Uniquely Represented Data Structures with Applications to Privacy

Daniel Golovin

CMU-CS-08-135

August 2008

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Guy E. Blelloch, Chair
Gary L. Miller
R. Ravi
Jon M. Kleinberg, Cornell University

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by the National Science Foundation under ITR grants CCR-0122581 (The Aladdin Center) and IIS-0121678.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government, or any other entity.

Keywords: Unique Representation, Canonical Forms, History Independence, Oblivious Data Structures, Privacy, Forensics, Data Security

Abstract

In a typical application storing some data, if the memory representations of the internal data structures are inspected, they may leave significant clues to the past use of the application. For example, a data structure with lazy deletions might retain an object that the user believes was deleted long ago; this is problematic in environments requiring high security or strict privacy guarantees. We can eliminate such problems entirely by demanding that a data structure implementation store *exactly* the information specified by an abstract data type (ADT), and nothing more. This property is sometimes called *strong history independence*. To attain it, it is often necessary and always sufficient to ensure the data structure is *uniquely represented*. That is, any two sequences of operations which bring the ADT to the same logical state will cause the implementation to generate the same memory representation. This observation begs the following question.

> *For each abstract data type, what is the added cost for uniquely represented implementations over their conventional counterparts, in terms of time, space, and randomness?*

In this dissertation, we will answer this question for several important abstract data types, and argue that the overhead for unique representation is sufficiently low to warrant its widespread use in high security and high privacy environments. Towards this end, we provide the theoretical foundation for the development of efficient uniquely represented systems that provably store exactly the information their designs specify, and nothing more.

Thesis Committee

Guy E. Blelloch (Chair)
Professor of Computer Science
Carnegie Mellon University

Gary L. Miller
Professor of Computer Science
Carnegie Mellon University

R. Ravi
Professor of Operations Research and Computer Science
Carnegie Mellon University

Jon M. Kleinberg
Professor of Computer Science
Cornell University

Acknowledgments

It takes a special environment to cultivate great scientific research and researchers, and I count myself very fortunate to have found such an environment at Carnegie Mellon. It has been a real privilege to work with many eminent faculty and outstanding colleagues, and I owe them many thanks.

First and foremost I want to thank my advisor, Guy Blelloch. He gave me nearly total independence to pursue whatever research projects interested me, to the point where I more or less acted as a free agent within the computer science department, yet was willing "to roll up his sleeves" and join me in working on tricky problems encountered during our research together. This thesis has benefited greatly from his contributions. Guy also taught me the importance of "thinking big," optimism, and being fearless in research. I also want to thank my other committee members, R. Ravi, Gary Miller, and Jon Kleinberg, for their encouragement and thoughtful insight. Over the years I have noticed that Ravi's advice is uncannily good; often I have sought it on a wide range of topics, yet I have never once regretted taking it. Gary always tried to ensure that research and teaching were not only stimulating but also fun. Jon introduced me to the intellectually rigorous parts of computer science when I was an undergraduate at Cornell, and is largely responsible for my entering the field of computer science in the first place. Together with Éva Tardos, he also introduced me to research in computer science and his example strongly influenced my conception of an ideal scholar. Thanks also to Bruce Maggs, for his unwaivering encouragement, great advice, and amusing jokes.

I also want to thank those coauthors of mine not mentioned above: Konstantin Andreev, Charlie Garrod, Vineet Goyal, Anupam Gupta, Amit Kumar, Adam Meyerson, Viswanath Nagarajan, Florin Oprea, Michael Reiter, Mohit Singh, Stephen F. Smith, Matt Streeter, Kanat Tangwongsan, and Virginia Vassilevska.

Besides those mentioned above, there were several faculty that helped me at various times during my graduate study. Particular thanks go to Stephen Brookes,

Ziv Bar-Joseph, Avrim Blum, Manuel Blum, Alan Frieze, Mor Harchol-Balter, Bob Harper, Tuomas Sandholm, and Danny Sleator.

Thanks also to my friends and colleagues who helped make my time at Carnegie Mellon enjoyable. Neel Krishnaswami, Vyas Sekar, and Matt Streeter proved excellent conversationalists on a *wide* range of topics. Ronit Slyper and Jim McCann seem to keep their lives, like the graphics lab in which they work, full of fun toys and cool ideas. Daniel Leeds kept me wondering on the moral implications of recent results in neuroscience ("If the amygdala is damaged in this particular way, the patient loses all empathy for other human beings. Hmmm."). Mike Gerber, Kevin Bowers, and Paul Massey kept me busy on the racquetball court. Thanks also to David Abraham, Michael Ashley-Rollman, Nina Balcan, David Brumley, Hubert Chan, Shuchi Chawla, Vince Conitzer, Liz Crawford, Michael De Rosa, Jon Derryberry, Kedar Dhamdhere, Mike Dinitz, Abie Flaxman, Deepak Garg, Andrew Gilpin, Varun Gupta, Benoît Hudson, Yan Ke, Ryan Kelly, Lea Kissner, Ioannis Koutis, Katrina Ligett, Sean McLaughlin, Viswanath Nagarajan, Sandeep Pandey, Harald Räcke, Maayan Roth, Don Sheehy, Elaine Runting Shi, Mohit Singh, Amitabh Sinha, Srinath Sridhar, Joshua Sunshine, Justin Weisz, Adam Wierman, Ryan Williams, Shan Leung Maverick Woo, Noam Zeilberger, and Shuheng Zhou. Thanks also to the Gerbers, Schusters, Fischers, Kleiners, Levaris, Smalls, Tewners, Wassermans, and Hoexters, for helping to make Pittsburgh feel like home.

This research was sponsored by the National Science Foundation under ITR grants CCR-0122581 and IIS-0121678. I would like to thank the NSF and the American taxpayers for their patronage, and I will try to ensure that it redounds to their benefit. (The next phase of my plan to achieve this goal is to move into a significantly higher tax bracket.)

Carnegie Mellon's computer science department provides a fantastic environment in which to conduct research. Thanks to Sharon Burks, Deb Cavlovich, Catherine Copetas, Sophie Park, and the whole jovial and efficient administrative staff for making sure I and the other graduate students could focus on our research and education.

Another interesting feature of the department is its culture, which often takes a playful stance towards the unique building in which it is currently housed, Wean hall. Built in the brutalist architectural style, Wean is a gigantic poured concrete structure that could benefit from some interesting decorations, which the students tend to provide. As I was writing this dissertation some new decorations happened to appear in random locations throughout the building – anonymous poems concerning Wean and its inhabitants. Since the department is scheduled to move into

new digs at the (currently under construction) Gates-Hillman complex within a year or so, I decided to reproduce one of these poems for posterity, in homage to this fun Carnegie Mellon tradition. See Figure 1. Thanks, anonymous whimsically-mischievous students.

Shall I compare thee to the Parthenon?
Thou art more sturdy and more apt to last;
Athena's temple stands as time wears on,
But you will withstand time AND nuclear blasts.

Or Fallingwater? If it stays intact,
It is a wondrous cantilevered feat.
And yet, I find its beauty too abstract;
Yours is more elegant, and more concrete.

The Sistine Chapel? That I can dispatch –
Despite the ceiling o'er its sacred halls;
For Michelangelo's great hand can't match
The bottlecaps in thy more holey walls.

Of all the arthitecture I have seen
None can compare to you, O mighty Wean!

– Anonymous

Figure 1: An ode to Wean Hall. This was poem #18 of the many anonymous poems that spontaneously appeared throughout the building in 2008.

Finally, I want to thank my family for all the love and support they have provided over the years. Though it's a bit of a cliché, my parents always thought I could do anything I put my mind to. My wife Jennifer provided plenty of love, brownies, and a sympathetic ear. She also did her best to make sure I had a healthy work-life balance, which is not an easy thing to maintain in a very competitive field. This thesis is dedicated to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Consider the following *secure redaction* problem: an organization possesses a classified version of some document. The organization wants to remove certain sensitive information from the document to create an unclassified version of it which will be released to the public. The unclassified version will then be analyzed by powerful adversaries (henceforth called "observers") that will try to extract all the information they can from it. You are tasked with preparing the unclassified version of the document. How can you prepare it so as to guarantee that the adversaries cannot learn any sensitive information?

This question is of some practical importance. Hartline *et al.* [HHM+05] describe an example of a CIA document that was released by the New York Times in 2000 as a PDF file with classified information redacted by overlaying black boxes. Unfortunately the overlays could easily be removed revealing key information about the CIA's role in the 1953 overthrow of the Iranian Government. In 2005 the US military released a PDF report on the accidental shooting of Italian intelligence agent Nicola Calipari in Iraq, again with classified information redacted by overlaying black boxes. Again the overlays could be removed, and among the information revealed was the name of the US military shooter, Mario Lozano. In 2007 the Fédération Internationale de l'Automobile leaked confidential technical information about the McLaren and Ferrari racing cars in much the same way. These are just a few prominent cases among many recent examples of such leaks.

The secure redaction problem is part of a more general problem. Computer users on a typical system leave significant clues to their recent activities, in the form of logs, unflushed buffers, files marked for deletion but not yet deleted, and so on. This can have significant security implications. To address these concerns,

the notion of *history independent* data structures was devised. Roughly, a data structure is history independent if an observer with access to the underlying hardware of the system (i.e., the memory layout of the data structure) can learn no more information than a legitimate user accessing the data structure via its standard interface. Intuitively, this property is called history independence because the memory layout of the data structure will in general be a function of the historical sequence of operations performed on the system, and so any sensitive information that is leaked can be thought of as containing some information about this historical sequence of operations.

The most stringent form of history independence is called *strong history independence* (SHI), which ensures that an observer with access to the memory layout of the data structure at several different points in time can learn no more information than a legitimate user accessing the data structure via its standard interface at those points in time. In other words, strong history independence guarantees that an abstraction provided by an interface cannot be violated by inspecting its physical implementation in a machine.

Note that this notion of history independence is still underspecified. To operationalize it we must develop a formal notion of what information the abstraction stores and what information the physical system stores. Such a formalism allows us to relate each concrete state of a machine (e.g., a sequence of bits stored on a disk drive) to abstract information (e.g., text, images, or music). We use the notion of *unique representation* as a concrete implementation of strong history independence. Unique representation is a slightly stronger condition than strong history independence, and is defined formally in Chapter 2. Informally, a data structure is said to be *uniquely represented* if its concrete state is uniquely determined by

1. the abstract information that the data structure is intended to be currently storing (which determines the behavior of the data structure under its standard interface), and

2. a specification of the machine that is running the data structure and thus providing the concrete representation of it.

Thus uniquely represented data structures have canonical representations on any fixed machine[1]. For example, a uniquely represented solution to the secure redac-

[1]If the machine specification includes a random bit string, and we think of this random bit string as a random variable, then it is more accurate to say that uniquely represented data structures have canonical representations up to randomness.

tion problem would specify a unique file (i.e., bit string) for each abstract document that can be created (as specified by some abstract document data type). Thus, any editing history resulting in some fixed abstract document results in the same file, so that file provably contains no information about the editing history, and in particular provably contains no sensitive information that was previously present but was deleted.

Uniquely represented data structures thus provide a principled solution to the secure redaction problem. More generally, they provide a principled way to build computer systems that provably store *exactly* the information specified in their designs. Of course, any correctly implemented system must store at least the information specified in its design. However, until very recently it was not known how to construct efficient systems that store no extra information beyond that required by their designs. Nor was it known whether efficient systems with such a guarantee were even possible. More fundamentally, almost nothing was known about the complexity of many important uniquely represented data structures in a RAM model of computation, including such fundamental data structures as queues, linked lists, heaps, hash tables, and binary search trees. In this dissertation, we will resolve these questions for a wide range of data structures, and show that the overhead for devising uniquely represented data structures is extremely low for many fundamental abstract data types. We will provide the first efficient uniquely represented implementations for several important abstract data types, including hash tables, linked lists, queues, binary search trees, heaps, and many others, thus paving the way for a wide variety of efficient uniquely represented systems.

We will also provide the first lower bounds proving a complexity gap between uniquely represented and conventional implementations of some natural abstract data types in the RAM model, namely union-find and meldable heaps. For example, in stark contrast to the other abstract data types investigated in this dissertation, taking the union of two sets in a union-find instance requires expected $\Omega(\frac{n}{\log n})$ time for any uniquely represented implementation, whereas a simple conventional implementation performs unions and finds in $O(\log n)$ time. The discovery of fundamental barriers to efficient uniquely represented implementations for these abstract data types accentuates the positive results discussed above. Finally, we will provide a rigorous way to specify history *dependence* in Chapter 6, and initiate the study of the tradeoffs between efficiency and history dependence.

1.1 Motivation: Theory and Applications

Uniquely represented data structures are interesting for many reasons, both theoretically and practically. We begin with potential applications and then discuss developments in theoretical computer science related to unique representation.

1.1.1 Applications

Efficient uniquely represented data structures can be applied to a wide variety of problems. In this section we discuss their applications, including some speculative ones.

History Independence and Privacy in Computation. Perhaps the most natural application of uniquely represented data structures is the design of strongly history-independent data structures. Uniquely represented data structures provide the strongest possible guarantees on history-independence; they store the absolute minimum amount of information necessary to be correct implementations. Here are some concrete examples of applications.

- **Filesystems** that have the property that deleting a file *provably leaves no trace whatsoever* that it ever existed, and reveals nothing about what order files where created or last modified or last accessed, unless this is part of the interface.
- **Databases** storing sensitive information (e.g., medical records) that provably reveal nothing about the order of record insertions or records that have been deleted, nor retain any evidence of previous queries.
- **Voting Machines** that provably store only the set of participating voters, and nothing about the order in which they voted. Retaining information about the order in which votes were cast might enable privacy violations, and some voting protocols (e.g., ThreeBallot and VAV [RS07]) require that the order of votes cast not be revealed.
- **Web Browsers** that support secure browsing sessions after which the browser reverts to the *precise* state it was in before the session began.

- **Advanced File Formats** that maintain search indices or other data structures embedded in the file itself to speed up certain operations on the file, yet are uniquely represented and thus provably store only the information specified by their creator via a well specified interface.

There are various *ad hoc* partial solutions to each of these problems that would address most of the risk associated with storing undesirable extra information. However, as systems and file formats grow more complex the need for a principled solution to the problem, such as that provided by uniquely represented data structures, will become more and more apparent. For example, already many document, spreadsheet, and presentation file formats are based on the extensible markup language (XML) and incorporate trees. It seems reasonable to assume that the current trend towards more complex file formats incorporating various data structures will continue. This trend is likely to further exacerbate the problem of sensitive information leaks due to failed or incomplete attempts at redaction, and *ad hoc* solutions are unlikely to scale well with this complexity.

It should be noted that because uniquely represented data structures are strongly history independent, no attack on a previously uncompromised system will reveal information that cannot be inferred by a user with access to the legitimate interface immediately before the attack. For example, in a conventional filesystem timing attacks might be used to extract information about previously deleted files, even if the actual disk contents are not revealed to the attacker. In a uniquely represented filesystem this is impossible; the timing of any operation depends only on the machine state, and is thus history independent.

Simplifying the Debugging and Verification of Parallel Computations. Parallel programs are notoriously hard to debug, in large part because execution order of the various threads of the program are not deterministic, but rather may change with each execution. Uniquely represented data structures can help, by significantly reducing the (possibly distributed) machine state's dependence on the execution order of various threads. For example, consider a parallel computation that runs in stages, and in each stage computes a well defined logical result. More concretely, the computation might be a divide-and-conquer algorithm and the stages might consist of either breaking up large problems into smaller subproblems or combining the solutions to several subproblems into solutions to larger subproblems. The computation must generate a particular set of logical results at the end of each stage; if uniquely represented data structures are used, this implies that

the computation must reach the unique machine state encoding that set of logical results, independently of how the execution threads were scheduled during that stage (or in previous stages). This can dramatically reduce the number of computation traces that are possible, and allows the debugging to be done stage by stage.

Certifying Parallel Computations. The remarks in the previous paragraph also suggest the following possibility. If a parallel computation running in stages must go through a particular sequence of machine states, one per stage, independent of the thread scheduling, then we can produce a certificate that we performed the computation by signing each machine state in this sequence. If called upon to prove that we ran the computation, we simply rerun the computation, let a judge sign the machine state at the end of each stage, and have the judge verify that these signatures match the previously generated ones. This may require significantly less space than other approaches which must take into account how the various threads are scheduled.

Fast Equality Testing. Often a computation will test two concrete objects x and y for equality. Typically, the notion of equality employed will be equality of the abstract objects that x and y represent. For example, we might define two ordered sets to be equal if they contain the same elements. Testing for this kind of equality then requires us to traverse x and y. Using uniquely represented data structures, equality testing takes time linear in the space used to store the object. Moreover, the hash-consing technique discussed in Section 5.2 allows probabilistic equality testing in *constant* time for some objects, with a small probability of error. Having a canonical form for various data structures may have applications speeding up theorem provers and verifiers, as was the case with ordered binary decision diagrams.

1.1.2 Theoretical Connections

The investigation of unique/canonical representations or normal forms for various objects is pervasive in mathematics, and developing such representations often results in important insights. Typically this due to the fact that the canonical form preserves and highlights certain properties of the object that are deemed important while eliminating other properties. For example, in linear algebra every normal

matrix over the complex numbers can be converted into a diagonal matrix normal form (which is canonical up to reordering). That this normal form exists is a consequence of the spectral theorem [HK71], and it very clearly highlights the eigenvalues of the matrix. Eliminating unimportant or undesirable information from the representation can have computational advantages as well. For example, the development of Ordered Binary Decision Diagrams [Bry86], a unique representation for boolean functions, was a crucial enabling factor in the development of model checking[2]. The question of "what is the time and space complexity of uniquely represented implementations of many fundamental data structures?" is thus a natural one. As a bonus, unique representation may simplify proofs that data structures have other properties. For example, we use this property to our advantage when bounding the running time of the uniquely represented hash table of Section 3.1.

