding by using simple balance criteria. In *Workshop on Algorithms and Data Structures*, 1989.

[12] David L. Applegate, Robert E. Bixby, Vasek Chvatal, and William J. Cook. *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)*. Princeton University Press, Princeton, NJ, USA, 2007.

[13] Sigal Ar, Gil Montag, and Ayellet Tal. Deferred, self-organizing BSP trees. *Computer Graphics Forum*, 21(3):269–278, September 2002.

[14] Sepehr Assadi, Sanjeev Khanna, Yang Li, and Val Tannen. Dynamic sketching for graph optimization problems with applications to cut-preserving sketches. In *FSTTCS*, 2015.

[15] Ahmet Arif Aydin and Kenneth M. Anderson. Incremental sorting for large dynamic data sets. In *International Conference on Big Data Computing Service and Applications*. IEEE, 2015.

[16] Mihai Bădoiu, Richard Cole, Erik D. Demaine, and John Iacono. A unified access bound on comparison-based dynamic dictionaries. *Theoretical Computer Science*, 382(2):86–96, August 2007.

[17] Jérémy Barbay. Optimal prefix free codes with partial sorting. *Algorithms*, 13(1):12, December 2019.

[18] Jérémy Barbay, Ankur Gupta, Seungbum Jo, Srinivasa Rao Satti, and Jonathan Sorenson. Theory and implementation of online multiselection algorithms. In *European Symposium on Algorithms (ESA)*, pages 109–120. Springer, 2013.

[19] Jérémy Barbay, Ankur Gupta, Srinivasa Rao Satti, and Jon Sorenson. Dynamic online multiselection in internal and external memory. In *WALCOM: Algorithms and Computation*, pages 199–209. Springer, 2015.

[20] Jérémy Barbay, Ankur Gupta, Srinivasa Rao Satti, and Jon Sorenson. Near-optimal online multiselection in internal and external memory. *Journal of Discrete Algorithms*, 36:3–17, 2016.

[21] Jérémy Barbay, Carlos Ochoa, and Srinivasa Rao Satti. Synergistic solutions on multisets. In *Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 78 of *LIPIcs*. Schloss Dagstuhl, 2017.

[22] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

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

[24] Samuel W. Bent, Daniel D. Selator, and Robert E. Tarjan. Biased search trees. *SIAM Journal on Computing*, 14(3):545–568, 1985.

[25] Jon L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 1975.

[26] Jon L. Bentley. Solutions to klee's rectangle problems. Technical report, Carnegie Mellon University, 1977.

[27] Jon L. Bentley and Andrew C. Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 1976.

[28] Jon Louis Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244–251, 1979.

[29] Nalin Bhardwaj, Antonio Molina Lovett, and Bryce Sandlund. A simple algorithm for minimum cuts in near-linear time. In *17th Scandinavian Symposium and Workshop on Algorithm Theory*, 2020.

[30] Daniel Bienstock and Clyde L. Monma. On the complexity of covering vertices by faces in a planar graph. *SIAM J. Comput.*, 17(1):53–76, February 1988.

[31] Garrett Birkhoff. Rings of sets. *Duke Mathematical Journal*, 3(3):443–454, 1937.

[32] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

[33] A. D. Booth and A. J. T. Colin. On the efficiency of a new method of dictionary construction. *Information and Control*, 3:327–334, 1960.

[34] A. Borodin, L. J. Guibas, N. A. Lynch, and A. C. Yao. Efficient searching using partial ordering. *Information Processing Letters*, 12(2):71–75, 1981.

[35] Prosenjit Bose, Jean Cardinal, John Iacono, Grigorios Koumoutsos, and Stefan Langerman. Competitive online search trees on trees. In *Symposium on Discrete Algorithms (SODA)*, pages 1878–1891. Society for Industrial and Applied Mathematics, January 2020.

[36] Prosenjit Bose, John Howat, and Pat Morin. A history of distribution-sensitive data structures. In *Space-Efficient Data Structures, Streams, and Algorithms*, pages 133–149. Springer, 2009.

[37] Prosenjit Bose, Evangelos Kranakis, Pat Morin, and Yihui Tang. Approximate range mode and range median queries. In *Proceedings of the 22nd Annual Symposium on Theoretical Aspects of Computer Science*, pages 377–388, 2005.

[38] Rodrigo A. Botafogo. Cluster analysis for hypertext systems. In *Proceedings of the 16th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '93, pages 116–125. ACM, 1993.

[39] Karl Bringmann, Fabrizio Grandoni, Barna Saha, and Virginia Williams. Truly sub-cubic algorithms for language edit distance and rna folding via fast bounded-difference min-plus product. *SIAM Journal on Computing*, 48, 07 2017.