DFA Minimization. As we shall see in Section 1.3, the study of uniquely represented data structures goes back to the 1970s. However, the goal of representing all equivalent logical states in a computation by one machine state is evident as far back as the 1950s in the work on minimization of deterministic finite automata (DFAs) by Huffman [Huf54], Moore [Moo56], Nerode [Ner58], and Hopcroft [Hop71], among others. In this work, the goal is to find the DFA with the smallest number of states representing some regular language R, where R may be represented as another DFA. The connection with unique representation is perhaps most evident in the Myhill-Nerode theorem [Myh57, Ner58] (see [Koz97] for a treatment), which, among other things, shows how to construct the unique minimal deterministic finite automata for a regular language R using the coarsest right congruence refining R. This work can be thought of as deriving uniquely represented data structures for an abstract data type that allows the user to provide a character, and outputs accept or reject based on whether the complete sequence of characters provided so far is in some fixed regular language R. The abstract data type is completely specified by R and an input alphabet, but must be implemented as a DFA. The machine state is then merely a state of the DFA. The logical state can be modeled as a function f from strings to $\{\mathsf{accept}, \mathsf{reject}\}$ such that $f(x)$ is the output of the abstract data type if the user provides x as a sequence of characters. Each machine state q will encode such a function f_q in a natural way: $f_q(x)$

[2]Model checking is an important verification technology. Recently the Association for Computing Machinery (ACM) recognized the value of model checking by awarding its primary developers the 2007 A.M. Turing Award.

is computed by starting from state q, executing the DFA's transition function on x, and outputting accept if and only if the final state is an accepting state. The unique minimal DFA for the regular language R ensures that all states q encode distinct functions f_q, and thus guarantees that each logical state is uniquely represented.

Ordered Binary Decision Diagrams. Binary Decision Diagrams are directed acyclic graphs used to represent boolean formulae. Bryant [Bry86] introduced a particularly useful variant, called Ordered Binary Decision Diagrams (OBDDs), together with a set of procedures for manipulating them. The OBDD representation of a boolean formula is unique up to isomorphism; two OBDDs for the same formula will be isomorphic. This makes functional equivalence easy to test. Fast equality testing is related to unique representation, and the technique used in OBDDs of sharing common subgraphs is reminiscent of the hash-consing technique discussed in Section 5.2. Ken McMillan applied OBDDs to model checking, and developed Symbolic Model Checking, a widely used verification technique.

Dynamization and Incremental Computation. *Dynamization* is the process of converting static algorithms that expect all of their input up front to dynamic algorithms that support operations on dynamically changing inputs (see [ABH$^+$04, Aca05] and references therein). For example, a static sorting algorithm simply outputs an input set of numbers in sorted order, whereas a dynamic sorting algorithm might allow the user to add and delete elements, and also print the stored elements in sorted order at any time. The problem of automatically dynamizing static algorithms is also known by the name *incremental computation* [Pug88]. The problem can be restated as follows. Suppose we have already performed some expensive computation on some input, and then the input changes slightly. Rather than redo the computation from scratch, to what extent can we use the work we have already done on the old input to speed up the computation on the new input? This clearly useful in many practical situations, such as running expensive simulations (e.g., of the global climate) and testing multiple related scenarios as input (e.g., what happens if we alter carbon dioxide emissions by Δ for various values of Δ). Typically, the emphasis is on developing programming language tools to automatically obtain such speed ups.

Effective incremental computation involves "rolling back" the computation when the input changes, and then propagating the effects of the change in input forward. This is typically done in such a way that the result is the same as running the static algorithm on the entire input provided up front. To do incremental computation

in some other way would in some sense be akin to designing a dynamic algorithm for the problem and would likely require additional insight into the problem. It is unclear how this could be done automatically. Thus incremental computation schemes tend to produce the output of the static algorithm when run on the entire (current) input. This implies the result is strongly history independent. If the data structures used by the machinery set up to support incremental computation are strongly history independent, then the entire data structure will be strongly history independent as well. Indeed, techniques related to incremental computation are often useful in the design of strongly history independent data structures. Some hints of this can be seen in Section 5.10, where we obtain a uniquely represented solution to the dynamic 2-D convex hull problem by modifying a data structure of Overmars and van Leeuwen [OvL81] that makes use of a dynamization technique that Overmars developed [Ove81, Ove83, Ove87]. Interestingly, uniquely represented data structures are also helpful in the construction of efficient schemes for incremental computation, as Pugh discusses in his thesis [Pug88].

The connection between incremental computation and history independence was noted by Acar *et al.* [ABH$^+$04], whose approach to incremental computation (i.e., automatic dynamization) has this property of outputting the result of the static algorithm run on the current input. The running time of their algorithms depend on the *trace stability* of the problem, which is a measure of how much the computation preformed by the static algorithm changes as the input is changed. We provide efficient strongly history independent versions of all of the data structures needed by the change propagation algorithm of Acar *et al.* for incremental computation. This implies that if there exists a static algorithm for a problem with trace stability $f(n)$ (in the restricted RAM computational model in [ABH$^+$04]) then there exists a strongly history independent solution for the problem with running time $O(f(n) \log f(n))$ (see Theorem 4.1 of [ABH$^+$04], reproduced as Theorem 26 on page 149). Since for virtually all natural abstract data types strong history independence implies unique representation, we can infer that in the computational model in [ABH$^+$04] efficient incremental computation (via any approach that outputs the solution the static algorithm would have if given the entire input up front) is only possible if efficient uniquely represented data structures for the problem exist. It remains an interesting open question whether or not a similar result holds for the RAM model of computation.

1.2 Outline and Overview of Our Contributions

Chapter 1: Introduction

This chapter includes the introduction, motivations, and related work on uniquely represented and history independent data structures.

Chapter 2: Background

In Chapter 2 we provide formal definitions, models, and notations used throughout this dissertation.

Chapter 3: Hashing

Hashing and Memory Allocation. Efficient uniquely represented memory allocation is perhaps the most fundamental problem facing the would-be designer of uniquely represented data structures. Often the development of a uniquely represented data structure can be partitioned into two tasks, namely, creating an oblivious data structure [Mic97], and mapping that data structure onto a RAM in a uniquely represented way. The latter problem essentially requires a uniquely represented memory allocator. Since such a memory allocator can be readily built from a uniquely represented hash table supporting deletions, such hash tables are particularly important.

With this in mind, we have developed a framework for constructing uniquely represented hash tables using a variety of open address hashing schemes, which appears in Chapter 3. This framework allows us to exploit a recent breakthrough result of Pagh *et al.* [PPR07], who showed that linear probing with 5-universal hash functions yields expected $O(1 + 1/(1 - \alpha)^3)$ time operations, where α is the load of the hash table. Specifically, using linear probing with a 5-universal hash function we obtain a uniquely represented hash table that stores $n := \alpha p$ keys in p slots of space such that the expected time to perform any search, insertion, or deletion is $O(1 + 1/(1 - \alpha)^3)$. This framework also reveals a subtle connection between history independent hashing and the Gale-Shapley stable marriage algorithm [GS62], which may be of independent interest.

Dynamic Perfect Hashing. Dynamic perfect hashing is the problem of maintaining a dynamic mapping from a set of keys to data items such that lookups are supported in $O(1)$ *worst-case* time. In previous work there are two approaches that yield a hash table with $O(1)$ worst-case time for lookups and $O(1)$ amortized expected time for insertions and deletions, while using space linear in the number of keys. The first approach is due to Dietzfelbinger *et al.* [DKM$^+$94], and is an extension of the scheme of Fredman *et al.* [FKS84]. The second approach, devised by Pagh and Rodler [PR04], is quite different, and is called *cuckoo hashing*. In this dissertation we will present a third way to achieve identical time and space bounds while maintaining unique representation, assuming our machine has access to a large sequence of random bits. See Section 3.2 for details. Unfortunately, due to the nature of the unique representation property, we cannot sample random bits "on demand." On the other hand, our approach is novel and relatively simple. Previously the only history independent hash table with these performance guarantees was a WHI hash table due to Naor and Teague [NT01], who built on the work in [DKM$^+$94].

Uniquely represented data structures with worst-case guarantees, such as our hash table, often require a potentially exponential amount of randomness, however they serve as the basis for more practical data structures that either

(i) Retain the running times but make the data structure uniquely represented with high probability and weakly history independent with certainty, or

(ii) Retain unique representation but replace worst-case time guarantees with "with high probability" time guarantees.

In the case of dynamic perfect hashing, these practical variants require only $O(n^\delta)$ random bits for any constant $\delta > 0$, and, in case (*i*), the ability to sample random bits as needed. See Section 3.2.2 for details.

Chapter 4: Basic Data Structures

In Chapter 4 we cover arrays, stacks, linked lists, queues, binary search trees, and heaps. Except for heaps, we match the running times of the best conventional versions up to constant factors, ignoring the fact that the running time guarantees for the uniquely represented versions are in expectation rather than worst-case. Our uniquely represented binary search tree is based on randomized

treaps [Vui80, SA96], which play a prominent role in many constructions throughout this dissertation. In the case of heaps, we provide a uniquely represented version that has good performance for most operations, but takes linear time to perform merges. In contrast, Fibonacci heaps support amortized constant time merges. However, in Chapter 7 we show that any uniquely represented heap must take expected $\Omega(n/\log n)$ time to merge two heaps of size $\Theta(n)$; to the best of my knowledge, this is the first complexity separation between uniquely represented and conventional implementations for a natural abstract data type in a RAM model of computation.

Chapter 5: Advanced Data Structures

Chapter 5 covers many additional uniquely represented data structures and techniques for their construction. We match the running times of the best conventional implementations for dynamic ordered dictionaries, dynamic order maintenance, dynamic ordered subsets, skip lists, and dynamic trees, up to taking expectations. We also provide efficient solutions for orthogonal range queries (range trees), horizontal point location, orthogonal segment intersection, and dynamic 2D convex hull whose running times are nearly as good as the best conventional versions. This chapter also features the *treap partitioning* scheme, which is a way to select a well-spaced random sample of points from an ordered set to partition it in a strongly history independent manner. This scheme has several nice properties, and we make extensive use of it, for example in the design of B-treaps. B-treaps are a uniquely represented analogue of B-trees and were designed for an external memory model of computation. They have similar performance guarantees as B-trees, and would be well suited to uniquely represented filesystems and databases.

Chapter 6: Adaptive Variants

In Chapter 6 we consider a relaxation of unique representation and strong history independence, in which the representation is allowed to depend on certain information about the historical use of the data structure that is specified up front. The hope is that limited history dependence will allow the data structure to adapt to the input and achieve improved performance without storing unauthorized information about the historical use of the data structures. Designing such data structures can be modeled as the task of designing a uniquely represented implementation for an abstract data type that explicitly stores certain historical information in its

logical state.

Chapter 7: Fundamental Limitations

Chapter 7 is concerned with fundamental barriers to the development of efficient uniquely represented implementations for various abstract data types, specifically those that do not plague conventional implementations. We show one such barrier; intuitively, when an operation has the potential to "destroy" a large quantity of historical information, any uniquely represented implementation must take a long time to perform it. We use this argument to derive complexity separations for uniquely represented implementations of the union-find abstract data type and of meldable heaps.

Chapter 8: Uniquely Represented Systems

Finally, in Chapter 8 we discuss how to combine uniquely represented data structures together to obtain uniquely represented systems.

Source Materials

Many of the contributions in this dissertation have been published previously. The constructions for uniquely represented hash tables (Chapter 3), binary search trees (Section 4.5), ordered dictionaries (Section 5.4), order maintenance (Section 5.6), dynamic trees (Section 5.12), and the treap partitioning scheme (Section 5.3) all appear or are discussed in [BG07]. The technique of hash consing (Section 5.2) and the technique of obtaining uniquely represented data structures from oblivious data structures (Section 5.11) are discussed in [BG06]. The constructions for horizontal point location and orthogonal segment intersection (Section 5.9), orthogonal range queries (Section 5.8), 2-D dynamic convex hull (Section 5.10), and dynamic ordered subsets (Section 5.7) appear in [BGV08a, BGV08b].

1.3 Overview of Related Work

Early Work on Uniquely Represented Data Structures. Snyder [Sny77] considered the problem of building uniquely represented dictionaries in a graph storage

model. He proved that in order to uniquely represent a dictionary on n items with a "tree-like" graph of bounded degree, $\Theta(\sqrt{n})$ time per insertion, deletion, and search is both necessary and sufficient. Moreover, Snyder [Sny77] states that his results "suggest a new general rule: for dynamic problems, avoid uniquely represented data structures," in stark contrast to the thesis presented here. We should note that Snyder considers worst-case running times, and, as we will show, with suitable randomization good bounds in expectation or with high probability can often be obtained. Additionally, Snyder's notion of unique representation is unusual; informally, he requires that two dictionaries with the same *number* of keys have the same tree structure. This is in contrast to the more natural requirement that any two dictionaries with the same *set* of keys have the same tree structure. We differentiate the two concepts by using the term *size-uniquely represented* for the former and *uniquely represented* for the latter.

Andersson and Ottmann [AO95] built on the work of Snyder, and considered size-uniquely representing a dictionary with graphs of bounded outdegree, and derive a matching lower and upper bound of $\Theta(n^{1/3})$ time per operation in the worst-case. In a model similar to Snyder's, Sundar and Tarjan [ST90] derive a bound of $\Theta(\sqrt{n})$ time per operation for the task of uniquely representing a dictionary with a binary search tree.

A Brief History of History Independence. The notion of *oblivious* data structures, for which the *pointer structure* of a data structure reveals nothing about its history, serves as a precursor to history-independent data structures and was first studied by Micciancio [Mic97]. The two main notions of history independence, *weak* and *strong*, were advanced by Naor and Teague [NT01], and further studied by Hartline *et al.* [HHM$^+$05]. They are defined formally in Section 2.2. Informally, we say a data structure has a *state*, which maps each allowed operation to an output and the following state. A data structure is *weakly history independent* (WHI) if any two sequences of operations resulting in the same state result in the same distribution over memory representations. Here, the distributions may vary with a sequence of random bits hidden from the observer. The definition of strong history independence is more involved (see Section 2.2), but was proved equivalent to being uniquely represented by Hartline *et al.* [HHM$^+$05] for *reversible* data structures, where a data structure is said to be reversible if any state is reachable from any other state via a sequence of legal operations.

Long before Naor and Teague's work on history independence, Amble and Knuth [AK74] developed a uniquely represented hash table that does not support

deletions. They showed that it has excellent performance assuming a random hash function is used. Naor and Teague [NT01] similarly develop an efficient uniquely represented hash table that does not support deletions, but require only $O(\log n)$ pair-wise independent hash functions rather than a truly random hash function. Naor and Teague also give a WHI hash table that supports deletion, is space efficient, and takes expected amortized $O(1)$ time for insertions and deletions, and $O(1)$ time for search.

Hartline *et al.* [HHM+05] gave an alternate characterization of SHI data structures, considered dynamically resizable data structures which are efficient against a non-oblivious adversary selecting the sequence of operations, and prove upper bounds for dynamically resizable WHI data structures and lower bounds for dynamically resizable SHI data structures.

Buchbinder and Petrank [BP06] studied the time complexity of WHI and SHI heaps and queues in a comparison based model of computation. They show a large complexity separation for these data structures. In particular, for a WHI heap they obtain $O(\log n)$ time insert, increase-key, and extract-max operations, and an $O(n)$ time build heap operation on n items. Yet for a SHI heap, insert and increase key must take $\Omega(n)$ time in the worst-case, and build heap $\Omega(n \log n)$. For queues, the maximum cost operation among $\{$enqueue, dequeue$\}$ takes $O(1)$ time in their WHI implementation, and worst-case $\Omega(n)$ time in any SHI implementation.

There has also been work on history-independent data structures in machine models with write-once memories. Molnar *et al.* [MKSW06] considered the problem of storing ballots in a voting machine with tamper-evident write-once memory in a history independent manner. They obtain a SHI data structure that takes $O(n)$ time for insertions and $O(n^2)$ space to store n ballots, as well as more space and time efficient WHI data structures. Moran *et al.* [MNS07] consider the same problem, and give a deterministic data structure for storing a set of at most n elements from a universe of size N in space $O(n \cdot \text{polylog}(N))$ such that insertion takes amortized $O(\text{polylog}(N))$ time.

Acar *et al.* [ABH+04] devised a model and algorithms to automatically *dynamize* static algorithms. That is, they show how to run an off-line algorithm in an on-line environment by maintaining enough state so that as the input changes the algorithm can exploit its earlier work as much as possible rather than restarting the computation from scratch each time. Acar *et al.* gave sufficient conditions for strong history independence of dynamized algorithms, as well as a framework for analyzing their performance. As an example, they gave an efficient SHI data structure for dynamic trees that supports expected $O(\log n)$ time link and cut operations

and uses $O(n \log n)$ space.

Subsequent to the development of our uniquely represented hash table and dynamic perfect hash table which support deletions and are based on linear probing, Naor, Segev, and Wieder [NSW08] developed a uniquely represented dynamic perfect hash table which support deletions based on the Cuckoo Hashing scheme of Pagh and Rodler [PR04]. We compare our dynamic perfect hash table with that of Naor *et al.* in Section 3.2.3.

What Unique Representation Is Not. Uniquely represented data structures have no redundancy of representation. This does not mean they cannot have redundancy in the sense of having low entropy, e.g., to be used in error correcting codes. Error correction is in some sense orthogonal to unique representation. Any code that maps each message to exactly one codeword (which may depend on some random bits used by the encoder) may be used. This will include all codes for which the encoder takes the entire message as input before encoding it. An encoder that did not have this property would need to maintain the codeword encoding a message that was dynamically changing in accordance with some user operations — an unusual coding scheme indeed.

Note that the guarantees on history independence provided by uniquely represented data structures are information theoretic rather than cryptographic. For the purposes of what information is stored by the system, both the observer and the user with access to the interface are assumed to be computationally unbounded. (As we discuss in Chapter 2, it is often fine to constrain the user to use only polynomial time computations.) We do not attempt to hide any information that the system is required to store from either the user or the observer. However, unique representation does not preclude the use the cryptography in the system; the information the system is required to store may well be encrypted files. As with error correcting codes, there is no difficulty if the encryption scheme is given the entire file up front. However, one must be careful if incremental cryptography is used to ensure history independence of the resulting encrypted files, independently of how they are stored.