[40] Karl Bringmann, Marvin Kunnemann, and Andre Nusser. Frechet distance under translation: Conditional hardness and an algorithm via offline dynamic grid reachability. *CoRR*, abs/1810.10982, 2018.

[41] Gerth Stølting Brodal. Worst-case efficient priority queues. In *Symposium on Discrete Algorithms (SODA)*. SIAM, 1996.

[42] Gerth Stølting Brodal. A survey on priority queues. In *Space-Efficient Data Structures, Streams, and Algorithms*, pages 150–163. Springer, 2013.

[43] Gerth Stølting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *Symposium on Discrete Algorithms (SODA)*, pages 546–554. SIAM, 2003.

[44] Gerth Stølting Brodal, Beat Gfeller, Allan Grønlund Jørgensen, and Peter Sanders. Towards optimal range medians. *Theoretical Computer Science*, 412(24):2588–2601, 2011.

[45] Gerth Stølting Brodal and Allan Grønlund Jørgensen. Data structures for range median queries. In *Proceedings of the 20th International Symposium on Algorithms and Computation*, pages 822–831, 2009.

[46] Gerth Stølting Brodal, George Lagogiannis, and Robert E. Tarjan. Strict Fibonacci heaps. In *Symposium on Theory of Computing (STOC)*. ACM, 2012.

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

[48] Mark R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM Journal on Computing*, 7(3):298–319, 1978.

[49] Yves Caseau. Efficient handling of multiple inheritance hierarchies. *ACM SIGPLAN Notices*, 28(10):271–287, 1993.

[50] Yves Caseau, Michel Habib, Lhouari Nourine, and Olivier Raynaud. Encoding of multiple inheritance hierarchies and partial orders. *Computational Intelligence*, 15(1):50–62, 1999.

[51] Parinya Chalermsook, Jittat Fakcharoenphol, and Danupon Nanongkai. A deterministic near-linear time algorithm for finding minimum cuts in planar graphs. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 828–829, 2004.

[52] Parinya Chalermsook, Mayank Goswami, László Kozma, Kurt Mehlhorn, and Thatchaphol Saranurak. Pattern-avoiding access in binary search trees. In *Symposium on Foundations of Computer Science (FOCS)*, pages 410–423. IEEE, 2015.

[53] Timothy M. Chan. Quake heaps: A simple alternative to Fibonacci heaps. In *Space-Efficient Data Structures, Streams, and Algorithms*, pages 27–32. Springer, 2009.

[54] Timothy M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. *SIAM J. Comput.*, 39(5):2075–2089, 2010.

[55] Timothy M. Chan, Stephane Durocher, Kasper Green Larsen, Jason Morrison, and Bryan T. Wilkinson. Linear-space data structures for range mode query in arrays. *Theory of Computing Systems*, 55:719–741, 2014.

[56] Timothy M. Chan, Stephane Durocher, Matthew Skala, and Bryan T. Wilkinson. Linear-space data structures for range minority query in arrays. *Algorithmica*, 72:901–913, 2015.

[57] Timothy M Chan and Konstantinos Tsakalidis. Dynamic orthogonal range searching on the ram, revisited. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 77. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[58] Timothy M. Chan and Bryan T. Wilkinson. Adaptive and approximate orthogonal range counting. *ACM Trans. Algorithms*, 12(4):45:1–45:15, 2016.

[59] Chandra S. Chekuri, Andrew V. Goldberg, David R. Karger, Matthew S. Levine, and Cliff Stein. Experimental study of minimum cut algorithms. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 324–333, 1997.

[60] Yu-Tai Ching, Kurt Mehlhorn, and Michiel H.M. Smid. Dynamic deferred data structuring. *Information Processing Letters*, 35(1):37 – 40, 1990.

[61] Richard Cole. On the dynamic finger conjecture for splay trees. part II: The proof. *SIAM Journal on Computing*, 30(1):44–85, 2000.

[62] Richard Cole, Bud Mishra, Jeanette Schmidt, and Alan Siegel. On the dynamic finger conjecture for splay trees. part I: Splay sorting $\log n$-block sequences. *SIAM Journal on Computing*, 30(1):1–43, 2000.

[63] Walter Cunto and J. Ian Munro. Average case selection. *Journal of the ACM*, 36(2):270–279, 1989.

[64] Mohit Daga, Monika Henzinger, Danupon Nanongkai, and Thatchaphol Saranurak. Distributed edge connectivity in sublinear time. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pages 343–354, 2019.