Chapter 2

Background and Semantic Foundations

2.1 Basic Terminology and Notations

Elementary Notation

Let $\mathbb{R}$ denote the real numbers, $\mathbb{Z}$ denote the integers, $\mathbb{Z}^+$ denote the positive integers, and $\mathbb{N} = \{0, 1, 2, \ldots\}$ denote the natural numbers. For $n \in \mathbb{Z}$, let $[n]$ denote $\{0, 1, 2, \ldots, n-1\}$ and for $a, b \in \mathbb{Z}$, let $[a : b] := [a, b] \cap \mathbb{Z}$. Let $\{a_i\}_{i=1}^{n}$ denote the sequence $a_1, a_2, \ldots, a_n$. We occasionally use the notation $\sum\{x : P(x)\}$ to denote the sum over all elements x satisfying a predicate P, and similarly use $\prod\{x : P(x)\}$ to denote the product over the same set. This notation can be viewed as interpreting $\sum$ and $\prod$ as operators mapping sets of numbers to numbers. Throughout this dissertation we use $\log(n)$ to denote $\log_2(n)$ by default and $\ln(n)$ to denote $\log_e(n)$.

Graph Theory Terminology

Graphs. We use the following standard graph theory terminology. A *graph* G is a pair of sets (V, E) where V is the set of *nodes* (also called vertices) and E is the set of *edges*. If there are many graphs to consider, we may use $V[G]$ and $E[G]$ to denote the nodes and edges of G, respectively. In *undirected* graphs, each edge is a set of nodes of size two, whereas in *directed* graphs each edge is an ordered pair of nodes. A *subgraph* of (V, E) is a graph (V', E') such that $V' \subseteq V$ and $E' \subseteq E$.

A directed edge (u, v) is said to have u as its *tail* and v as its *head*. An undirected edge $\{u, v\}$ is said to be *incident* on u and v, and the *degree* of a node is the number of edges incident on it. In directed graphs, the *in-degree* of a node v is the number of edges with head v, i.e., $|\{(u, v) : u \in V, (u, v) \in E\}|$, and the *out-degree* of a node v is the number of edges with tail v, i.e., $|\{(v, w) : w \in V, (v, w) \in E\}|$. A node u is said be a *neighbor* of v in an undirected graph $G = (V, E)$ if $\{u, v\} \in E$. A *path* P in an undirected (directed) graph G is a sequence of nodes $\{v_i\}_{i=1}^{|P|}$ such that $\{v_i, v_{i+1}\} \in E$ ($(v_i, v_{i+1}) \in E$) for all $i \in [1 : P - 1]$. A *simple path* is a path in which no node appears more than once; paths that are not simple are called *complex*. Where convenient, we will sometimes identify a path with the set of edges between adjacent nodes in the path. A *cycle* is a path $\{v_i\}_{i=1}^{n}$ such that $v_1 = v_n$. A graph with no cycles is called *acyclic*. An undirected graph such that there is a path between any two nodes is called *connected*. A directed graph such that there is a path between any two nodes is called *strongly connected*. An undirected graph that is connected and acyclic is called a *tree*.

Trees. We use the following concepts related to trees. A *leaf* is a node in the tree with degree one; any node that is not a leaf is called an *internal node*. A tree $T = (V, E)$ may be *rooted* at a distinguished node $r \in V$, which we call the *root*. Rooting the tree induces various relationships on the nodes. The *parent* of a node u is the neighbor of u that is on the unique u to root path in T; every node but the root has a parent. If v is the parent of u, then u is said to be a *child* of v. Let $R \subset V \times V$ be the relation $R = \{(v, v) : v \in V\} \cup \{(u, v) : v = \text{parent}(u)\}$. Let $(V, \leq_{\text{parent}})$ be the partial order that is the transitive closure of R. We say that u is an *ancestor* of v if $v \leq_{\text{parent}} u$, and say that u is a *descendant* of v if $u \leq_{\text{parent}} v$. In other words, the ancestors of a node u are the nodes in the u to root path in T, and the descendants of u are those nodes v such that u lies on the v to root path in T. We will sometimes refer to the *proper ancestors* or the *proper descendants* of a node u, these are simply the ancestors or descendants of u excluding u itself, respectively.

For a rooted tree $T = (V, E)$, let $|T|$ be the number of nodes in T, and for a node $v \in T$, let T_v denote the *subtree rooted at* v, which is the subgraph (V', E') of T with V' equal to the descendants of v and $E' = \{U \subseteq V' : |U| = 2\} \cap E$. Additionally, let $\text{depth}(x)$ denote the length of the path from x to the root of T minus one.

Asymptotic Growth Notation

We use the asymptotic growth notation that is customarily used to describe the complexity of algorithms. This notation sometimes goes by the name "big O notation." Fix two functions $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$.

- We write $f = O(g)$ if there exist positive integers c, n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.
- We write $f = \Omega(g)$ if there exist positive integers c, n_0 such that $f(n) \geq \frac{1}{c} \cdot g(n)$ for all $n \geq n_0$.
- We write $f = \Theta(g)$ if $f = O(g)$ and $f = \Omega(g)$.
- We write $f = o(g)$ if for all positive integers c there exists a positive integer n_0 such that $f(n) \leq \frac{1}{c} \cdot g(n)$ for all $n \geq n_0$.
- We write $f = \omega(g)$ if for all positive integers c there exists a positive integer n_0 such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.

Technically, $O(g), \Omega(g), \Theta(g), o(g)$, and $\omega(g)$ are often defined as sets of functions of type $\mathbb{N} \to \mathbb{N}$, e.g., $O(g) = \{f \ : \ \exists c, n_0 \in \mathbb{Z}^+ \ \forall n \geq n_0, \ f(n) \leq c \cdot g(n)\}$. In this case it would be more appropriate to write $f \in O(g)$, however the convention is to abuse notation and write $f = O(g)$ instead. Another conventional abuse of notation is to write $f = O(e)$ for an expression e rather than a function; for example, we write $f = O(n^2)$ instead of the technically correct $f = O(n \mapsto n^2)$. Finally, we use the notation $f = \mathrm{poly}(n)$ to indicate that there exists a $c \in \mathbb{N}$ such that $f = O(n^c)$.

Probability Theory Terminology

Let $\mathbf{Pr}[\varepsilon]$ denote the probability of event ε. Let $\mathbf{E}[X]$ denote the expectation of random variable X. Throughout this dissertation, we let *with high probability* (whp) mean with probability at least $1 - 1/n^c$, where $c > 0$ is any user defined constant, and n is (as is customary) some natural measure of the size of the problem. Sometimes for some random variable X we will write "X is $O(f(n))$ with high probability" as shorthand for "for all constants $c > 0$ there exists a constant c' such that $\mathbf{Pr}[X \leq c' \cdot f(n)] \geq 1 - 1/n^c$." Likewise, we will write "X is $\Omega(f(n))$ with high probability" as shorthand for "for all constants $c > 0$ there exists a constant c' such that $\mathbf{Pr}\big[X \geq \frac{1}{c'} \cdot f(n)\big] \geq 1 - 1/n^c$."

2.2 Unique Representation and History Independence

Abstract Data Types

An *abstract data type* (ADT) specifies data to be stored, as well as operations that can be performed on the data. Abstract data types can be formally specified algebraically; for example, Micciancio uses Σ-algebras [Mic97]. We will use a more lightweight formalism to model them. We view the ADT as defining a set of *states*, and operations map their inputs and the current state to a new state and, optionally, an output. The ADT itself may then be thought of as a directed graph on its states, with edges labeled by operation (including their inputs) and output. Following Hartline *et al.* [HHM$^+$05], we call this directed graph the *state transition graph*. Sequences of operations on the data structure then correspond to paths in the state transition graph.

More precisely, we model an ADT as a set V of logical states, a special starting state $v_0 \in V$, a set of allowable operations $\mathcal{O}$ and outputs Λ, a transition function $\tau : V \times \mathcal{O} \to V$, and an output function $\lambda : V \times \mathcal{O} \to \Lambda$. The ADT is initialized to v_0, and if operation $O \in \mathcal{O}$ is applied when the ADT is in state v, the ADT outputs $\lambda(v, O)$ and transitions to state $\tau(v, O)$.

Models of Computation

A *machine model* $\mathcal{M}$ is itself an ADT, typically at a relatively low level of abstraction, endowed with a programming language. For our purposes a programming language for $\mathcal{M}$ can be thought as a way to specify (possibly infinite) paths in $\mathcal{M}$'s state transition graph. A program is then identified with a set of paths $\mathcal{P}$ called its *trace set*, where $P \in \mathcal{P}$ starting at $v \in V$ represents the execution trace of the program starting at state v. The program itself can be modeled as a function $p : V \to V \cup \{\infty\}$ where $p(v)$ is the machine state resulting from running p starting from machine state v and $p(v) = \infty$ in the event that p does not terminate when executed from initial state v. Example machine models include the *random access machine* (RAM) with a simple programming language that resembles C, the *Turing machine* (where the program corresponds to the finite state control of the machine), and various *pointer machines*. For an detailed discussion of various machine models, we refer the interested reader to [vEB90].

An *implementation* of an ADT $\mathcal{A}$ on a machine model $\mathcal{M}$ is a mapping f from the operations of $\mathcal{A}$ to programs over the operations of $\mathcal{M}$.

The machine model that we will typically use is the standard *transdichotomous random access memory* (RAM) machine, specifically a *word RAM*. The model is called transdichotomous to describe the fact that the machine is allowed to grow with the problem size, in two ways. The machine we use has a array of *words* of some finite length. The length is assumed to be large enough to accommodate whatever program we are running on the input. Each word is simply a sequence of w bits, and we assume $w \geq \log |\mathcal{U}|$, where $\mathcal{U}$ is the universe of object labels (including all indices in the word array). This ensures an object label can be stored in a single word. We use a unit-cost RAM model, so that our machine can perform basic arithmetic and logical operations (including multiplication, division, and modulo) in one time step. These basic operations all take a constant number of arguments, and, true to the RAM name, take only one time step regardless of where in the word array their arguments are stored. We further assume that the machine model is endowed with a programming language that resembles C in its capabilities. Thus, the programming language should have integers, booleans, strings, pointers, if-then-else conditionals, while loops, and functions. The machine model also has separate (read-only) space for a program of finite length l to execute. We assume $w \geq \log(l)$. The problem input is initially written in the word array, and for dynamic data structures operations can be specified by writing data into the word array and specifying a line in the program to start executing from. Finally, our machine model has access to an infinitely long sequence of random bits, though we will be careful to indicate how many random bits are required by each of our data structures. We assume the machine's programming language has a facility to read from the sequence of random bits; in this way the random bits can affect how programs alter the machine state.

History Independence and Unique Representation

Recall that our goal is to store exactly the information specified by an ADT, namely its current state. However, this task is complicated by the fact that in any implementation of the ADT, operations map their inputs and the current physical machine state to a new physical machine state and, optionally, an output. We will call a physical machine state corresponding to a ADT state a *memory representation* of that ADT state. History independence is defined in terms of how paths through the state transition graph relate to changes in the memory representation. The definitions of weak and strong history independence presented here are found in [HHM$^+$05], and are equivalent to the definitions given in [NT01].

Definition 1 (Weak History Independence)**.** *A data structure is* weakly history independent (WHI) *if, for any two sequences of operations X and Y that take the data structure from initialization to state A, the distribution over memory representations after X is performed is identical to the distribution after Y is performed. (The distribution in question is over the random bits that the WHI data structure may hide from the observer.)*

Definition 2 (Strong History Independence)**.** *Fix states A and B of an ADT, and two (possibly empty) sequences of operations X and Y that take the ADT from state A to state B. Let α be any memory representation of state A in some data structure implementation of that ADT. The data structure is* strongly history independent (SHI) *if, for any such A, B, X, Y, and α, the distribution over memory representations after X is performed starting from memory representation α is identical to the distribution after Y is performed starting from memory representation α. (The distribution in question is over the random bits that the SHI data structure may hide from the observer.)*

Strong history independence was defined to capture the notion that an observer with access to the memory layout of the data structure at several different points in time can learn no more information than a legitimate user accessing the data structure via its standard interface at those points in time. Interestingly, the first strongly history independent data structures were uniquely represented, in the following sense.

Definition 3 (Unique Representation)**.** *A data structure is* uniquely represented *if it has canonical representations up to initial randomness. Formally, given a machine model $\mathcal{M}$, an implementation f of some ADT (V, v_0, τ, λ) is said be uniquely represented if for each ADT state $v \in V$, there is a unique machine state $\sigma(v)$ of $\mathcal{M}$ that encodes it. Thus, if for each ADT operation O we run on (V, v_0, τ, λ) we run the the program $f(O)$ implementing O on $\mathcal{M}$, and perform no additional operations on $\mathcal{M}$, then the machine is in state $\sigma(v)$ iff the ADT is in logical state v.*

The observation that early strongly history independent data structures were uniquely represented led Naor and Teague [NT01] to ask "whether unique representation is necessary for strong history independence." Hartline *et al.* [HHM+05] provide an affirmative answer in the case of data structures implementing *reversible* abstract data types, defined as follows.

Definition 4 (Reversible Abstract Data Types [HHM+05])**.** *An abstract data type is* reversible *if and only if its state transition graph is strongly connected.*

In other words, reversible abstract data types are those for which there always exists some sequence of operations which returns the data type to its initial state. Typically this involves a sequence of deletions.

Theorem 1 ([HHM+05]). *Any strongly history independent implementation of a reversible ADT is uniquely represented.*

In this dissertation, we will restrict ourselves to the study of uniquely represented data structures, and Theorem 1 provides additional justification for doing so, at least within the context of a search for efficient strongly history independent data structures.

Why Unique Representation is the Strongest Notion of History Independence

Fix an abstract data type $\mathcal{A} = (V, v_0, \tau, \lambda)$ and a machine model $\mathcal{M} = (V', v_0', \tau', \lambda', \mathcal{P})$ implementing $\mathcal{A}$. For a sequence ω of operations in $\mathcal{A}$, let ω' be the implementation of ω in $\mathcal{M}$. Let $v(\omega)$ and $v'(\omega')$ denote the logical and machine states obtained after running ω and ω', respectively, starting from the respective initial states of $\mathcal{A}$ and $\mathcal{M}$. We can model an observer as an *interpretation* from machine states to some (arbitrary) information set I, that is, a function $f : V' \to I$. The interpretation models the strength of the observer. For example, the weakest observer has a constant function as its interpretation; this observer cannot learn anything at all from the machine state. The most powerful observer interprets each machine state differently; in other words, the most powerful observer has an injective interpretation. Uniquely represented data structures are the only data structures that are history independent with respect to *all* such observers. This is because in an uniquely represented implementation, if operation sequences ω_0 and ω_1 satisfy $v(\omega_0) = v(\omega_1)$ then $v'(\omega_0') = v'(\omega_1')$, which immediately implies $f(v'(\omega_0')) = f(v'(\omega_1'))$. Conversely, in a non-uniquely represented implementation there exist ω_0 and ω_1 such that $v(\omega_0) = v(\omega_1)$ and $v'(\omega_0') \neq v'(\omega_1')$. Therefore any observer with $f(v'(\omega_0')) \neq f(v'(\omega_1'))$ will be able to distinguish these two historical sequences of operations. This same argument can be easily extended to the case in which the observer gets access to the machine state at several times, each time between the execution of operations in some sequence of operations.

History Independence as Distinguishability

The original goal of history independence – to ensure that the data structure stores only the information specified by the abstract data type that it implements – can be alternatively modeled as follows: the observer should not be able to make distinctions that the user with access to the legitimate interface cannot. In other words, if a user is given access to the interface of two systems implementing some abstract data type, and the observer is given access to the machine state of those two systems, then the observer should be able to distinguish the two systems if and only if the user can distinguish them.

This perspective allows us define history independence with respect to computationally bounded users and observers. One subtle way in which this differs from the definition of strong history independence given in Definition 2 is that for some abstract data types, access to the legitimate interface alone might possibly be insufficient to infer the exact logical state. This may be true for the abstract data type for deterministic finite automata described in Section 1.1.2, in which each operation causes an update to the logical state. However, we will typically assume that the exact logical state can be computed using the legitimate interface. For example, in a binary search tree the interface may allow the user to do a traversal of the tree. In a queue, it is possible to simply dequeue all the elements. A basic hash table may not allow an efficient way to list its contents, however we can suppose a linear time successor operation, or a polynomial time operation that lists the keys of the hash table in sorted order.

Given that the logical state can be computed in polynomial time using the legitimate interface, distinguishing the logical states of two systems then becomes a non-isomorphism problem on the contents of the abstract data type. If this non-isomorphism problem can be solved in polynomial time, then the uniquely represented implementation is strongly history independent with respect to a computationally unbounded observer and a polynomial-time bounded user. Note that if the uniquely represented implementation is efficient then the non-isomorphism problem is likely to be easy. This is because if the user can find sequences of operations generating the two logical states, then the user can run the uniquely represented implementation on both sequences (on a separate machine controlled by the user) and simply compare the memory representations generated; if the logical states are equal, then the memory representations will be as well[1].

[1]Alternately, if the non-isomorphism problem on the contents of the abstract data type is hard, then no time and space efficient uniquely represented implementation will be forthcoming.

Randomization and Unique Representation

Note that when constructing uniquely represented data structures, we allow the machine representation of a logical state to depend on the random bits of the machine, however the data structures will be uniquely represented and strongly history independence no matter what random string is used. Therefore a computationally unbounded observer with access to the machine state, including the random bits it uses, can learn no more than if told what the current logical state is. We use randomization solely to improve performance; in our performance guarantees we compute probabilities and expectations over these random bits. Our performance guarantees assume the sequence of operations is chosen independently of these random bits (i.e., the user may be an *oblivious* adversary but not an *adaptive* adversary). Since we will focus on randomized data structures, we will also make use of the following definitions.

Definition 5 (k-Universal Hash Family)**.** *A family $\mathcal{H}$ of functions from X to Y is* k-universal *if for all distinct $x_1, x_2, \ldots, x_k \in X$ and for all $y_1, y_2, \ldots, y_k \in Y$*

$$\mathbf{Pr}_{h \in \mathcal{H}} \left[\bigwedge_{i=1}^{k} h(x_i) = y_i \right] \leq |Y|^{-k}$$

where h is chosen uniformly at random from $\mathcal{H}$.

Such hash families are often described as *k-wise independent*. We use the phrases k-universal and k-wise independent interchangeably in this dissertation.

Universal hash families were defined by Carter and Wegman [CW79]. There are various k-universal hash functions that can be evaluated in $O(1)$ time and are easy to sample. For example, the family of degree $k-1$ polynomials over a finite field is k-wise independent, as was shown by Wegman and Carter [WC81]. Thorup and Zhang provide a 4-wise independent hash function that is very fast in practice [TZ04]; Pagh *et al.* [PPR07] claim that this hash family is in fact 5-wise independent. Where $\Theta(\log n)$-universal hash functions are needed, the constructions of Siegel [Sie89, Sie95a], Östlin and Pagh [OP03], and Dietzfelbinger and Woelfel [DW03] are suitable, assuming the keys are integers. The latter are also suitable where full randomness is called for.

For technical reasons, some uniquely represented data structures with worst-case performance guarantees require very large amounts of randomness in the worst case. We will exclude the random bits used from the space usage, as consis-

tent with our machine model. However, in the interests of full-disclosure, we will refer to the space used by such data structures as *data space* rather than just space.

Definition 6 (Data Space)**.** *The* data space *of a data structure is the space it uses excluding any space used to store random bits.*

2.3 Basic Methods to Construct Uniquely Represented Data Structures

Before proceeding to complex constructions, we note that there are some fairly simple methods for generating uniquely represented data structures. We regard the following methods as folklore, as they are based on the ideas of bit-vector encodings, finite state automata, and sorting, respectively.

Foremost among these basic methods is to store the *characteristic vector* of some *factored representation* of the state. For example, a subset $S \subset \mathcal{U}$ can be stored as its characteristic vector $\chi(S) \in \{0,1\}^{|\mathcal{U}|}$ defined by $\chi(S)_e = 1$ for all $e \in S$ and $\chi(S)_e = 0$ otherwise. By a factored representation, we mean that the state can be uniquely represented by the truth values of a set of predicates – in this case, the subset S can be uniquely represented by the collective truth values of the predicates "$e \in S$" for all $e \in \mathcal{U}$. Of course, this requires $|\mathcal{U}|$ bits of space to store, which may make it inappropriate if $|S| << |\mathcal{U}|$.

A second simple method for generating uniquely represented data structures is to explicitly store the entire state transition graph (defined in Section 2.2), and maintain a single pointer to the current state. Typically the state transition graph is far too large to apply this method, however it may be helpful to represent some small part of the state this way. This often amounts to table lookup. We use this technique in our solution to the order maintenance problem in Section 5.6, to store comparison information for very small subsets of the elements.

Yet another simple method for generating uniquely represented data structures is to label the data items stored by the ADT, define a total ordering σ on the labels, and store them in order according to σ. Thus, to store a set S of at most n integers, one can create an array of size n, and store S at the $|S|$ smallest indices of the array, in sorted order. Though this method is typically very space efficient, updates take linear time – not only in the worst-case but also on average when inserting or deleting a random element.

Chapter 3

Uniquely Represented Hash Tables

In this chapter we describe our constructions of efficient uniquely represented hash tables. We rely on the fact that the RAM provides a native implementation of the array ADT that is uniquely represented (see Section 4.1). Uniquely represented hash tables crucially underlie the development of uniquely represented data structures for the RAM model of computation, because they provide a way to map various structures, such as binary trees, into the one-dimensional memory provided by the RAM in such a way that the mapping is one-to-one and supports fast updates. In this context, supporting fast updates means that if the input structure is modified slightly, then the RAM memory configuration that it maps to is modified only slightly, and we can efficiently implement the necessary modification. Interestingly, the existence of efficient uniquely represented hash tables can be interpreted as a *trace stability* result [ABH$^+$04], which we will discuss further in the context of *dynamization* in Section 5.12.

3.1 Efficient Uniquely Represented Hashing

Our approach is based on exploiting an interesting property of the stable marriage algorithm of Gale and Shapley [GS62], stated below in Theorem 2. The stable marriage problem is as follows: Given a set M of n men and a set W of n women, and a preference list over the opposite sex for each person, find a *stable* matching $E \subset M \times W$ of size n. Here, a man's preference list is a permutation over W, a woman's preference list is a permutation over M, and persons appearing earlier in the permutation are considered strictly preferable to those appearing later in the

permutation. A matching E is stable if for all $(m, w), (m', w') \in E$, it is *not* the case that m prefers w' to w and w' prefers m to m'; such a case is called an *instability*.

Recall that in the Gale-Shapley stable marriage algorithm, the men propose to the women in decreasing order of their preferences, and each woman is tentatively matched with her favorite man among all those who have proposed to her. The algorithm terminates when all men are tentatively matched with some woman.

Note that the algorithm is underspecified in the sense that there may be many men who are not tentatively matched, and they may propose in arbitrary order. Thus, there are many different *valid implementations* of this algorithm, corresponding to various ways of selecting among tentatively unmatched men. Nevertheless, the following theorem, implicit in [GS62] and treated explicitly in [KT06], shows the outcome is not affected by the choice of valid implementation.

Theorem 2 ([GS62])**.** *For each instance of the stable marriage problem, every valid implementation of the Gale-Shapley algorithm outputs the same stable matching on that instance.*

We provide the proof for completeness.

Proof: We first prove by contradiction that the Gale-Shapley algorithm outputs some stable matching $E \subset M \times W$. Suppose $\{(m, w), (m', w')\} \subset E$ is an instability because m prefers w' to w and w' prefers m to m'. Then m must have proposed to w' before w and been rejected. Since each woman is tentatively matched with her favorite man among all those who have proposed to her, this means that w' must have been tentatively matched with a man m'' that she preferred to m. Since $(m', w') \in E$, this implies that w' prefers m' at least as much as m'', and thus strictly prefers m' to m, contradicting our assumption. So there are no instabilities in E.

We next prove by contradiction that E is a perfect matching. Assume E is not perfect, so that some man m is unmatched in E. Note a woman can only be unmatched in E if no man ever proposes to her. However m must have proposed to every woman, so every woman must be matched. Since there are an equal number of men and women, this contradicts the assumption that E is not perfect.

Finally, we prove that every valid implementation of the Gale-Shapley algorithm outputs the same stable matching E. For each man $m \in M$, define $\mathrm{best}(m)$ to be m's favorite woman in $\{w : \exists$ stable matching E' s.t. $(m, w) \in E'\}$. In other words, $\mathrm{best}(m)$ is the best match m can get among all stable matchings. We claim that every valid implementation of the Gale-Shapley algorithm outputs $E = \{(m, \mathrm{best}(m)) : m \in M\}$, which if true is clearly sufficient to complete the proof.

Suppose that this is not the case, so that some man m is rejected by $\text{best}(m)$. Let m be the first man rejected by $\text{best}(m)$ in the execution of the algorithm, and suppose $\text{best}(m)$ is tentatively matched with m' when rejecting m. Thus $\text{best}(m)$ prefers m' to m. Now fix a stable matching E' with $(m, \text{best}(m)) \in E'$ and suppose $(m', w') \in E'$. Returning to the execution of the Gale-Shapley algorithm, at the moment m is rejected by $\text{best}(m)$, by assumption m' has not yet been rejected by $\text{best}(m')$. This implies m' has not yet been rejected by any woman that m' prefers less than or equally well as $\text{best}(m')$, including w'. From this we infer that m' prefers $\text{best}(m)$ to w', which implies $\{(m, \text{best}(m)), (m', w')\}$ is an unstable pair, contradicting the stability of E'. ■

The Framework. Theorem 2 suggests the following approach to constructing a uniquely represented hash table that supports insertions and searches: interpret the keys as men and the slots of the hash table as women, and construct a distribution on stable marriage instances between the universe of keys, U, and the set of all slots. This distribution is based on the random bits of the hash table. The correspondence between stable marriage and hashing that we use[1] is given in Table 3.1.

Stable Marriage	Hashing
men	keys
women	slots
male preference lists	key probe sequences
female preference lists	collision/eviction policies
matchings	hash table memory configurations

Table 3.1: The correspondence between stable marriage and hashing.

In particular, the probe sequence for a key **k** will equal **k**'s preference list over the slots. (If the probe sequence has duplicate entries, retain only the first occurrence of each slot to obtain the preference list). The preference lists for each slot will be used to resolve collisions. In this case, Theorem 2 ensures that for each set of keys of size at most n, the resulting memory representation is the same no

[1]To the best of my knowledge, the first peer-reviewed paper identifying this correspondence is [BG07], however an anonymous reviewer pointed out that Donald Knuth used this correspondence in analyzing random stable matching instances during a lecture series at the Université de Montréal. Scribe notes for these lectures were originally published in French, and are also available in English [Knu97].

matter what order the keys are inserted in. So the resulting hash table is uniquely represented under insertions. To ensure that the hash table performs well, we must ensure that

1. We can sample efficiently from the distribution on stable marriage instances.
2. Each instance in the distribution can be represented compactly, such that the following operations take constant time: determining the i^{th} slot in $\mathbf{k}$'s preference list, determining slot x's rank in $\mathbf{k}$'s preference list, and comparing any two keys with respect to x's preference list, for arbitrary $i, \mathbf{k}$ and x.
3. For each set of keys $S \subset U$ such that $|S| \leq n$, the expected running time of the Gale-Shapley algorithm on an instance drawn from the distribution and restricted to the set of men S is $O(|S|)$ on every valid execution of the algorithm in the set of valid executions that maintain a total order σ on the keys and select tentatively unmatched keys in order of σ.

We note that the uniquely represented hash table of Naor and Teague [NT01], which does not support deletions, fits directly into this framework. Intuitively, to ensure property (3) it makes sense to construct the slot preference lists so that each slot prefers keys that rank it high on their preference lists. Not surprisingly then, Naor and Teague favor what they call "youth-rules" for collision resolution, which do exactly this.

To enable support for deletions, the crux of the matter is efficiently computing, for a slot x currently holding key $\mathbf{k}$, the slot x' of the most preferred key $\mathbf{k}' \neq \mathbf{k}$ (according to x's preference list) that prefers x to its current slot. This requirement is what keeps us from extending the uniquely represented hash table of Naor and Teague to support deletions. Though this is a function of the state of the hash table, we will abuse notation slightly and denote it by $\mathrm{next}(x)$.

Pseudocode for insertion and deletion are given in Figure 3.1 on the next page. In the pseudocode, $\mathrm{probe}(\mathbf{k}, i)$ is the i^{th} slot in $\mathbf{k}$'s probe sequence, A is the array of the hash table, and $\mathrm{rank}(\mathbf{k}, x) = i$ if x is the i^{th} slot in $\mathbf{k}$'s probe sequence.

Implementation. Though there are many hashing schemes, such as quadratic probing, that can be implemented in our framework to give efficient uniquely represented hash tables, perhaps the simplest implementation is based on linear probing. Interestingly, specializing our framework to linear probing results in essentially the same insertion algorithm as the linear probing specialization of Amble and Knuth's

```
find(key k)
    For (i = 0, 1, 2, ..., p − 1)
        If (A[probe(k, i)] is empty OR slot
            probe(k, i) prefers k to A[probe(k, i)])
            return null;
        Else if (A[probe(k, i)] equals k)
            return probe(k, i);

insert(key k)
    Set x = probe(k, 0), i = 0, and k' = k;
    While (A[x] not empty)
        If (A[x] equals k') then return;
        Else if (slot x prefers A[x] to k')
            Increment i; Set x = probe(k', i);
        Else (slot x prefers k' to A[x])
            Swap the values of k' and A[x];
            Set i = rank(k', x); Increment i;
            Set x = probe(k', i);
    Set A[x] = k'; return;

delete(key k)
    Let slot x = find(k).
    If x is null, return.
    While (next(x) is not null)
        Set y = next(x); Set A[x] = A[y]; Set x = y;
    Set A[x] to be empty; return;
```

Figure 3.1: Pseudocode for a generic uniquely represented hash table following our framework.

'Ordered hash table' framework [AK74]. Of course, our hash table will also support deletions.

To build a hash table for n keys we fix $p = (1 + \epsilon)n$ for some $\epsilon > 0$, and a total ordering on the keys. As long as we can compare two keys in constant time, this ordering can be arbitrary, however for simplicity of exposition we will assume the keys are integers and use the natural ordering. That is, each slot prefers $\mathbf{k}$ to $\mathbf{k}'$ if $\mathbf{k} > \mathbf{k}'$. Then sample a 5-universal hash function $h : U \to [p]$ that can be evaluated

in constant time. The functions

$$\begin{aligned} \text{probe}(\mathbf{k}, i) &:= (h(\mathbf{k}) + i) \mod (p), \text{ and} \\ \text{rank}(\mathbf{k}, x) &:= (x - h(\mathbf{k})) \mod (p) \end{aligned}$$

can both be computed in constant time.

Search proceeds in a fashion similar to a standard linear probing hash table. Specifically, we try $\text{probe}(\mathbf{k}, i)$ for $i = 0, 1, 2,$ etc., until reaching a slot containing $\mathbf{k}$, an empty slot, or a slot containing a key $\mathbf{k}'$ such that $\mathbf{k}' < \mathbf{k}$. In the last case, if $\mathbf{k}$ had been inserted, it would have displaced the current contents of slot $\text{probe}(\mathbf{k}, i)$, so we can report that $\mathbf{k}$ is not present.

Deletions are slightly more involved. We supply the pseudocode for $\text{next}(\cdot)$ in Figure 3.2. In the case of linear probing, $\text{next}(x)$ is the slot x' containing the largest key $\mathbf{k}'$ that probed x but was rejected (or displaced) in favor of another key. Thus $\mathbf{k}'$ residing in slot x' satisfies $\text{rank}(\mathbf{k}', x) < \text{rank}(\mathbf{k}', x')$. Furthermore, since all slot preference lists are the same, $\text{next}(x)$ is the slot y that minimizes $(y - x) \mod p$ from among all slots with keys satisfying this condition.

```
next(slot x)
    Set x' = (x + 1) mod (p);
    While (A[x'] not empty)
        If (rank(A[x'], x) < rank(A[x'], x'))
            return x';
        Set x' = (x' + 1) mod (p);
    return null;
```

Figure 3.2: Pseudocode for $\text{next}(\cdot)$ in the linear probing implementation.

Canonical Memory Representation. We first prove that any hash table following our framework is indeed uniquely represented.

Theorem 3. *For any hash table following our framework, after fixing the random bits there is a unique representation of the slots array for each set of p or fewer keys.*

We will sketch a proof of Theorem 3. Fix the hash table's random bits and a set of keys S such that $|S| \leq p$. Searches do not change the memory representation of the slots array, and thus we can safely ignore them. Defer the treatment of deletions for the moment. We will use Theorem 2 to show that any sequence of insertions resulting in the table having contents S results in the same memory representation.

Suppose key $\mathbf{k} \in S$ is stored in slot $s(\mathbf{k}) \in [p]$. Then $\{(\mathbf{k}, s(\mathbf{k})) : \mathbf{k} \in S\}$ is the stable matching output by the Gale-Shapley algorithm on a particular stable marriage instance. In this instance, $M := S$ and $W := [p]$. The preference lists for each $\mathbf{k} \in M$ are built from the probe sequence for $\mathbf{k}$: if $\mathrm{rank}(\mathbf{k}, w) < \mathrm{rank}(\mathbf{k}, w')$, then $\mathbf{k}$ prefers w to w'. The preference lists for each $w \in [p]$ can be arbitrary. It is now straightforward to verify that any sequence of insertions corresponds to a valid execution of the stable matching algorithm. Note also that Theorem 2 easily extends to the case that $|M| < |W|$, and thus we can apply it to show that there is a unique representation of the slots array if only insertions and searches are permitted.

Finally, consider deletions. Proving that deletions preserve the unique representation property amounts to proving that any two sequences of operations ρ and ρ' resulting in the same hash table contents result in the same representation. Suppose that this holds for any sequence pairs (ρ, ρ') such that ρ has no deletions and ρ' has at most one. Then an easy induction on the maximum number of deletions in $\{\rho, \rho'\}$ yields the desired result. So we consider (ρ, ρ') such that ρ has no deletions and ρ' has exactly one. Without loss of generality, we can assume that the deletion in ρ' deletes a key that was present at the time, that neither ρ nor ρ' has any searches, and that the deletion in ρ' is the last operation in ρ'. So suppose $\rho' = (\mathrm{insert}(\mathbf{k}'_1), \mathrm{insert}(\mathbf{k}'_2), \ldots, \mathrm{insert}(\mathbf{k}'_r), \mathrm{delete}(\mathbf{k}'_r))$. Since insertions maintain the unique representation and ρ contains only insertions, we know that ρ results in the same representation as $(\mathrm{insert}(\mathbf{k}'_1), \mathrm{insert}(\mathbf{k}'_2), \ldots, \mathrm{insert}(\mathbf{k}'_{r-1}))$, so without loss of generality we let ρ equal this sequence.

We will show that $\mathrm{delete}(\mathbf{k}'_r)$ exactly undoes all the changes $\mathrm{insert}(\mathbf{k}'_r)$ makes to the slot array. Set $\mathbf{k}_0 := \mathbf{k}'_r$. During an insertion, whenever two keys collide and the current key in the slot is evicted, we say that the evicted key is *displaced* by the other key. In the pseudocode on page 31, $\mathbf{k}'$ displaces $A[x]$ when we reach the case "x prefers $\mathbf{k}'$ to $A[x]$." We will define $\mathbf{k}_i$ as the key displaced by $\mathbf{k}_{i-1}$ during the insertion of $\mathbf{k}_0 \equiv \mathbf{k}'_r$. Suppose the chain of displaced keys is $\{\mathbf{k}_i : i = 1, 2, \ldots, d\}$. Let $s(\mathbf{k})$ be the slot containing $\mathbf{k}$ immediately after the operation $\mathrm{insert}(\mathbf{k}_0)$ in ρ', and let $s'(\mathbf{k})$ be the slot containing $\mathbf{k}$ immediately before the operation $\mathrm{insert}(\mathbf{k}_0)$ in ρ'. It is easy to see that $s(\mathbf{k}_i) = s'(\mathbf{k}_{i+1})$ for all $i \in \{0, 1, \ldots, d-1\}$, and that the keys not in $\{\mathbf{k}_i : i \in [0 : d]\}$ are not affected. Now consider $\mathrm{delete}(\mathbf{k}_0)$. It first finds $x = s(\mathbf{k}_0) = s'(\mathbf{k}_1)$. It then repeatedly sets $A[x]$ to $A[\mathrm{next}(x)]$ and sets x to $\mathrm{next}(x)$ while $\mathrm{next}(x)$ exists. To ensure this is the desired behavior, we require that $\mathrm{next}(s(\mathbf{k}_i)) = s(\mathbf{k}_{i+1})$. Fortunately, this is relatively easy to confirm via proof by contradiction, as is the fact that the delete operation correctly clears the slot that

used to contain $\mathbf{k}_d$, and leaves all other slots unaffected. Thus ρ' results in the same representation as ρ.