[65] Soren Dahlgaard. On the hardness of partially dynamic graph problems and connections to diameter. In *43rd International Colloquium on Automata, Languages, and Programming*, 2016.

[66] Brian A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[67] Erik D. Demaine, Dion Harmon, John Iacono, Daniel Kane, and Mihai Patrascu. The geometry of binary search trees. In *Symposium on Discrete Algorithms (SODA)*, pages 496–505. SIAM, 2009.

[68] Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Patrascu. Dynamic optimality–almost. *Siam Journal of Computing*, 37(1):240–251, 2007.

[69] Giuseppe Di Battista and Roberto Tamassia. On-line graph algorithms with spqr-trees. In *Proceedings of the seventeenth international colloquium on Automata, languages and programming*, pages 598–611, New York, NY, USA, 1990.

[70] Paul F. Dietz. Fully persistent arrays (extended array). In *Proceedings of Workshop on Algorithms and Data Structures (WADS)*, pages 67–74, 1989.

[71] Paul F. Dietz. Optimal algorithms for list indexing and subset rank. In *Workshop on Data Structures and Algorithms*, pages 39–46, 1989.

[72] David Dobkin and J. Ian Munro. Optimal time minimal space selection algorithms. *Journal of the Association for Computing Machinery*, 28(3):454–461, 1981.

[73] A. S. Douglas. Techniques for the recording of, and reference to data in a computer. *The Computer Journal*, 2:1, 1959.

[74] Adrian Dumitrescu. A selectable sloppy heap. *Algorithms*, 12(3):58, 2019.

[75] David Durfee, Yu Gao, Gramoz Goranci, and Richard Peng. Fully dynamic spectral vertex sparsifiers and applications. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC '19. ACM, 2019.

[76] David Durfee, Rasmus Kyng, John Peebles, Anup B. Rao, and Sushant Sachdeva. Sampling random spanning trees faster than matrix multiplication. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 730–742, 2017.

[77] Stefan Edelkamp, Amr Elmasry, and Jyrki Katajainen. The weak-heap data structure: Variants and applications. *Journal of Discrete Algorithms*, 16:187–205, October 2012.

[78] H. Edelsbrunner. Dynamic data structures for orthogonal intersection queries. Technical report, Tech. Univ. Graz, 1980.

[79] Hicham El-Zein, Meng He, J. Ian Munro, Yakov Nekrich, and Bryce Sandlund. On approximate range mode and range selection. In *30th International Symposium on Algorithms and Computation*, 2019.

[80] Hicham El-Zein, Meng He, J. Ian Munro, and Bryce Sandlund. Improved Time and Space Bounds for Dynamic Range Mode. In *26th Annual European Symposium on Algorithms*, volume 112, pages 25:1–25:13, 2018.

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

[82] Amr Elmasry. Pairing heaps with $O(\log \log n)$ decrease cost. In *Symposium on Discrete Algorithms (SODA)*. SIAM, 2009.

[83] Amr Elmasry, Meng He, J Ian Munro, and Patrick K Nicholson. Dynamic range majority data structures. In *International Symposium on Algorithms and Computation*, pages 150–159. Springer, 2011.

[84] David Eppstein. Offline algorithms for dynamic minimum spanning tree problems. *J. Algorithms*, 17(2):237–250, September 1994.

[85] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsification–a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, September 1997.

[86] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Thomas H. Spencer. Separator-based sparsification II: edge and vertex connectivity. In *Siam Journal on Computing*, 2006.

[87] Marcel Erné, Jobst Heitzig, and Jürgen Reinhold. On the number of distributive lattices. *Electron. J. Combin*, 9(1):23, 2002.

[88] Stefan Fafianie, Eva-Maria C. Hols, Stefan Kratsch, and Vuong Anh Quyen. Preprocessing under uncertainty: Matroid intersection. In *41st International Symposium on Mathematical Foundations of Computer Science (MFCS 2016)*, volume 58, pages 35:1–35:14, 2016.

[89] Stefan Fafianie, Stefan Kratsch, and Vuong Anh Quyen. Preprocessing under uncertainty. In *33rd Symposium on Theoretical Aspects of Computer Science (STACS 2016)*, volume 47, pages 33:1–33:13, 2016.

[90] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4):614–656, June 2003.

[91] Arash Farzan and J Ian Munro. Succinct representation of finite abelian groups. In *Proceedings of the 2006 international symposium on Symbolic and algebraic computation*, pages 87–92. ACM, 2006.

[92] Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011.

[93] Michael J Fischer and Albert R Meyer. Boolean matrix multiplication and transitive closure. In *n 12th Annual Symposium on Switching and Automata Theory*, pages 129–131. IEEE, 1971.