Since the linear probing hash table stores only the slot array, we can immediately infer the following.

Corollary 1. *The hash table implementation described above is uniquely represented.*

Space and Time Complexity. The hash table implementation based on linear probing requires $p = n/\alpha$ slots to store n keys and requires no auxiliary memory other than that used to store and compute the hash function. As we will show, the expected cost for all operations is $O(1/(1-\alpha)^3)$.

We bound the expected time per operation for our hash table implementation by comparing it to a standard linear probing hash table. Recall that this standard hash table selects a hash function $h(\cdot)$, uses probe sequences $\text{probe}(\mathbf{k}, i) = h(\mathbf{k}) + i \mod (p)$, and resolves all collisions in favor of the key already residing in the contested slot. In a recent breakthrough result, Pagh *et al.* [PPR07] showed that linear probing with 5-universal hash functions yields expected $O(1/(1-\alpha)^3)$ time operations.

Theorem 4. *The linear probing hash table implementation described above performs searches, insertions, and deletions in expected $O(1/(1-\alpha)^3)$ time, where the hash table has p slots and $n = \alpha p$ keys.*

Proof: First consider only searches and insertions. Fix a set of keys S of size at most n. It is easy to see that if the standard hash table and our hash table use the same hash function $h(\cdot)$, then after inserting S (using any sequence of operations that does not contain delete operations to do so) both hash tables will have exactly the same set of occupied slots, even though they likely have different memory representations. Note that for the standard hash table the cost to insert $\mathbf{k} \notin S$ is $\Theta(d_S^h(\mathbf{k}))$, where $d_S^h(\mathbf{k})$ is one plus the smallest i such that slot $(h(\mathbf{k}) + i) \mod (p)$ is unoccupied. It is not hard to see that in our hash table, during insertion of $\mathbf{k}$ the while loop is executed at most $d_S^h(\mathbf{k})$ times, and each iteration takes constant time. Thus if the standard table takes time t to insert $\mathbf{k}$ after a sequence of operations ρ, our hash table takes $t' = O(t)$ time. Using the result of Pagh *et al.* [PPR07], $\mathbf{E}[t'] = O(\mathbf{E}[t]) = O(1/(1-\alpha)^3)$.

Searching for $\mathbf{k} \notin S$ takes the same amount of time as inserting $\mathbf{k}$, up to multiplicative constants. Searching for $\mathbf{k} \in S$ similarly takes less time than inserting $\mathbf{k}$,

assuming we have inserted all keys in $S \setminus \mathbf{k}$ first. So this is expected $O(1/(1-\alpha)^3)$ time as well.

Now consider deletions. Suppose we insert a set of keys S and then delete key $\mathbf{k} \in S$. We can compute $\mathrm{rank}(\cdot)$ in constant time. Looking at the pseudocode for delete and $\mathrm{next}(\cdot)$, it is easy to prove that $\mathrm{delete}(\mathbf{k})$ takes time $O(d_S^h(\mathbf{k}))$. So, as before, we consider inserting all elements of $S \setminus \mathbf{k}$ before inserting $\mathbf{k}$, and then inserting $\mathbf{k}$ last before deleting it. By our analysis above, the $\mathrm{insert}(\mathbf{k})$ operation takes time $O(d_S^h(\mathbf{k}))$ which is $O(1/(1-\alpha)^3)$ in expectation, so the deletion takes $O(1/(1-\alpha)^3)$ in expectation as well. Note that for a hash table which is not uniquely represented this line of reasoning is invalid, because changing the order of insertions might change the memory representation of the hash table and conceivably reduce the amount of time the delete operation takes. However we may safely dispense with this concern because our hash table is uniquely represented. ■

Remark: Our result is modular in the following sense. If linear probing with a hash function drawn from a hash family $\mathcal{H}$ results in $f(\alpha, n)$ expected time for insertions, then using a hash function drawn from $\mathcal{H}$ in our construction results in $O(f(\alpha, n))$ searches, insertions, and deletions.

Dynamic Resizing. We can dynamically resize the hash table using the standard technique of doubling its size and rehashing all keys upon reaching a threshold number of keys. For good performance against an adversary, we select the threshold randomly, as done in previous work [HHM+05, NT01].

3.2 Uniquely Represented Perfect Hashing

Building on the work of Fredman *et al.* [FKS84], Dietzfelbinger *et al.* [DKM+94] gave a hash table with $O(1)$ *worst case* time for lookups and $O(1)$ amortized expected time for insertions and deletions, while using space linear in the number of keys. Naor and Teague [NT01] then built on the work of Dietzfelbinger *et al.*, and developed a weakly history independent hash table with the same performance guarantees. More recently, Pagh and Rodler [PR04] developed a different technique to obtain the same guarantees, called *cuckoo hashing*. In this section we will present a third way to achieve identical time and data-space bounds while maintaining unique representation (and thus strong history independence), assuming

our machine has access to a large sequence of random bits. Unfortunately, for reasons which we will discuss later, we cannot sample random bits "on demand." On the other hand, our approach is novel and relatively simple.

3.2.1 Our Construction Based on Linear Probing

For ease of exposition, we begin with an impractical design that requires exponentially many random bits, and afterwords describe how to modify it for practical use. We will assume that the number of keys to be stored, n, is known in advance[2].

Theorem 5. *There exists a uniquely represented hash table that executes insertions and deletions in expected $O(1)$ time, executes search in worst case $O(1)$ time, and uses $O(n)$ data space to store n keys.*

We will describe a simplified version of the hash table that works assuming various low probability events do not occur, and then address these problematic events.

The Simplified Version. We begin with an array with $p = c_0 n$ slots to store the keys, where $c_0 > 1$ is a constant, and a fast $\Omega(\log n)$-universal hash function h mapping keys to slots. We will insert and delete keys from the hash table as we did in Section 3.1 using linear probing, but will need to maintain some additional state. Let $\delta(\mathbf{k})$ denote the *displacement* of key $\mathbf{k}$ in the hash table, which, if the current location of $\mathbf{k}$ is x, is defined as $(x - h(\mathbf{k})) \bmod (p)$. Fix a parameter $\beta = \Theta(\log n)$, and maintain a fast 2-universal hash function f from keys to a set of labels $\mathcal{L} := \{1, 2, \ldots, |\mathcal{L}|\}$, where $\beta^3 \leq |\mathcal{L}| = \text{poly}(\beta)$. Each key $\mathbf{k}$ receives a label $f(\mathbf{k})$ when inserted into the hash table. Each slot x will have an *index* I_x associated with it. The index for x will store tuples $(\delta(\mathbf{k}), f(\mathbf{k}))$ for each key $\mathbf{k}$ such that $h(\mathbf{k}) = x$, in sorted order.

Let us assume for the moment that the displacement of any key is $O(\log n)$. (This occurs with high probability if the hash function is truly random [Jan05].) In this case, each displacement requires only $O(\log \log n)$ bits to store. Note that the labels require only $O(\log \log n)$ bits to store as well. It is well known that using an $c \log n$-universal hash function (with c sufficiently large) to hash n keys into $p \geq (1 + \epsilon)n$ slots (for $\epsilon > 0$) ensures that at most $O(\log n / \log \log n)$ keys are

[2]We can dispense with this assumption using a uniquely represented variant of the standard resizing technique, as in Section 3.1.

hashed to any one slot with high probability. Assuming this is the case, for each slot x we can store its index I_x using a word-packed vector of only $O(\log n)$ bits. (For simplicity, we will require all index tuples to use the same number of bits.) Word-level parallelism then allows us to perform various queries and updates to the index in constant time. For example, we can insert and delete tuples in constant time, even while maintaining a canonical form (that is, the records should be maintained in sorted order, and if r records are stored in I_x, they must be stored in the first r spaces in the index). Thus, in the course of inserting and deleting keys, we can amortize the cost of updating the index tuples for each key which moved against the cost of moving it. We can also answer in constant time queries of the form "for what values of d is there a tuple of the form (d, l) in I_x" for any $l \in \mathcal{L}$. Assuming the labels for each key hashed to slot x are distinct, we can use this fact to do searches in constant time as follows. Find all d such that $(d, f(\mathbf{k}))$ is in $I_{h(\mathbf{k})}$. Since the labels in I_x are distinct, the output contains at most one value for displacement, say d, at which point we may immediately test the slot $(x + d) \bmod (p)$ to see if it contains key **k**.

The Full Version. In the simplified version we made three assumptions that are false in general, namely we fixed constants c_1 and c_2 and assumed

1. No slot has more than $c_1 \frac{\log n}{\log \log n}$ keys hashed to it.
2. No key has a displacement more than $c_2 \log n$.
3. For all slots x, the keys hashed to x get distinct labels.

To remove assumption $\#1$, it is tempting to simply sample a new the hash function using fresh randomness whenever it is violated. However, we cannot do this in a naive way and maintain unique representation. Instead, we maintain a random permutation π^{hash} on an $\Omega(\log n)$-universal family $\mathcal{H}$ of hash functions mapping the keys to the p slots. The idea is that we will use the first hash function, π_0^{hash}, in π^{hash} until and unless assumption $\#1$ is violated. In that case we will iterate through the hash functions $\{\pi_i^{\text{hash}} : i = 0, 1, \ldots\}$, rehashing everything using the current hash function π_i^{hash} until we find the first one which satisfies the assumption (and the "no block overflow assumption," explained in the treatment of assumption $\#2$ below). We will denote the current hash function by h. To maintain unique representation, we will need to take extra precautions during a deletion if the current hash function is not π_0^{hash}. Specifically, we will need to completely clear the

hash table, reset the current hash function to π_0^{hash}, and reinsert every key (other than the one to be deleted) from scratch. (Note that it would not be uniquely represented to simply iterate backwards through π^{hash} and stop at π_{i+1}^{hash} if π_i^{hash} is the first hash function we encounter that violates assumption $\#1$.)

We remove assumption $\#2$ by essentially treating the hash table as $\Theta(n/\log n)$ hash tables of size $\Theta(\log n)$. We partition the slots into *blocks*, each block will be a contiguous set of $\beta = \Theta(\log n)$ slots (assume p is a multiple of β). Keys hashed into a block do not leave it, unless h is changed. Rather, the probe sequence wraps around the block boundaries. Formally, if the block B contains slots in the range $[a, b]$, then the probe sequence for a key $\mathbf{k}$ with $h(\mathbf{k}) \in [a, b]$ is given by $\text{probe}(\mathbf{k}, i) := a + ((h(\mathbf{k}) - a + i) \mod (\beta))$. This ensures the displacement of any key is at most β, assuming the block has at most β keys hash into it. Call this the "“no block overflow assumption.” If π_0^{hash} hashes more than β keys to some block, we will ensure each block has at most β keys in it by iterating through the hash functions $\{\pi_i^{\text{hash}} : i = 1, \ldots\}$, rehashing everything using the current hash function π_i^{hash} until we find the first one which satisfies the no block overflow assumption (as well as assumption $\#1$).

To remove assumption $\#3$, we store a random permutation π^{label} on a 2-universal family $\mathcal{H}'$ of hash functions mapping the keys to a set of labels $\mathcal{L}$, where $\beta^3 \leq |\mathcal{L}| \leq \text{poly}(\beta)$. Each block B will store a pointer to a hash function in π^{label}, which we will call its *label function*, and denote by f_B. Each block's label function initializes to π_0^{label}, and we maintain the invariant that f_B is the first hash function in π^{label} that gives distinct labels to every key which has been hashed to x, for each $x \in B$. Again, we will need to be careful with deletions. Upon deleting a key $\mathbf{k}$ in block B for which $f_B \neq \pi_0^{\text{label}}$, we iterate through $\{\pi_i^{\text{label}} : i = 0, 1, \ldots\}$, relabeling all the keys hashed into block B (excluding $\mathbf{k}$) using π_i^{label} until we find a label function that satisfies assumption $\#3$ for all slots in B with this set of keys.

Analysis. It is relatively straightforward to prove that the above data structure is uniquely represented given the results of Section 3.1; we simply need to show that the hash table state is a function of the set of keys and the random bits of the hash table. We omit the details.

The space usage of the data structure is $O(n)$ if we exclude the random bits. The slots and their indices use $O(n)$ space. Using, for example, the hash functions of Östlin and Pagh [OP03] or Dietzfelbinger and Woelfel [DW03], the single pointer into π^{hash} requires $O(n)$ words of space, and each pointer into π^{label}, of which there

are $O(n/\log n)$, requires $O(1)$ words of space each.

We now analyze the running time of each operation. First, consider a search for key $\mathbf{k}$. By construction, assumptions #1 through #3 hold. Thus in the worst case we need only evaluate $x := h(\mathbf{k})$, compute $\mathbf{k}$'s label l, find any record of the form (d, l) that may exist in I_x, compute $y := \text{probe}(\mathbf{k}, d)$, and determine if $\mathbf{k}$ is in slot y. All of these are constant time operations.

Next, consider insertions and deletions of a key $\mathbf{k}$ under "best case circumstances", which we define to mean that $h = \pi_0^{\text{hash}}$, $f_B = \pi_0^{\text{label}}$ where block B contains $h(\mathbf{k})$, and at most $\beta/2$ keys have been hashed into B. In this case, we can amortize each index update (a constant time operation) against a specific operation that moves a key. Thus we can ignore their cost. Note that inserting or deleting $\mathbf{k}$ costs essentially the same as the corresponding operation into a hash table with β slots and at most $\beta/2$ keys, using an $\Omega(\beta)$-wise independent hash function – that is, expected constant time. Now relax the condition that a block B has at most $\beta/2$ keys, and allow it to have up to β keys. The cost to do any operation is bounded by β in this case. Let $|B|$ denote the number of keys in B. We will argue that $\mathbf{Pr}[|B| \geq \beta/2] \leq 1/n^c$ if the parameters are set appropriately, and thus the cost contribution from this case is negligible. We set $p = 4e \cdot n$, $\beta = 2c \log n$, $\gamma := \beta/2$ and use γ-universal hash functions in π^{hash}. Then we get the following estimate: $\mathbf{Pr}[|B| \geq \gamma] \leq \binom{n}{\gamma} \left(\frac{\beta}{p}\right)^{\gamma} \leq n^{-c}$.

Next, we consider the contribution of various bad events to the expected running time. Consider insertions. The probability that we must rehash everything using the next hash function in π^{hash} can be made as small as $O(n^{-c})$ for any constant $c > 0$ by adjusting various parameters, since this only occurs when more than $c_1 \log n / \log\log n$ keys hash to some slot x, which occurs with probability $O(n^{-c})$, or when some block B gets more than β keys, which we have argued above occurs with probability $O(n^{-c})$. Thus we only have to rehash everything with probability $O(n^{-c})$. Since this takes $O(n)$ expected time, its contribution to the expected running time is negligible. For a set of keys S, call a hash function in π^{hash} *bad for* S if it hashes more than $\beta/2$ keys from S into some block or more than $c_1 \log n / \log\log n$ keys into some slot. Fix S with $|S| \leq n$, let $\varepsilon(S, i)$ denote the event that π_i^{hash} is bad for S, and note that for any i,

$$\mathbf{Pr}\left[\varepsilon(S,i) \;\middle|\; \bigcap_{j<i} \varepsilon(S,j)\right] \leq \mathbf{Pr}[\varepsilon(S,i)]$$

because by the principle of deferred decisions we can imagine selecting π_i^{hash} after

$\{\pi_j^{\text{hash}} : j < i\}$, and noting that if the concentration of "bad for S" sets is higher than average among $\{\pi_j^{\text{hash}} : j < i\}$, then it must be lower than average among the remaining hash functions, from which we draw π_i^{hash}. Using this fact, we conclude that the total expected time to do all the rehashing is $O\left(\sum_{t\geq 1} tn/n^{ct}\right) = o(1)$.

The expected cost of rehashing after a deletion can be bounded similarly. That is, the probability that $h = \pi_t^{\text{hash}}$ is $O(n^{-ct})$, and the work involved in this case to rehash everything up to t times is $O(tn)$. The total expected time to do the rehashing is thus $O\left(\sum_{t\geq 1} tn/n^{ct}\right) = o(1)$.

Finally, we need to bound the cost due to relabeling within a block. Fix a block B, and let S be the set of keys being stored in B. If a 2-universal hash function f_B is used to generate the labels, then we can bound the probability that all keys in B get distinct labels as follows. Via the union bound the probability that two keys get the same label (an event we call a *label collision*) is bounded by

$$\textbf{Pr}[\text{label collision}] \;\leq\; \sum_{\mathbf{k},\mathbf{k}'\in S} \textbf{Pr}\big[f_B(\mathbf{k}) = f_B(\mathbf{k}')\big]$$

Since f_B is 2-universal, for any distinct $\mathbf{k}, \mathbf{k}' \in S$,

$$\begin{aligned}\textbf{Pr}\big[f_B(\mathbf{k}) = f_B(\mathbf{k}')\big] &= \textstyle\sum_{\ell=1}^{|\mathcal{L}|} \textbf{Pr}\big[f_B(\mathbf{k}) = \ell \text{ and } f_B(\mathbf{k}') = \ell\big] \\ &= \textstyle\sum_{\ell=1}^{|\mathcal{L}|} 1/|\mathcal{L}|^2 \\ &= 1/|\mathcal{L}|\end{aligned}$$

Thus

$$\textbf{Pr}[\text{label collision}] \;\leq\; \binom{|S|}{2}\frac{1}{|\mathcal{L}|} \;\leq\; \frac{\beta(\beta-1)}{2|\mathcal{L}|}.$$

Since we can relabel all $|S| \leq \beta$ keys in block B in $O(\beta)$ time, and setting $|\mathcal{L}| \geq \beta^3$ ensures that $\textbf{Pr}[\text{label collision}] = O(1/\beta)$, the expected work to relabel everything once is $O(1)$. By a similar argument as above, we can bound the probability that we relabel everything in a block t times by $O(\beta^{-t})$, so the total expected work from relabeling is $O\left(\sum_{t\geq 1} t\beta/\beta^t\right) = O(1)$.

3.2.2 Practical Variants

Unfortunately, we cannot resort to sampling random bits "on demand." To see why, suppose we repeatedly insert keys S, then delete them, and S requires the hash

table to sample new random bits. Whether or not we retain these new random bits after deleting S, we will violate unique representation. Thus we are forced to do all the sampling we will ever need to do during data structure initialization. This will require access to a possibly exponential sized sequence of random bits. We ensure the number of random bits we need is finite by sampling new hash functions without replacement – in other words, we sample a random permutation on a suitable hash family.