[94] Robert W. Floyd and Ronald L. Rivest. Expected time bounds for selection. *Communications of the ACM*, 18(3):165–172, 1975.

[95] Kyle Fox. Upper bounds for maximally greedy binary search trees. In *Lecture Notes in Computer Science*, pages 411–422. Springer Berlin Heidelberg, 2011.

[96] Gereon Frahling, Piotr Indyk, and Christian Sohler. Sampling in dynamic data streams and applications. *International Journal of Computational Geometry & Applications*, 18(1/2):3–28, 2008.

[97] Greg N. Frederickson. An optimal algorithm for selection in a min-heap. *Information and Computation*, 104(2):197–214, 1993.

[98] M. L. Fredman and D. E. Willard. Blasting through the information theoretic barrier with fusion trees. In *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*, STOC '90, pages 1–7, 1990.

[99] Michael Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the Association for Computing Machinery*, 34(3):596–615, 1987.

[100] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, June 1984.

[101] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

[102] Harold N. Gabow, Jon Louis Bentley, and Robert E. Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 135–143, 1984.

[103] H.N. Gabow. A matroid approach to finding edge connectivity and packing arborescences. *Journal of Computer and System Sciences*, 50(2):259 – 273, 1995.

[104] Travis Gagie, Meng He, and Gonzalo Navarro. Compressed dynamic range majority data structures. In *Data Compression Conference (DCC), 2017*, pages 260–269. IEEE, 2017.

[105] Travis Gagie, Simon J. Puglisi, and Andrew Turpin. Range quantile queries: Another virtue of wavelet trees. In *Proceedings of the 16th International Symposium on String Processing and Information Retrieval*, pages 1–6, 2009.

[106] François Le Gall. Powers of tensors and fast matrix multiplication. In *International Symposium on Symbolic and Algebraic Computation, ISSAC 2014, Kobe, Japan, July 23-25, 2014*, pages 296–303, 2014.

[107] François Le Gall and Florent Urrutia. Improved rectangular matrix multiplication using powers of the coppersmith-winograd tensor. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1029–1046. SIAM, 2018.

[108] Igal Galperin and Ronald L. Rivest. Scapegoat trees. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, 1993.

[109] Bernhard Ganter, Gerd Stumme, and Rudolf Wille. *Formal concept analysis: foundations and applications*, volume 3626. Springer, 2005.

[110] Paweł Gawrychowski, Shay Mozes, and Oren Weimann. Minimum cut in $O(m \log^2 n)$ time. *CoRR*, abs/1911.01145, 2019.

[111] Pawel Gawrychowski and Patrick K. Nicholson. Optimal encodings for range top- k k , selection, and min-max. In *Proceedings of the 42nd International Colloquium on Automata, Languages, and Programming*, pages 593–604, 2015.

[112] B. Geissmann and L. Gianinazzi. Cache oblivious minimum cut. In *International Conference on Algorithms and Complexity*, 2017.

[113] Barbara Geissmann and Lukas Gianinazzi. Parallel minimum cuts in near-linear work and low depth. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 1–11, 2018.

[114] Mohsen Ghaffari and Fabian Kuhn. Distributed minimum cut approximation. In *International Symposium on Distributed Computing*, pages 1–15, 2013.

[115] Mohsen Ghaffari, Krzysztof Nowicki, and Mikkel Thorup. Faster algorithms for edge connectivity via random 2-out contractions. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms*, 2020.

[116] R. E. Gomory and T. C. Hu. Multi-terminal network flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):pp. 551–570, 1961.

[117] Michael T. Goodrich and John G. Koss II. Tiered vectors: Efficient dynamic arrays for rank-based sequences. In *Workshop on Data Structures and Algorithms*, 1999.

[118] Gramoz Goranci, Monika Henzinger, and Pan Peng. The power of vertex sparsifiers in dynamic graph algorithms. In *European Symposium on Algorithms (ESA)*, pages 45:1–45:14, 2017.

[119] Gramoz Goranci, Monika Henzinger, and Pan Peng. Dynamic effective resistances and approximate schur complement on separable graphs. In *26th Annual European Symposium on Algorithms (ESA 2018)*, volume 112, pages 40:1–40:15, 2018.

[120] Gramoz Goranci, Monika Henzinger, and Thatchaphol Saranurak. Fast incremental algorithms via local sparsifiers. *CoRR*, 2018.

[121] Gramoz Goranci, Monika Henzinger, and Mikkel Thorup. Incremental exact min-cut in poly-logarithmic amortized update time. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 57. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[122] George Grätzer and Friedrich Wehrung. *Lattice theory: special topics and applications*. Springer, 2016.

[123] Mark Greve, Allan Grønlund Jørgensen, Kasper Dalgaard Larsen, and Jakob Truelsen. Cell probe lower bounds and approximations for range mode. In *International Colloquium on Automata, Languages, and Programming*, 2010.

[124] Roberto Grossi, John Iacono, Gonzalo Navarro, Rajeev Raman, and S. Srinivasa Rao. Asymptotically optimal encodings of range data structures for selection and top-k queries. *ACM Transactions on Algorithms*, 13(2):28:1–28:31, 2017.

[125] Ben Gum and Richard Lipton. Cheaper by the dozen: Batched algorithms. In *International Conference on Data Mining (ICDM)*. SIAM, 2001.

[126] Michel Habib, Raoul Medina, Lhouari Nourine, and George Steiner. Efficient algorithms on distributive lattices. *Discrete Applied Mathematics*, 110(2):169–187, 2001.

[127] Michel Habib and Lhouari Nourine. Tree structure for distributive lattices and its applications. *Theoretical Computer Science*, 165(2):391–405, 1996.

[128] Bernhard Haeupler, Siddhartha Sen, and Robert E. Tarjan. Rank-pairing heaps. *SIAM Journal on Computing*, 40(6):1463–1485, 2011.

[129] Thomas Dueholm Hansen, Haim Kaplan, Robert E. Tarjan, and Uri Zwick. Hollow heaps. *ACM Transactions on Algorithms*, 13(3):1–27, 2017.

[130] J.X. Hao and J.B. Orlin. A faster algorithm for finding the minimum cut in a directed graph. *Journal of Algorithms*, 17(3):424 – 446, 1994.

[131] Meng He, J. Ian Munro, and Patrick K. Nicholson. Dynamic range selection in linear space. In *International Symposium on Algorithms and Computation*, 2011.

[132] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing*, page 21–30, 2015.

[133] Monika Henzinger, Alexander Noe, Christian Schulz, and Darren Strash. Practical minimum cut algorithms. *J. Exp. Algorithmics*, 23:1.8:1–1.8:22, 2018.

[134] Monika Henzinger, Satish Rao, and Di Wang. Local flow partitioning for faster edge connectivity. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1919–1938, 2017.

[135] T. Hibbard. Some combinatorial properties of certain trees. *Assoc. Comp. Mach.*, 9:13, 1962.

[136] C. A. R. Hoare. Algorithm 65: Find. *Communications of the ACM*, 4(7):321–322, 1961.

[137] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC '98, pages 79–89, New York, NY, USA, 1998. ACM.

[138] Jacob Holm, Eva Rotenberg, and Mikkel Thorup. Dynamic bridge-finding in $\tilde{O}(\log^2 n)$ amortized time. In *Symposium on Discrete Algorithms (SODA)*, 2018.

[139] Jacob Holm, Eva Rotenberg, and Christian Wulff-Nilsen. Faster fully-dynamic minimum spanning forest. In Nikhil Bansal and Irene Finocchi, editors, *Algorithms - ESA 2015*, pages 742–753, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[140] John E. Hopcroft and Robert Endre Tarjan. Dividing a graph into triconnected components. *SIAM J. Comput.*, 2(3):135–158, 1973.

[141] Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, and Seth Pettie. Fully dynamic connectivity in $O(\log n(\log\log n)^2)$ amortized expected time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 510–520, 2017.

[142] John Iacono. Alternatives to splay trees with $O(\log n)$ worst-case access time. In *Symposium on Discrete Algorithms (SODA)*, pages 516–522. SIAM, 2001.

[143] John Iacono and Stefan Langerman. Weighted dynamic finger in binary search trees. In *Symposium on Discrete Algorithms (SODA)*. SIAM, 2016.

[144] Allan Grønlund Jørgensen and Kasper Green Larsen. Range selection and median: Tight cell probe lower bounds and adaptive data structures. In *Proceedings of the 22nd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 805–813, 2011.

[145] M. Jünger, G. Rinaldi, and S. Thienel. Practical performance of efficient minimum cut algorithms. *Algorithmica*, 26:172, 2000.

[146] Arthur B Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

[147] Kanela Kaligosi, Kurt Mehlhorn, J. Ian Munro, and Peter Sanders. Towards optimal multiple selection. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 103–114. Springer, 2005.

[148] Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*, pages 1131–1142, 2013.

[149] Adam Karczmarz and Jakub Łącki. Fast and simple connectivity in graph timelines. In Frank Dehne, Jorg-Rudiger Sack, and Ulrike Stege, editors, *Algorithms and Data Structures: 14th International Symposium, WADS 2015*, pages 458–469, 2015.