This is still hopelessly impractical. The saving grace is that our data structures can be made to inspect only $O(n^\delta)$ random words with high probability, for any constant $\delta > 0$, if we use the hash functions of Siegel [Sie95a] in π^{hash} and those of Östlin and Pagh [OP03] or Dietzfelbinger and Woelfel [DW03] for π^{label}. This suggests the following approach: set thresholds $\tau^{\text{hash}}, \tau^{\text{label}} \in \mathbb{N}$, and only sample the first τ^{hash} elements of π^{hash} and the first τ^{label} elements of π^{label}. Reasonable values would be $\tau^{\text{hash}} = 1$ and $\tau^{\text{label}} = \Theta(\log n)$. Then, if the data structure should ever need to access a hash function that has not been sampled, enter one of two failure modes.

The first failure mode is to sample fresh random bits on demand from now on. The resulting data structure is uniquely represented with high probability, weakly history independent with certainty, and has the same running time guarantees as before. Furthermore, if we additionally store the fresh random bits and treat them as the needed elements of π^{hash} and/or π^{label}, then other than the random bits, the data structure is uniquely represented. Thus the *only* historical information that might be inferred in this case is that something was inserted and later deleted that forced the data structure to sample additional random bits.

The second failure mode is to store everything as in the uniquely represented hash table of Section 3.1. The whole data structure then remains uniquely represented with certainty, provided on each delete we reinsert everything from scratch to determine if we should still be operating in this failure mode. The expected time guarantees for insertions and deletions still hold, however searches now take $O(1)$ time with high probability rather than with certainty as before.

Another approach to making our dynamic perfect hash table more practical is to use a secondary data structure, or *stash*, to store keys that are "problematic" for the hash functions we have chosen. This approach is taken by Naor, Segev, and Wieder [NSW08] in their construction, as we discuss in the next section. For example, if a slot x has $t > c_1 \frac{\log n}{\log \log n}$ keys hashed to it, thus violating assumption $\#1$, we can store the largest $t - c_1 \frac{\log n}{\log \log n}$ keys hashed to x in the stash. (Note that

we assume the keys have a natural total order on them.) Similarly, if a block B overflows, so that $t > \beta$ keys are hashed into it, we can store the largest $t - \beta$ keys hashed to B in the stash. We allow the keys to be relabeled in the event of a label collision as before. With high probability the stash will be constant sized, and even a small stash would go a long way towards making rehashing extremely unlikely. As for implementations of the stash, any of those suggested by Naor, Segev, and Wieder [NSW08] would be suitable, including our dynamic perfect hash table and various static perfect hash tables.

Remark. Note that this approach is fairly general. It provides a method to convert impractical uniquely represented data structures with worst case guarantees (and require a random permutation over a suitable hash family) into more practical data structures that either

(i) Retain the running times but make the data structure uniquely represented with high probability and weakly history independent with certainty, or

(ii) Retain unique representation but replace worst-case time guarantees with "with high probability" time guarantees.

3.2.3 Comparison to Uniquely Represented Cuckoo Hashing

Naor, Segev, and Wieder [NSW08] developed a uniquely represented dynamic perfect hash table which support deletions based on the *cuckoo hashing* scheme of Pagh and Rodler [PR04]. Their hash table supports insertions and deletions in expected constant time and lookups in worst case constant time assuming truly random hash functions that can be evaluated in constant time. (The hash functions of Östlin and Pagh [OP03] or Dietzfelbinger and Woelfel [DW03] are suitable substitutes). Let us review the (conventional) cuckoo hashing scheme.

In the cuckoo hashing scheme, there are $d \geq 2$ hash functions $h_1, h_2, \ldots, h_d$ from the universe of keys $\mathcal{U}$ to a set of slots $[p]$. Let $H(\mathbf{k}) := \{h_i(\mathbf{k}) : 1 \leq i \leq d\}$. Each key $\mathbf{k}$ stored in the hash table is guaranteed to be stored in a slot in $H(\mathbf{k})$. This ensures only d probes are needed to lookup any key, and thus yields a worst case $O(1)$ time guarantee for lookups. Deletions simply lookup the key and remove it if it is present. The most complex operation is insertion. If when inserting $\mathbf{k}_0$ there is some empty slot in $H(\mathbf{k}_0)$, simply place $\mathbf{k}_0$ there. Otherwise, the hash table must select some slot $x_1 \in H(\mathbf{k}_0)$ storing a key $\mathbf{k}_1$, evict $\mathbf{k}_1$ and place $\mathbf{k}_0$ there. We

then place $\mathbf{k}_1$ as if it had just been inserted, being careful to evict some key other than $\mathbf{k}_0$ if assigning $\mathbf{k}_1$ to some slot in $H(\mathbf{k}_1)$ requires an eviction. This process is repeated until some evicted key $\mathbf{k}_t$ has an unoccupied slot in $H(\mathbf{k}_t)$ for it. There is some small probability that this process will not terminate, for example if some subset S of keys in the hash table satisfies

$$|\bigcup_{\mathbf{k}\in S} H(\mathbf{k})| < |S|.$$

However, if $d \geq 2$ this probability is $O(1/n)$ and in general this problem can be dealt with by storing some elements in a secondary data structure (often called a *stash*).

Naor, Segev, and Wieder construct a uniquely represented version of a cuckoo hash table (with $d = 2$) intended to store up to n keys using two tables T_1, T_2 of size $p = (1 + \epsilon)n$. For each $i \in \{1, 2\}$, the hash function h_i maps keys to slots in T_i. Given a set of keys S, they define the *cuckoo graph* $G = (V, E)$ where V consists of the slots of T_1 and T_2 and $E = \{\{h_1(\mathbf{k}), h_2(\mathbf{k})\} : \mathbf{k} \in S\}$. Let $e(\mathbf{k}) := \{h_1(\mathbf{k}), h_2(\mathbf{k})\}$. Assume there is a total order on the universe of keys. For a connected component C of G, define $\mathrm{keys}(C)$ to be the set of keys such that C contains the edges $\{e(\mathbf{k}) : \mathbf{k} \in \mathrm{keys}(C)\}$. Similarly define $\mathrm{keys}(E')$ for a set of edges E'. Naor, Segev, and Wieder ensure unique representation by enforcing the following constraints.

1. For a connected component C of G that is acyclic, store the minimum key $\mathbf{k}$ in $\mathrm{keys}(C)$ in both $h_1(\mathbf{k})$ and $h_2(\mathbf{k})$, and store the remaining keys in $\mathrm{keys}(C)$ in the unique way that satisfies the constraint that each key $\mathbf{k}'$ is stored in a slot in $H(\mathbf{k}')$.

2. For a connected component G that has exactly one cycle C, store the minimum key $\mathbf{k}$ in $\mathrm{keys}(C)$ in $h_1(\mathbf{k})$, and store the remaining keys in the component in the unique way that satisfies the constraint that each key $\mathbf{k}'$ is stored in a slot in $H(\mathbf{k}')$.

3. For a connected component C of G that has more than one cycle, repeatedly store the maximum key in some cycle in the stash until the remaining elements in C have at most one cycle. Then store the remaining elements as described above, depending on whether the remaining elements induce a component with one or zero cycles.

4. The stash is a uniquely represented dictionary supporting lookups in worst case $O(1)$ time, and updates in polynomial time.

This cuckoo hashing construction avoids the rehashing used in our dynamic perfect hash table. However, both uniquely represented dynamic perfect hash tables perform updates in at most $O(\log n)$ time with high probability, so rehashing is very infrequent. As mentioned in Section 3.2.2, the probability of rehashing can be significantly reduced by augmenting our table with a stash. Our hash tables are also significantly simpler, and the construction of Section 3.1 requires only 5-wise independent hash functions (which can be sampled using only $O(\log n)$ random bits) to guarantee expected constant time updates and lookups. Ultimately, both uniquely represented hash tables based on linear probing and those based on cuckoo hashing will likely prove useful in applications, just as both linear probing and cuckoo hashing are used today.

3.3 Uniquely Represented Memory Allocation

The most basic structure that is required throughout this dissertation is a hash table with insert, delete and lookup operations. The most common use of hashing in this dissertation is for memory allocation. Traditional memory allocation depends on the history since locations are allocated based on the ordering in which they are requested. We maintain data structures as a set of *blocks*. Each block has its own unique integer label which is used to hash the block into a unique *memory cell*. It is not too hard to construct such block labels if the data structures and the basic elements stored therein have them. For example, we can label points in $\mathbb{R}^d$ using their coordinates and if a point p appears in multiple structures, we can label each copy using a combination of p's label, and the label of the data structure containing that copy. For more information on labeling schemes, see Section 5.2.

This representation of memory contains no traditional pointers but instead uses labels as "abstract pointers." For example for a tree node with label l_p, and two children with labels l_1 and l_2, we store a cell containing (l_1, l_2) at label l_p. This also allows us to focus on the construction of data structures whose *pointer structure* is uniquely represented; such structures together with this memory allocation scheme yield uniquely represented data structures in a RAM. Note that nearly all of the tree structures we use in later chapters have pointer structures that are uniquely represented. Given the memory allocation scheme above, the proofs that our data structures are uniquely represented are thus quite straightforward and we will often omit them.

3.4 Experimental Evaluation

In this section we present the results of some modest experiments on a proof-of-principle implementation of the hash table described in Section 3.1, which we denote by `URHashMap`. We compare the `URHashMap` against the standard Java `HashMap` data structure. In particular, we used the Java™ 2 runtime environment, standard edition, version 1.5.0. The problem of garbage collection occurring at uncontrolled times was mitigated by ensuring that only one hash map resides in memory at one time, and specifying the capacity of that hash map on initialization. The `System.gc()` command was also used; this command encourages (but does not actually force) the system to garbage collect when it is issued. Timing was performed using `System.currentTimeMillis()`, which has a temporal resolution of 1 millisecond on the Linux workstation on which the tests were run.

The `URHashMap` implementation supported a generic map from keys to values, where keys must implement a hashable interface – that is, they must support a function that provides an integer label for hashing. It used essentially the same hash function that the Java `HashMap` uses which is quite simple and works well in practice, rather than more complex high performance hash functions (e.g., those in [TZ04]). In general the `URHashMap` was implemented with simplicity rather than high performance in mind, and further optimizations are possible.

Here is code for the hash function that the `URHashMap` used[3].

```
public int hash(int x){
  // this hash function requires that the capacity is a power of two.
  x +=  ~(x << 9);
  x ^=  (x >>> 14);
  x +=  (x << 4);
  x ^=  (x >>> 10);
  return x &= (capacity-1);
}
```

Both the `URHashMap` and the Java `HashMap` were tested while instantiated with type `Integer` → `Integer`, where `Integer` is the Java wrapper class for integers.

[3]The source code from which this function is derived is available from Sun Microsystems via a Java Research License. Permission to publish "relevant excerpts that do not in the aggregate constitute a significant portion of the Technology" in scholarly publications is granted by Section III.A.3 of the Java Research License version 1.5.

For both the `URHashMap` and the Java `HashMap`, a map of capacity $C := 2^{20} \approx 10^6$ was created. The total time to insert xC keys was measured, for all $x \in \{0.01, 0.02, 0.03, \ldots, 0.99\}$. Similarly, the total time to lookup all keys in a map storing xC keys was measured for all $x \in \{0.01, 0.02, \ldots, 0.99\}$, as was the time to delete all keys stored in the map. The results, averaged over 20 trials, are displayed in Figures 3.3 through 3.5 on page 48. Figure 3.6 depicts the (time) performance ratio between the `URHashMap` and the Java `HashMap` for insertions, lookups, and deletions. The figures reveal that at loads at or below 50%, the the `URHashMap` performs insertions faster than the Java `HashMap` – about 50% faster at a load of 40% – and performs lookups and deletions about 50% slower than the Java `HashMap`. At higher loads the relative performance of the `URHashMap` degrades, so that with an 80% load it takes between 1.5 and 2 times as long for all operations. However, it is worth noting that the typical load for a hash table based on open addressing is far less than 80%, and the performance of the `URHashMap` is in fact better than these figures would suggest because the Java `HashMap` is implemented using *separate chaining* and uses more space than the `URHashMap` as the load factor approaches one.

Recall that in separate chaining each hash table slot contains a linked list, and key collisions are resolved by storing all keys hashing to a particular slot in its corresponding list. Thus a map with m slots and n entries, each of which is a pair of integers, will require $m+3n$ words of space if implemented via separate chaining – one pointer per slot, and two integers and one pointer per entry. In contrast, the `URHashMap` is based on open addressing and thus requires $2m$ words of space – two per slot. Thus if a separate chaining implementation $\mathcal{I}$ stores n entries of type `Integer` $\rightarrow$ `Integer` with load α and an open addressing implementation stores n entries using the same amount of space as $\mathcal{I}$ but with load α', then $\frac{n}{\alpha} + 3n = \frac{2n}{\alpha'}$ or equivalently

$$\alpha' = \frac{2\alpha}{1 + 3\alpha}.$$

Assume that the the expected time to perform an operation in the `URHashMap` is a function of the load alone, so that, for example, inserting a key into a hash table with $2n$ slots and n keys takes the same amount of time as inserting a key into a hash table with $4n$ slots and $2n$ keys. This is true in the limit as $n \rightarrow \infty$ and is reasonable for hash maps with more than a few hundred elements. Using this assumption, we can define the *space adjusted performance* of the `URHashMap` against separate chaining hash maps. The space adjusted performance of the `URHashMap` is displayed in Figures 3.3 through 3.5 on page 48 as a blue broken line. Given a load x for the separate chaining implementation $\mathcal{I}$ with n keys, compute the time per

operation for the `URHashMap` at load $\alpha'(x) := \frac{2x}{1+3x}$ and multiply it by n to obtain an upper bound on the time to insert n keys into a `URHashMap` using the same amount of space as $\mathcal{I}$. Since the time per operation for the `URHashMap` is a non-decreasing function of the load, we use the time per operation at load $\alpha'(1) = \frac{1}{2}$ as an upper bound on the time per operation at lower loads.

The space adjusted performance ratios are displayed in Table 3.2. For example, at the Java `HashMap` default load of 0.75, the `URHashMap` is roughly 29% faster on insertions, 41% slower on lookups, and 46% slower on deletions.

Performance Ratios at Various Loads			
	50%	**75%**	**100%**
insert	0.722	0.773	1.03
lookup	1.48	1.41	1.27
delete	1.61	1.46	1.51

Table 3.2: The space adjusted performance ratios of the `URHashMap` against that of a separate chaining implementation at loads of 50%, 75%, and 100%. The corresponding loads for the `URHashMap` are 40%, approximately 46%, and 50%, respectively.

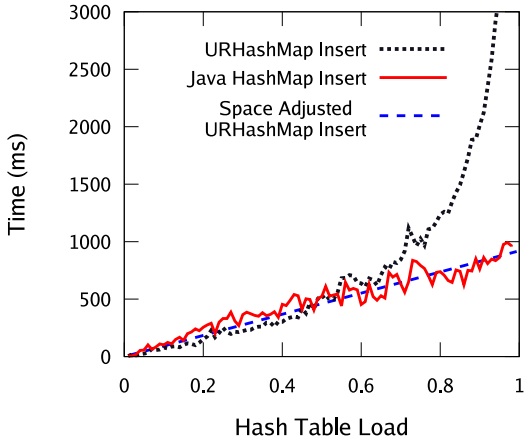


Figure 3.3: The total time to insert $x \cdot 2^{20}$ keys into a map of capacity 2^{20}, displayed as a function of x.

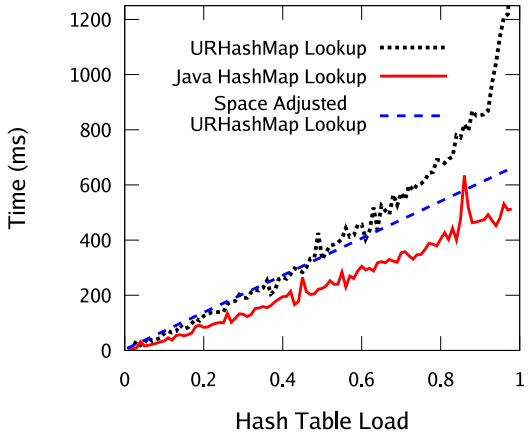


Figure 3.4: The total time to lookup all keys in a map of capacity 2^{20} with $x \cdot 2^{20}$ keys stored in it, as a function of x.

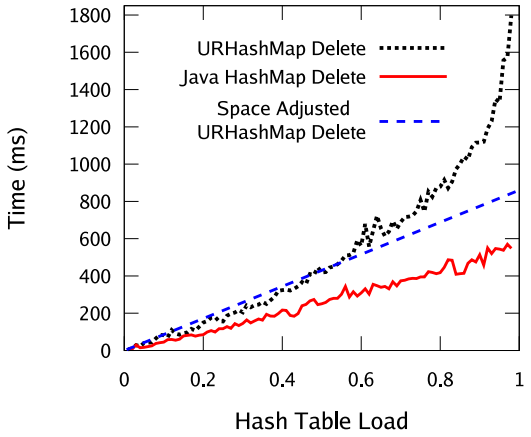


Figure 3.5: The total time to delete all keys from a map of capacity 2^{20} with $x \cdot 2^{20}$ keys stored in it, as a function of x.

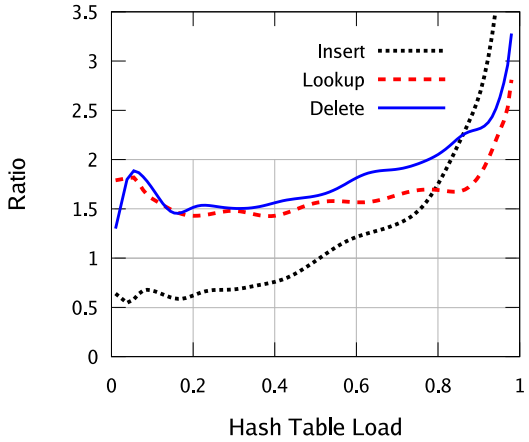


Figure 3.6: The raw performance ratios of the `URHashMap` and the Java `HashMap` for insertions, lookups, and deletions, based on the data displayed in Figures 3.3 through 3.5 and suitably smoothed.