[150] David R. Karger. Global min-cuts in RNC, and other ramifications of a simple min-out algorithm. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, pages 21–30, 1993.

[151] David R. Karger. A randomized fully polynomial time approximation scheme for the all terminal network reliability problem. In *Proceedings of the Twenty-seventh Annual ACM Symposium on Theory of Computing*, pages 11–17, 1995.

[152] David R. Karger. Random sampling in cut, flow, and network design problems. *Mathematics of Operations Research*, 24(2), 1999.

[153] David R. Karger. Minimum cuts in near-linear time. *J. ACM*, 47(1):46–76, January 2000.

[154] David R. Karger, Philip N. Klein, and Robert E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42(2):321–328, 1995.

[155] David R. Karger and Clifford Stein. A new approach to the minimum cut problem. *J. ACM*, 43(4):601–640, 1996.

[156] David Ron Karger. *Random Sampling in Graph Optimization Problems*. PhD thesis, Stanford University, Stanford, CA, USA, 1995. UMI Order No. GAX95-16851.

[157] Richard M. Karp, Rajeev Motwani, and Prabhakar Raghavan. Deferred data structuring. *SIAM Journal on Computing*, 17(5):883–902, 1988.

[158] Ken-Ichi Kawarabayashi and Mikkel Thorup. Deterministic edge connectivity in near-linear time. *J. ACM*, 66:4:1–4:50, 2018.

[159] Krzysztof C. Kiwiel. On floyd and rivest's SELECT algorithm. *Theoretical Computer Science*, 347(1-2):214–238, November 2005.

[160] DJ Kleitman and KJ Winston. The asymptotic number of lattices. *Annals of Discrete Mathematics*, 6:243–249, 1980.

[161] Walter Klotz and Lutz Lucht. Endliche verbände. *Journal für die Reine und Angewandte Mathematik*, 247:58–68, 1971.

[162] Donald Knuth. *The Art of Computer Programming*, volume 3, Sorting and Searching. Addison-Wesley, 1973.

[163] Sergey Kopeliovich. Offline solution of connectivity and 2-edge-connectivity problems for fully dynamic graphs. Master's thesis, Saint Petersburg State University, 2012.

[164] László Kozma and Thatchaphol Saranurak. Smooth heaps and a dual view of self-adjusting data structures. *SIAM Journal on Computing*, pages STOC18–45–STOC18–93, November 2019.

[165] Andreas Krall, Jan Vitek, and R Nigel Horspool. Near optimal hierarchical encoding of types. In *European Conference on Object-Oriented Programming*, pages 128–145. Springer, 1997.

[166] Stefan Kratsch and Magnus Wahlstrom. Representative sets and irrelevant vertices: New tools for kernelization. In *Proceedings of the 2012 IEEE 53rd Annual Symposium on Foundations of Computer Science*, FOCS '12, pages 450–459, 2012.

[167] Danny Krizanc, Pat Morin, and Michiel Smid. Range mode and range median queries on lists and trees. *Nordic Journal of Computing*, 2005.

[168] François Le Gall. Faster algorithms for rectangular matrix multiplication. In *2012 IEEE 53rd annual symposium on foundations of computer science*, pages 514–523. IEEE, 2012.

[169] Caleb Levy and Robert E. Tarjan. A new path from splay to dynamic optimality. In *Symposium on Discrete Algorithms (SODA)*, pages 1311–1330. SIAM, 2019.

[170] Huan Li, Stacy Patterson, Yuhao Yi, and Zhongzhi Zhang. Maximizing the number of spanning trees in a connected graph. *CoRR*, abs/1804.02785, 2018.

[171] Huan Li and Zhongzhi Zhang. Kirchhoff index as a measure of edge centrality in weighted networks: Nearly linear time algorithms. In *Symposium on Discrete Algorithms (SODA)*, pages 2377–2396, 2018.

[172] Jakub Łącki and Piotr Sankowski. Reachability in graph timelines. *ITCS*, 2013.

[173] Grazia Lotti and Francesco Romani. On the asymptotic complexity of rectangular matrix multiplication. *Theoretical Computer Science*, 23(2):171–185, 1983.

[174] Antonio Molina Lovett and Bryce Sandlund. A simple algorithm for minimum cuts in near-linear time. *CoRR*, abs/1908.11829, 2019.

[175] David W. Matula. A linear time $2+\epsilon$ approximation algorithm for edge connectivity. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 500–504, 1993.

[176] E. M. McCreight. Efficient algorithms for enumerating intersecting intervals and rectangles. Technical report, Xerox Palo Alto Research Center, 1980.