Chapter 4

Basic Data Structures

In this chapter we discuss efficient uniquely represented implementations of various elementary abstract data types. To simplify the exposition, we consider only statically allocated data structures here, and defer the treatment of dynamically allocated data structures to Chapter 8. That is, we assume that a contiguous block of memory is allocated to the data structure in question on initialization, and the memory locations allocated to the data structure do not change over time. For concreteness, this can be achieved by having just one data structure on a RAM and allocating the entire RAM memory to it.

The data structures presented here have many variants, so we will define the relevant abstract data types carefully. For further information about these abstract data types and their various implementations, we refer the reader to [AHU83, CLRS01].

4.1 Arrays

The *array* abstract data type provides a finite set S of *slots*, where each slot is labeled with a distinct integer in $\{0, 1, 2, \ldots, |S| - 1\}$ called its *index*. Once initialized, the array supports the following operations.

- $\mathrm{length}()$: return the length of the array, $|S|$.
- $\mathrm{read}(x)$: return the contents of array slot with index x.
- $\mathrm{write}(x, v)$: set the contents of array slot with index x to v.

The read and write operations are valid for $x \in \{0, 1, 2, \ldots, |S| - 1\}$. Of course, the RAM model of computation natively supports arrays, with all operations taking worst-case constant time. The RAM's memory is in fact a large array whose slots can each store a *word* (i.e., bit-strings of fixed length). Furthermore, it is easy to see that the native implementation of arrays provided by the RAM is uniquely represented, if we neglect the problem of memory allocation. Fortunately, the array corresponding to the RAM's entire memory need not be dynamically allocated, and we can use this giant array as the foundation for a uniquely represented memory allocator built from a uniquely represented hash table of Chapter 3. In this way we can bootstrap our way to dynamically allocated uniquely represented data structures as described in Chapter 8.

4.2 Stacks

The *stack* abstract data type stores a list of objects and supports the following operations.

- $\mathrm{push}(x)$: append x to the end of the list.
- $\mathrm{pop}()$: remove the last object in the list and return it.
- $\mathrm{length}()$: return the length of the list.

A *bounded stack* is a stack with an upper limit on the length of its list. If the objects to be stored in the stack have a fixed size, these operations can be implemented in worst-case constant time. Furthermore, in this case there is a particularly simple implementation of a bounded stack that is uniquely represented. This implementation stores a bounded stack of length at most n in an array of length n, and simply stores the i^{th} element of the list in the slot with index i. It also stores the length of the stack in an integer field. To ensure this implementation is uniquely represented, we need only make sure to clear the array slot storing the last list element during a $\mathrm{pop}()$. We thus obtain the following result.

Proposition 1. *There exists a uniquely represented implementation of a bounded stack whose memory is statically allocated and stores elements of a fixed size, such that the above operations run in $O(1)$ worst-case time.*

4.3 Linked Lists

The *linked list* abstract data type stores an ordered list with the operations listed in Table 4.1. The running times are from a conventional doubly-linked list implementation.

Name	Description	Running Time
$\text{first}()$	Return the first element of the list	$O(1)$ time
$\text{last}()$	Return the last element of the list	$O(1)$ time
$\text{next}(x)$	Return the list element following list element x	$O(1)$ time
$\text{previous}(x)$	Return the list element preceding list element x	$O(1)$ time
$\text{insert}(x, y)$	Insert y immediately following list element x	$O(1)$ time
$\text{delete}(x)$	Delete list element x	$O(1)$ time

Table 4.1: The linked list operations, with the running times for a conventional implementation.

We postpone the problem of dynamic memory allocation by restricting our attention to lists of bounded size, and whose list elements can be stored in a constant number of words. The latter requirement is not very restrictive, in the sense that typical link lists of large objects will store the pointers to those objects rather than the objects themselves. We show how to implement dynamically allocated link lists of dynamically allocated objects without adversely affecting the running time guarantees in Chapter 8.

Our linked list implementation is initialized with a capacity N, which denotes the maximum number of list elements allowed at any one time. We will also require that each list element be distinct and *hashable* in the following sense.

Definition 7 (Hashable)**.** *A set of elements E is* hashable *if there is a family of hash functions $\mathcal{H}$ from E to $\mathbb{N}$ which is sufficiently independent, whose hash functions can be evaluated in constant time, and which can be efficiently sampled from. (The sufficient level of independence depends on the hash tables used and the corresponding running time guarantees that they result in. Typically, 5-wise independence will do.)*

Note that since we have assumed the list elements can be stored in $O(1)$ words, it is possible to treat them as integers and thus they are hashable. We will prove the following.

Proposition 2. *There exists a uniquely represented implementation of a bounded linked list whose memory is statically allocated and stores elements of a fixed size, such that* $\mathrm{first}()$ *and* $\mathrm{last}()$ *take* $O(1)$ *worst-case time, and all other operations in Table 4.1 take* $O(1)$ *expected time. Furthermore, this implementation requires only* $O(\log N)$ *random bits, and requires only* $O(N)$ *space to store a linked list of capacity* N.

Our implementation consists of two list element fields, first and last, and an array A of $(1+\epsilon)N$ slots for some $\epsilon > 0$, where each slot is large enough to hold a list element and two integers. These may be allocated in order on the available RAM memory, as shown below.

first	last	array A

We now describe how a given list appears in memory. Each element e is stored along with the previous and next elements in the list, in fields we denote by e.prev and e.next. (We fix some label associated with the null list element.) The location of the triple $(e, e.\mathsf{prev}, e.\mathsf{next})$ is determined by hashing it into a hash table H built on top of array A. Of course, H must be a uniquely represented hash table, such as the one described in Section 3.1. Finally, first stores the first element in the list, and last stores the last element in the list.

Given this layout, it is fairly simple to describe the operations. The $\mathrm{first}()$ and $\mathrm{last}()$ operations merely output the contents of the corresponding fields. The $\mathrm{next}(x)$ and $\mathrm{previous}(x)$ operations perform a hash table lookup on x followed by a hash table lookup on x.next or x.prev, respectively. The $\mathrm{insert}(x, y)$ operation involves a looking up x in the hash table, finding $z := \mathrm{next}(x)$, inserting (y, x, z) into the hash table (recall the triples have the form $(e, e.\mathsf{prev}, e.\mathsf{next})$) and setting x.next to y and z.prev to y. The $\mathrm{delete}(x)$ operation is similar. Look up $w := \mathrm{previous}(x)$ and $y := \mathrm{next}(x)$ in the hash table, delete (x, w, y) from the hash table, set w.next to y and y.prev to w. The running times of this uniquely represented implementation are $O(1)$ worst-case time for $\mathrm{first}()$ and $\mathrm{last}()$, and $O(1)$ expected time for all of the other operations.

Reducing Space Usage with Element Labels

In the above implementation, we store each list element three times (except the first and last element, which are stored twice). Though this may seem wasteful, if the list elements take up no more space than pointers (e.g., if they are integers) then this scheme uses essentially the same amount of space as conventional

doubly-linked list implementations. If the list elements are significantly larger than pointers, we can reduce the space usage at the cost of allowing the linked list to fail with some tiny probability (e.g., $1/n^c$ for any user specified constant c, where there are n elements in the list) as follows. We generate *labels* for the list elements by hashing them into the set $\{0, 1, \ldots, L-1\}$ for some L to be determined. As long as all the elements in the list have distinct labels, we may hash on the labels rather than the list elements themselves, and store the label of x in place of x in the prev field of its successor in the list and in the next field of its predecessor in the list. The parameter L determines a tradeoff between the probability of failure (i.e., if two list elements get the same label) and the space requirements for the labels. Clearly, the space to store an integer in $\{0, 1, \ldots, L-1\}$ is $\lceil \log_2 L \rceil$ bits or $\lceil \log_2 L/w \rceil$ words where each machine word is w bits. Let X be the set of list elements being stored, so that $n = |X|$. If a pairwise independent hash function h is used to generate the labels, then via the union bound the probability that two elements get the same label can be bounded by

$$\textbf{Pr}[\text{collision}] \leq \sum_{x,x' \in X} \textbf{Pr}[h(x) = h(x')]$$

By the pairwise independence of h, for any distinct $x, x' \in X$,

$$\begin{aligned} \textbf{Pr}[h(x) = h(x')] &= \textstyle\sum_{k=0}^{L-1} \textbf{Pr}[h(x) = k \text{ and } h(x') = k] \\ &= \textstyle\sum_{k=0}^{L-1} 1/L^2 \\ &= 1/L \end{aligned}$$

Thus

$$\textbf{Pr}[\text{collision}] \leq \binom{|X|}{2} \frac{1}{L} \leq \frac{n(n-1)}{2L}.$$

Interestingly, the probability of a collision does not decrease much if an n-wise independent or truly random hash function is used to generate the labels. In this case, the probability that two elements get the same label is

$$1 - \prod_{k=0}^{n-1} \left(1 - \frac{k}{L}\right).$$

We can bound this quantity as follows.

$$\begin{aligned}
1 - \prod_{k=0}^{n-1}\left(1 - \tfrac{k}{L}\right) &\leq 1 - \prod_{k=0}^{n-1}\exp\left(-\tfrac{k}{L} - \tfrac{k^2}{L^2}\right) && \text{[assuming } L \geq 2n\text{]} \\
&= 1 - \exp\left(-\sum_{k=0}^{n-1}\tfrac{k}{L} - \sum_{k=0}^{n-1}\tfrac{k^2}{L^2}\right) \\
&= 1 - \exp\left(-\tfrac{n(n-1)}{2L} - \tfrac{(2n-1)(n-1)n}{6L^2}\right) \\
&\leq \tfrac{n(n-1)}{2L} + \tfrac{(2n-1)(n-1)n}{6L^2}
\end{aligned}$$

Above we have used the fact that $1 - x \geq e^{-x-x^2}$ for all $x \in [0, 1/2]$ on the first line, the facts that $\sum_{x=0}^{k} x = \frac{k(k+1)}{2}$ and $\sum_{x=0}^{k} x^2 = \frac{(2k+1)(k+1)k}{6}$ on the third line, and the fact that $e^{-x} \geq 1 - x$ for all $x \in \mathbb{R}$ on the fourth line.

The failure probability is thus $O(\frac{n^2}{L})$ in both cases. In fact it is straightforward to prove that the probability is $\Theta(\frac{n^2}{L})$ if $L \geq 2n$ and a truly random hash function is used. Thus using a truly random hash function only reduces the collision probability by at most a constant factor. However, in either case the failure probability decreases exponentially in the number of bits used to store a label.

4.4 Queues

The *queue* abstract data type stores a list of objects and supports the following operations.

- $\mathrm{enqueue}(x)$: append x to the end of the list.
- $\mathrm{dequeue}()$: remove the first object in the list and return it.

A queue can easily be implemented using a doubly-linked list, so conventional implementations exists supporting the above operations in worst-case $O(1)$ time. Using the data structure of Section 4.3, we can support these operations in expected $O(1)$ time if the assumptions for the linked list hold. That is, we require the queue be *bounded*, meaning that it is initialized with a capacity N specifying the maximum length the queue's underlying list is allowed to reach. Furthermore, we require that the objects in the queue can be stored in a constant number of words. We show how to remove these assumptions in Chapter 8.

4.5 Binary Search Trees via Treaps

There is no single definitive binary search tree abstract data type. Instead, there is a proliferation of binary search tree data structures providing various running time guarantees for various operations. Some better-known examples include AVL trees [AVL62], randomized treaps (also known as randomized search trees) [SA96], red-black trees [Bay72, GS78], and splay trees [ST85]. In the most abstract setting, we may consider binary search trees as storing a set of elements from an ordered universe (called *keys*), where each element may have some auxiliary data stored with it. Formally, the abstract data type states may be modeled as partial functions from an ordered universe to auxiliary data items. We call the element along with its auxiliary data a *node* of the tree. Of course, we would like to support the broadest set of operations possible. Table 4.2 lists the operations that our uniquely represented binary search tree will support.

Name	**Description**
$\text{lookup}(x)$	Given key x, find the tree node containing x.
$\text{insert}(x, d)$	If a node with key x already exists, set its auxiliary data to d. Otherwise insert key x with auxiliary data d.
$\text{delete}(x)$	Delete the tree node containing x, if it exists.
$\text{join}(T_1, T_2)$	Given two binary search trees T_1 and T_2 such that $x_1 < x_2$ for all keys $x_1 \in T_1$ and $x_2 \in T_2$, join T_1 and T_2 into a single tree $\{(x, d) : (x, d) \in T_1 \text{ or } (x, d) \in T_2\}$.
$\text{split}(T, x)$	Given binary search tree T and key x, split T into two trees $T_1 := \{(y, d) : (y, d) \in T, y \leq x\}$ and $T_2 := \{(y, d) : (y, d) \in T, y > x\}$.

Table 4.2: The binary search tree operations that our uniquely represented implementation will support.

Let n denote the number of keys in the tree. A modified red-black tree can support lookup, insert, delete, and split in $O(\log n)$ worst-case time, and join operations in amortized $O(\log n)$ time [Tar83]. We present a uniquely represented implementation that supports all of the above operations in expected $O(\log n)$ time, using randomized treaps [SA96]. As in previous sections, we defer the treatment of dynamic memory allocation to Chapter 8. Thus in this section we restrict our attention to bounded treaps which may store at most N keys (where N is provided as a parameter on initialization), and assume each key (along with its auxiliary data) can be stored in a constant number of machine words.

Recall the definition of a treap.

Definition 8 (Treap)**.** *A* treap *is a binary search tree in which each node has both a* key *and a* priority*. The nodes appear in-order by their keys (as in a standard binary search tree) and are heap-ordered by their priorities, so that the each parent has a higher priority than its children.*

Figure 4.1 shows two depictions of a treap. Treaps were first described by Vuillemin [Vui80], who called them *Cartesian trees*. The term "treap" comes from McCreight [McC85], who used it for a different data structure. McCreight later changed the name of these data structures to *priority search trees*. The current usage of the term is due to Seidel and Aragon [SA96], who analyzed randomized treaps in depth and proved many useful facts about them.

We use treaps in large part because there is a unique treap for each fixed set of keys and priorities, when treaps are interpreted as rooted, node-labeled binary trees. This is not the case with AVL trees, red-black trees, or splay trees. For example, Figure 4.2 on the facing page illustrates two red-black trees with the same set of keys. The suitability of treaps for constructing uniquely represented trees (for a pointer machine) has been noted before; Seidel and Aragon report that Bob Tarjan suggested the idea to them. Our contribution here is to implement uniquely represented binary search trees in a RAM.

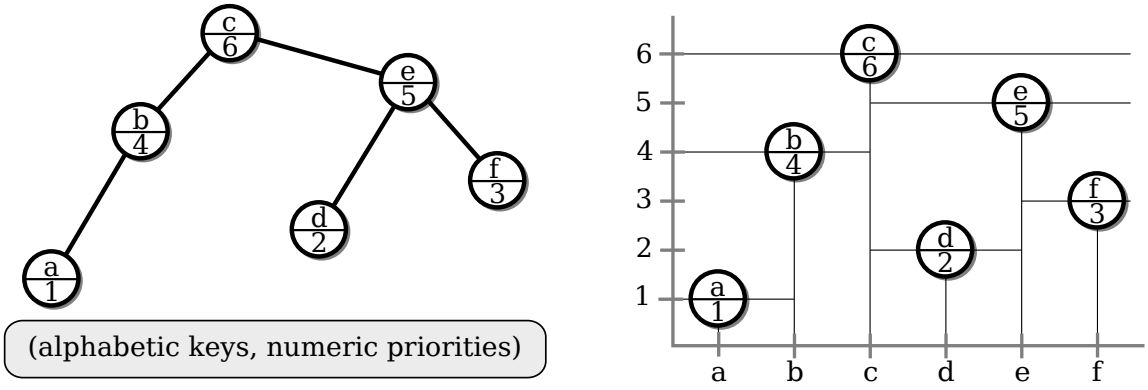


Figure 4.1: Two representations of a treap, one standard and one geometric.

Claim 1. *Given a ordered universe of keys U, an ordered universe of priorities V, and a mapping $p : U \to V$, there is a unique treap for each finite set of keys $S \subset U$ with relative priorities determined by a total order $<_p$ given by p with ties broken by the key order.*

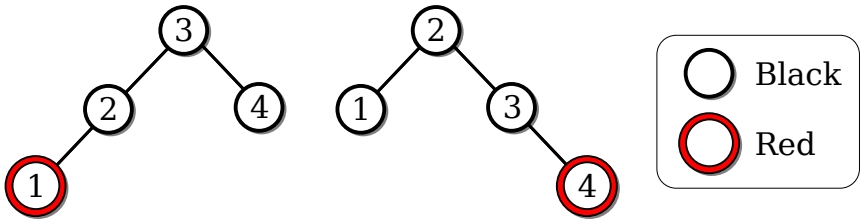


Figure 4.2: Two red-black trees with the same set of keys.

Proof: Formally, the total ordering $<_p$ on S is defined via $x <_p y$ if $p(x) < p(y)$ or if $p(x) = p(y)$ and $x < y$. We use $<_p$ to compare priorities. We now proceed by induction on $|S|$. The base case $|S| \leq 1$ is trivial. For the induction step, find the maximum element $e \in S$ with respect to $<_p$ and make it the root. By the heap invariant e must be the root of any treap on S, and the keys in its left and right subtree must be, respectively, $\{x : x < e\}$ and $\{x : x > e\}$. By the inductive hypothesis there are unique treaps T_1 and T_2 for these sets, so the left and right subtrees of e must be T_1 and T_2. Moreover, by construction there can be no violations of the heap constraint or the key in-order constraint involving e, nor can there be any constraint violations involving one node from T_1 and one node from T_2. Since any treap constraint violation has a witness consisting of two nodes, it is straightforward to show by induction that the resulting tree is indeed a treap. ■

Claim 1, together with the performance guarantees of randomized treaps, allow us to reduce the problem of constructing a uniquely represented binary search tree to the problem of providing sufficiently independent priorities to the keys in a principled way and dynamically maintaining a canonical mapping from keys to memory slots.

To generate priorities, we sample a hash function p from a suitable hash family from keys to $\mathbb{N}$. The priority of a key e is then $p(e)$. (If $p(e) = p(e')$, use the key ordering to resolve priorities, so that the larger key has higher priority.) This ensures that there is a total order on the priorities of the keys that does not change as keys are inserted and deleted. The idea of generating priorities via a hash function is not new; it appears in [SA96], where Seidel and Aragon attribute it to Danny Sleator. Given this total order on priorities, the treap implementations of the operations in Table 4.2 remain essentially the same, and we use a uniquely represented hash table to map tree nodes to memory locations.