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

[178] Kurt Mehlhorn and Stefan Näher. Bounded ordered dictionaries in $O(\log \log N)$ time and $O(n)$ space. *Information Processing Letters*, 35(4):183–189, 1990.

[179] Antonio Molina and Bryce Sandlund. personal communication.

[180] Bernard Monjardet. The presence of lattice theory in discrete problems of mathematical social sciences. Why? *Mathematical Social Sciences*, 46(2):103–144, 2003.

[181] Christian W. Mortensen and Seth Pettie. The complexity of implicit and space efficient priority queues. In *Workshop on Algorithms and Data Structures (WADS)*, pages 49–60. Springer, 2005.

[182] Sagnik Mukhopadhyay and Danupon Nanongkai. Weighted min-cut: Sequential, cut-query and streaming algorithms. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (to appear)*, 2020.

[183] J. Ian Munro, Bryce Sandlund, and Corwin Sinnamon. Space-efficient data structures for lattices. In *17th Scandinavian Symposium and Workshop on Algorithm Theory*, 2020.

[184] J Ian Munro and Corwin Sinnamon. Time and space efficient representations of distributive lattices. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 550–567. Society for Industrial and Applied Mathematics, 2018.

[185] Hiroshi Nagamochi and Toshihide Ibaraki. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM Journal on Discrete Mathematics*, 5(1):54–66, 1992.

[186] Hiroshi Nagamochi and Toshihide Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7:583–596, 1992.

[187] Danupon Nanongkai and Hsin-Hao Su. Almost-tight distributed minimum cut algorithms. In *International Symposium on Distributed Computing*, 2014.

[188] C. St.J. A. Nash-Williams. Edge-Disjoint Spanning Trees of Finite Graphs. *Journal of the London Mathematical Society*, s1-36(1):445–450, 1961.

[189] Gonzalo Navarro and Rodrigo Paredes. Quickheaps: Simple, efficient, and cache-oblivious, 2008.

[190] Gonzalo Navarro and Rodrigo Paredes. On sorting, heaps, and minimum spanning trees. *Algorithmica*, 57(4):585–620, March 2010.

[191] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2015.

[192] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, STOC '72, pages 137–142, New York, NY, USA, 1972. ACM.

[193] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[194] Mark H Overmars. *The design of dynamic data structures*, volume 156. Springer Science & Business Media, 1983.

[195] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[196] Rodrigo Parades and Gonzalo Navarro. Optimal incremental sorting. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, January 2006.

[197] Mihai Patracscu and Erik D. Demaine. Lower bounds for dynamic connectivity. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, STOC '04, pages 546–553, New York, NY, USA, 2004. ACM.

[198] M. Patrascu. Succincter. In *2008 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 305–313, 2008.

[199] Mihai Pătraşcu. Unifying the landscape of cell-probe lower bounds. *SIAM Journal on Computing*, 40(3):827–847, 2011.

[200] Mihai Pătraşcu and Emanuele Viola. Cell-probe lower bounds for succinct partial sums. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 117–122. Society for Industrial and Applied Mathematics, 2010.

[201] Richard Peng, Bryce Sandlund, and Daniel D. Sleator. Optimal offline dynamic 2,3-edge/vertex connectivity. In *Algorithms and Data Structures Symposium*, 2019.

[202] Richard Peng, Bryce Sandlund, and Daniel Dominic Sleator. Offline dynamic higher connectivity. *CoRR*, abs/1708.03812, 2017.

[203] Holger Petersen. Improved bounds for range mode and range median queries. In *Proceedings of the 34th Conference on Current Trends in Theory and Practice of Computer Science*, pages 418–423, 2008.

[204] Holger Petersen and Szymon Grabowski. Range mode and range median queries in constant time and sub-quadratic space. *Information Processing Letters*, 109(4):225–228, 2009.

[205] S. A. Plotkin, D. B. Shmoys, and E. Tardos. Fast approximation algorithms for fractional packing and covering problems. In *Proceedings 32nd Annual Symposium of Foundations of Computer Science*, pages 495–504, 1991.

[206] Mihai Pătraşcu. Towards polynomial lower bounds for dynamic problems. In *Proceedings of the forty-second annual ACM symposium on Theory of computing*, pages 603–610, 2010.

[207] Aparna Ramanathan and Charles J. Colbourn. Counting almost minimum cutsets with reliability applications. *Mathematical Programming*, 39:253–261, 1987.

[208] Erik Regla and Rodrigo Paredes. Worst-case optimal incremental sorting. In *Conference of the Chilean Computer Science Society (SCCC)*. IEEE, November 2015.

[209] Bryce Sandlund and Sebastian Wild. Lazy search trees. In *Proceedings of the 61st Annual Symposium on Foundations of Computer Science*, 2020.

[210] Bryce Sandlund and Yinzhan Xu. Faster dynamic range mode. In *47th International Colloquium on Automata, Languages and Programming*, 2020.

[211] Robert Sedgewick. Quicksort with equal keys. *SIAM Journal on Computing*, 6(2):240–267, 1977.

[212] Robert Sedgewick. Implementing Quicksort programs. *Communications of the ACM*, 21(10):847–857, 1978.

[213] Raimund Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *J. Comput. Syst. Sci.*, 51(3):400–403, 1995.

[214] Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 1996.

[215] Zhong Shao, John H. Reppy, and Andrew W. Appel. Unrolling lists. In *Proceedings of the 1994 ACM conference on LISP and functional programming*, 1994.

[216] Avi Shoshan and Uri Zwick. All pairs shortest paths in undirected graphs with integer weights. In *40th Annual IEEE Symposium on Foundations of Computer Science*, pages 605–615, 1999.

[217] Daniel D. Sleator and Robert E. Tarjan. Self-adjusting binary seach trees. *Journal of the ACM*, 32(3):652–686, 1985.

[218] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362 – 391, 1983.

[219] Michiel Smid. *Dynamic Data Structures on Multiple Storage Media.* PhD thesis, University of Amsterdam, 1989.

[220] Mechthild Stoer and Frank Wagner. A simple min-cut algorithm. *J. ACM*, 44(4):585–591, 1997.

[221] Maurizio Talamo and Paola Vocca. Fast lattice browsing on sparse representation. In *Orders, Algorithms, and Applications*, pages 186–204. Springer, 1994.

[222] Maurizio Talamo and Paola Vocca. A data structure for lattice representation. *Theoretical Computer Science*, 175(2):373–392, 1997.

[223] Maurizio Talamo and Paola Vocca. An efficient data structure for lattice operations. *SIAM journal on computing*, 28(5):1783–1805, 1999.

[224] Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2), 1985.

[225] Robert E. Tarjan and Renato F. Werneck. Self-adjusting top trees. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 813–822, 2005.

[226] Mikkel Thorup. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, STOC '00, pages 343–350, New York, NY, USA, 2000. ACM.

[227] Mikkel Thorup. Compact oracles for reachability and approximate distances in planar digraphs. *Journal of the ACM (JACM)*, 51(6):993–1024, 2004.

[228] Mikkel Thorup. Fully-dynamic min-cut. *Combinatorica*, 27(1):91–127, February 2007.

[229] Yung H Tsin. Yet another optimal algorithm for 3-edge-connectivity. *Journal of Discrete Algorithms*, 7(1):130–146, 2009.

[230] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

[231] Virginia Vassilevska Williams. Multiplying matrices faster than Coppersmith-Winograd. In *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA, May 19 - 22, 2012*, pages 887–898, 2012.

[232] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.

[233] Rene Weiskircher. *New Applications of SPQR-Trees in Graph Drawing*. PhD thesis, Universität des Saarlandes, 2002.

[234] Robert E. Wilber. Lower bounds for accessing binary search trees with rotations. *Siam Journal of Computing*, 18(1):56–69, 1989.

[235] Rudolf Wille. Restructuring lattice theory: an approach based on hierarchies of concepts. In *Ordered sets*, pages 445–470. Springer, 1982.

[236] J. W. J. Williams. Algorithm 232 - heapsort. *Communications of the ACM*, 7(6):347–348, 1964.

[237] Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *Proceedings of the fourty-fourth annual symposium on theory of computing*, 2012.

[238] Virginia Vassilevska Williams and Yinzhan Xu. Truly subcubic min-plus product for less structured matrices, with applications. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms*, 2020.

[239] P. F. Windley. Trees, forests, and rearranging. *The Computer Journal*, 3:84, 1960.

[240] Kenneth James Winston. *Asymptotic analysis of lattices and tournament score vectors.* PhD thesis, Massachusetts Institute of Technology, 1979.

[241] Christian Wulff-Nilsen. Fully-dynamic minimum spanning forest with improved worst-case update time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC*, pages 1130–1143, 2017.

[242] Neal E. Young. Randomized rounding without solving the linear program. In *Proceedings of the 6th Annual Symposium on Discrete Algorithms.*, 1995.

[243] Raphael Yuster. Efficient algorithms on sets of permutations, dominance, and real-weighted apsp. In *20th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 950–957, 2009.

[244] Uri Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *J. ACM*, 49(3):289–317, 2002.