Let us briefly review the treap operations. An essential primitive operation in

the treap is *rotation about an edge*, of the standard variety common to many binary search trees. Figure 4.3 depicts a rotation about edge $\{x, y\}$.

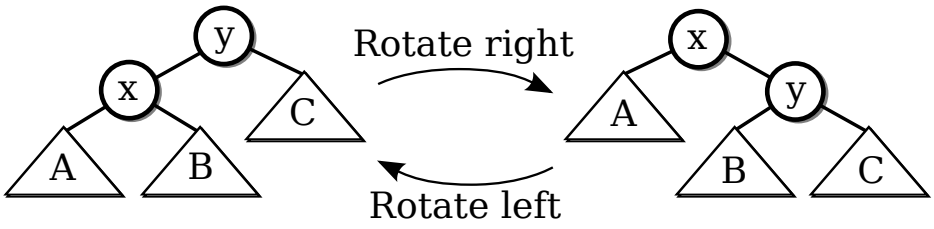


Figure 4.3: A binary search tree rotation about edge $\{x, y\}$.

Lookup: Searching for a key x proceeds as usual. That is, start at the root, and if the current node is v, proceed to v's left child if $x < v$, proceed to v's right child if $x > v$, return v if $x = v$, and return null if v is null.

Insertion: To insert a key x that is not already present, search for it until reaching the leaf position it would have occupied if it were in the treap and had the minimum priority of all keys in the treap. Insert x at that leaf position, and then repeatedly rotate on the edge between x and its parent until its parent has higher priority than it.

Deletion: To delete a key x, search for it, then repeatedly rotate on the edge between x and its higher priority child until x is a leaf, and then delete it from the tree.

Join: To join two treaps T_1 and T_2 where all keys in T_1 precede all keys in T_2, construct a new node v such that $x < v < y$ for all nodes $x \in T_1$ and $y \in T_2$, and set v to the root of the treap with left subtree T_1 and right subtree T_2. Finally, delete v from the treap as described above.

Split: To split a treap T along x, first insert x into T if it is not already in T, then repeatedly rotate on the edge between x and its parent until x is the root. Let T_1 be the left subtree of x and T_2 be the right subtree. If x was not originally in T, then return T_1 and T_2. Otherwise, insert x into T_1 and return T_1 and T_2.

Treaps have many desirable properties when priorities are randomly generated. For ease of reference, we list some these properties proved by Seidel and Aragon in Theorem 6.

Theorem 6 (Selected Treap Properties [SA96])**.** *Let T be a random treap on n nodes with priorities generated by an 8-wise independent hash function from nodes to $[p]$, where $p \geq n^3$. Then for any $x \in T$,*

(1) $\mathbf{E}[\text{depth}(x)] \leq 2\ln(n) + 1$*, so access and update times are expected $O(\log n)$.*

(2) Given a predecessor handle, the expected insertion or deletion time is $O(1)$.

(3) If the time to rotate a subtree of size k is $f(k)$ for some $f : \mathbb{N} \to \mathbb{R}_{\geq 1}$ such that $f(k)$ is polynomially bounded, then the total time due to rotations to insert or delete an element is $O\left(\frac{f(n)}{n} + \sum_{0<k<n} \frac{f(k)}{k^2}\right)$ in expectation. Thus even if the cost to rotate a subtree is linear in its size (e.g., $f(k) = \Theta(k)$), updates take expected $O(\log n)$ time.

To store the treap, we first construct nodes, each of which has fields for a key, a left and right child key, and a parent key. We denote these fields key, left, right, and parent. Given a node v, we use the field $v.\text{key}$ to hash v into a hash table of capacity $(1+\epsilon)N$ for some $\epsilon > 0$. By analogy with our linked list construction in Section 4.3, the left, right, and parent fields store the keys of v's left child, right child, and parent, respectively. These are used in lieu of pointers.

The main reason for avoiding pointers is that in a uniquely represented implementation, unlike its conventional counterpart, an object may need to be moved repeatedly – perhaps even every operation depending on the sequence of operations. If there are many pointers into that object, they will all need to be updated every time it moves. Thus instead of pointers we seek *labels* for our objects that are unique, immutable, and allow us to locate the objects in memory quickly. The simplest labeling scheme is use the object itself as its own label, and hash on these labels to determine where in memory the object is stored.

In addition to the hash table storing the treap nodes, we keep an extra copy of the root in a special root field. Using a uniquely represented hash table from Chapter 3, each pointer dereference is replaced by a hash table lookup that takes expected constant time. Moreover, treaps support the operations in Table 4.2 in expected $O(\log n)$ time even if an 8-wise independent hash function from keys to $[N^3]$ is used to generate the priorities. If we use different random bits for the hash table and the treap priorities, we obtain the following result.

Theorem 7. *There exists a uniquely represented implementation of a bounded treaps whose memory is statically allocated and stores elements of a fixed size, such that*

all of the operations in Table 4.2 take expected $O(\log n)$ time. Furthermore, this implementation requires only $O(\log N)$ random bits, and requires only $O(N)$ space to store a treap of capacity N.

To formally prove Theorem 7, we need to show that if there are a random number X of treap pointer dereferences (i.e., accesses to fields $v.\mathsf{left}, v.\mathsf{left}$ and $v.\mathsf{parent}$), each of which generates a hash table operation that takes expected $O(1)$ time, then the total expected time will be $O(\mathbf{E}[X])$. We prove this below. We will end up using the following lemma many times in running time analyses in this dissertation.

Lemma 1. *Let X be a random variable taking values in $\mathbb{N}$, and let $\{Y_i : i \geq 1\}$ be a set of random variables such that $\mathbf{E}[Y_i] \leq \mu$ for all i. Furthermore, suppose X has finite expectation and is independent of Y_i for all i. Then*

$$\mathbf{E}\left[\sum_{i=1}^{X} Y_i\right] \leq \mathbf{E}[X] \cdot \mu.$$

Proof: Define indicator random variables X_i such that $X_i = 1$ if $X \geq i$, and $X_i = 0$ otherwise. Clearly, X_i is independent of Y_i. Also note that $X = \sum_{i=1}^{\infty} X_i$. By linearity of expectation and the fact that $\mathbf{E}[AB] = \mathbf{E}[A] \cdot \mathbf{E}[B]$ for independent random variables A and B, we infer

$$\begin{aligned}
\mathbf{E}\left[\sum_{i=1}^{X} Y_i\right] &= \mathbf{E}\left[\sum_{i=0}^{\infty} X_i \cdot Y_i\right] \\
&= \sum_{i=1}^{\infty} \mathbf{E}[X_i] \cdot \mathbf{E}[Y_i] \\
&\leq \mu \sum_{i=1}^{\infty} \mathbf{E}[X_i] \\
&= \mu \mathbf{E}\left[\sum_{i=1}^{\infty} X_i\right] \\
&= \mu \mathbf{E}[X].
\end{aligned}$$

■

Lemma 1 allows us to "multiply expectations" when bounding the running time of our data structure. This means that we can support many other operations that randomized treaps support in the same running time, up to constant factors. In Table 4.3 on the next page we present some additional operations that treaps are known to support [SA96], along with their running times.

4.6 Heaps via Treaps

A *heap* is a tree on keys from an ordered set that satisfies the *heap constraint*, namely that each node is less than its parent. (This variant is known as a *max-heap*. Another variant where each node is greater than its parent is known as a *min-heap*. For the rest of this section we will consider the implementation of max-heaps. The implementation of min-heaps is analogous.) At its most abstract, the (max-)heap abstract data type stores an ordered set and supports the operations listed in Table 4.4 on the following page.

There are several heap implementations, for example binary heaps [Wil64], binomial heaps [Vui78], and Fibonacci heaps [FT87]. Table 4.5 lists their performance guarantees, together with those of our uniquely represented implementation. As in previous sections, we restrict out attention to statically allocated heaps, and defer the treatment of dynamic memory allocation to Chapter 8. Thus in this section we restrict our attention to bounded heaps which may store at most N keys (where N is provided as a parameter on initialization), and assume each key can be stored in a constant number of machine words.

We will prove the following result.

Theorem 8. *There exists a uniquely represented implementation of a bounded heap whose memory is statically allocated and stores elements of a fixed size, such that all of the operations in Table 4.4 are supported with the running time guarantees indicated in Table 4.5. Furthermore, this implementation requires only $O(\log N)$ random bits, and requires only $O(N)$ space to store a heap of capacity N.*

Table 4.5 reveals some tradeoffs between the running times for the uniquely represented implementation versus Fibonacci heaps. The Fibonacci heap is faster for insert and $\mathrm{increase\text{-}key}$, but slower for $\mathrm{delete\text{-}max}$ and delete. This is because our uniquely represented implementation stores the keys in sorted order, whereas Fibonacci heaps do not. Overall the performance of our uniquely represented heap

Name	Description	Expected Running Time
finger-insert(x, d, y)	Given a pointer to the predecessor or successor y of the new key x, insert key x with auxiliary data d.	$O(1)$
finger-delete(x)	Given a pointer to key x, delete it.	$O(1)$
finger-search(x, y)	Given a pointer to key x, find y, where there are d keys between x and y	$O(\log d)$
Costly-updates	An insertion or deletion where we perform $O(s)$ work when rotating subtrees of size s	$O(\log n)$
Costly-updates	An insertion or deletion where we perform $f(s)$ work when rotating subtrees of size s, with f non-negative.	$O(\frac{f(n)}{n} + \sum_{k=1}^{n-1} \frac{f(k)}{k^2})$

Table 4.3: Some additional treap operations our uniquely represented implementation will support. See [SA96] for more detail.

Name	Description
build-heap(A)	Given a (possibly unsorted) array A with n keys, construct a heap on those keys.
insert(x)	Insert key x into the heap.
delete-max()	Delete the maximum key from the heap and return it.
find-max()	Return the maximum key in the heap.
increase-key(x, Δ)	Given a pointer to x in the heap, increase its key by Δ
delete(x)	Given a pointer to x in the heap, delete it.
merge(H_1, H_2) (also called meld)	Given two heaps H_1 and H_2, merge them into a single heap with all the keys from both H_1 and H_2.

Table 4.4: The heap operations.

	Binary	**Binomial**	**Fibonacci**	**Uniquely Represented**
build-heap(A)	$O(n)$	$O(n)^{\ddagger}$	$O(n)^{\ddagger}$	$O(n \log n)^{\dagger}$
insert(x)	$O(\log n)$	$O(1)^{\ddagger}$	$O(1)^{\ddagger}$	$O(\log n)^{\dagger}$
delete-max()	$O(\log n)$	$O(\log n)^{\ddagger}$	$O(\log n)^{\ddagger}$	$O(1)^{\dagger}$
find-max()	$O(1)$	$O(1)$	$O(1)$	$O(1)$
increase-key(x, Δ)	$O(\log n)$	$O(\log n)^{\ddagger}$	$O(1)^{\ddagger}$	$O(\log n)^{\dagger}$
delete(x)	$O(\log n)$	$O(\log n)^{\ddagger}$	$O(\log n)^{\ddagger}$	$O(1)^{\dagger}$
merge(H_1, H_2)	$O(n)$	$O(1)^{\ddagger}$	$O(1)^{\ddagger}$	$O(n)^{\dagger}$
† indicates the running time guarantee is in expectation. ‡ indicates the running time guarantee is amortized.				

Table 4.5: Running times for various heap implementations with n keys. Running times for binary heaps are taken from [CLRS01], while those for binomial and Fibonacci heaps are either taken from [Koz91] or may be inferred from the implementation described therein.

implementation compares well with conventional implementations, with the exception of the merge operation. Unfortunately, we will show in Chapter 7 by a very general argument that every uniquely represented heap implementation using polynomial space must take $\Omega(\frac{n}{\log n})$ expected time to merge two heaps. See Section 7.3 for details.

Implementation via Treaps. We use the treap of Section 4.5 to implement heaps. Given a set of keys from an ordered universe $(U, <)$, we simply store them in uniquely represented treap. The binary relation $<$ on keys serves as the binary search tree order within the treap, and priorities are generated by hashing the keys into a sufficiently large integer range. Thus, the root of the treap is the key with the maximum priority, as opposed to the maximum element with respect to $<$. The operations are then implemented as follows.

Build-Heap: Build heap is implemented in the naïve way. Simply run through the array inserting each element in turn into the treap. Since each insertion takes expected $O(\log n)$ time, the total time for this operation is $O(n \log n)$. Note that, once the treap is built, we may output the elements in sorted order in expected $O(n)$ time using the fast finger-search supported by the treap. Since the input array for build-heap is initially unsorted, this fact combined with the $\Omega(n \log n)$ lower bound for sorting in a comparison-based model of

computation implies that there is no asymptotically faster way to build the treap.

Insertion: Simply use the treap insertion method, which takes expected $O(\log n)$ time.

Find-Max: Our implementation will store a copy of the maximum key in a special field called max_key. To implement find-max we simply return its contents in worst case constant time.

Delete-Max: Find the maximum key x via a find-max operation as described above. Hash the key to find the treap node v containing x. Then use fast finger search to find the successor y of x in the treap in expected $O(1)$ time. Finally, set max_key to y and use fast finger delete to delete x from the treap in expected $O(1)$ time.

Increase-Key: To increase the key of x by Δ, construct a copy x' of x with its key increased by Δ. Do a fast finger search for the predecessor y of x' in the treap. (This can be done by searching for x' itself. See [SA96] for details.) Then use y to do a fast finger insertion of x'. Finally, do a fast finger delete of x. Inserting x' given y and deleting x both take expected $O(1)$ time. Searching for y takes expected $O(\log d)$ time, where d is the *distance* between x and y (i.e., the number of keys z such that $x < z \leq y$). Clearly $d \leq n$ so this operation runs in expected $O(\log n)$ time.

Delete: If the key to be deleted is not the maximum key, simply use fast finger deletion to delete the relevant key from the treap in expected $O(1)$ time. Otherwise, call delete-max.

Merge: Let H_1 and H_2 be the heaps to be merged. Because we wish to defer the treatment of dynamic memory allocation to Chapter 8, we will assume that H_1 and H_2 are stored in separate memory blocks, and that the we have allocated a third memory block to store the result of the merge operation, which we denote by H_3. In this section we focus on ensuring that H_3 is uniquely represented within its block, without concern to where within the RAM memory that block resides. First, initialize H_3 to be a uniquely represented heap with the desired capacity. This might be the sum of the capacities of H_1 or H_2, or we could add a parameter to the merge operation allowing the user to specify it. We can easily set H_3.max_key to be the maximum of H_1.max_key and H_2.max_key. The remainder of the merge operation is given in pseudocode

in Figure 4.6. In the figure T_i is the treap storing the keys of heap H_i and $\text{pred}(x, H_i) = \max\{y : y \in H_i,\ y < x\}$ is the predecessor of x in the treap T_i. If the predecessor does not exist $\text{pred}(x, H_i)$ returns null. Additionally, null is less than all keys for purposes of comparison.

```
merge-heaps(heap H_1, heap H_2, heap H_3)
    Set x_1 and x_2 to be the maximum keys in H_1 and H_2, respectively.
    Set i = arg-max{x_j : j ∈ {1,2}}.
    Set y = x_i.
    Insert y into the treap T_3 storing H_3.
    Set x_i = pred(x_i, H_i).
    While (x_1 ≠ null or x_2 ≠ null){
        Set i = arg-max{x_j : j ∈ {1,2}}.
        Insert x_i into T_3 using fast finger insertion with y.
        Set y = x_i.
        Set x_i = pred(x_i, H_i).
    }
```

Figure 4.4: Pseudocode for merging two uniquely represented heaps.

The pseudocode in Figure 4.6 runs in expected $O(n)$ time, where there are n keys in the union of H_1 and H_2. Given a binary search tree, let the *reverse-order traversal* be the traversal that visits the right subtree of a node v, then v, then finally the left subtree of v. Thus the reverse-order traversal is basically an in-order traversal in reverse, where we start with the maximum node and proceed towards the minimum node. The pseudocode essentially does a staggered reverse-order traversal of H_1 and H_2, and inserts the greater of the two currently visited nodes into H_3 in a manner strongly resembling the merging of two sorted arrays à la merge-sort. Thus, while each individual call to $\text{pred}(x_i, H_i)$ takes expected $O(1)$ time via fast finger search [SA96], all of these calls together traverse less than $2n$ treap edges and take expected $O(n)$ time. Also, each insertion into H_3 other than the first one is a fast finger insertion taking expected $O(1)$ time. (Note that even though y is a key and not a treap node, we can find the treap node in T_3 corresponding to y in expected constant time using the hash table underlying T_3). Thus, the insertions collectively take expected $O(n)$ time as well. The entire merge operation therefore takes expected $O(n)$ time.

To complete the proof of Theorem 8, note that the bounds on the number of random bits required and the space usage are immediate consequences of the properties of uniquely represented treaps stated in Theorem 7 on page 59.

Chapter 5

Advanced Data Structures and Techniques

In this chapter we continue our discussion of efficient uniquely represented implementations for various abstract data types, and cover more complex implementations than those encountered in Chapter 4. We additionally discuss various techniques that may be employed to construct uniquely represented data structures, in the hope that they may prove useful in expanding the range of abstract data types with efficient uniquely represented implementations in the future.

As in Chapter 4, we will simplify the exposition here by considering only statically allocated data structures. We defer the treatment of dynamically allocated data structures to Chapter 8. That is, we assume that a contiguous block of memory is allocated to the data structure in question on initialization, and which memory locations are allocated to the data structure does not change over time.

5.1 Skip Lists

Skip lists were developed by William Pugh [Pug90] as a simple randomized alternative to balanced binary search trees. They implement the binary search tree abstract data type described in Section 4.5 on page 55. Figure 5.1 depicts a skip list and Table 5.1 lists the operations that skip lists support, all of which take expected $O(\log n)$ time. We will provide a uniquely represented skip list implementation that supports the same guarantees.

Theorem 9. *There exists a uniquely represented implementation of a bounded skip list whose memory is statically allocated and stores elements of a fixed size, such that all of the operations in Table 5.1 take expected $O(\log n)$ time. Furthermore, with high probability this implementation requires only $O(N)$ space to store a skip list of capacity N.*

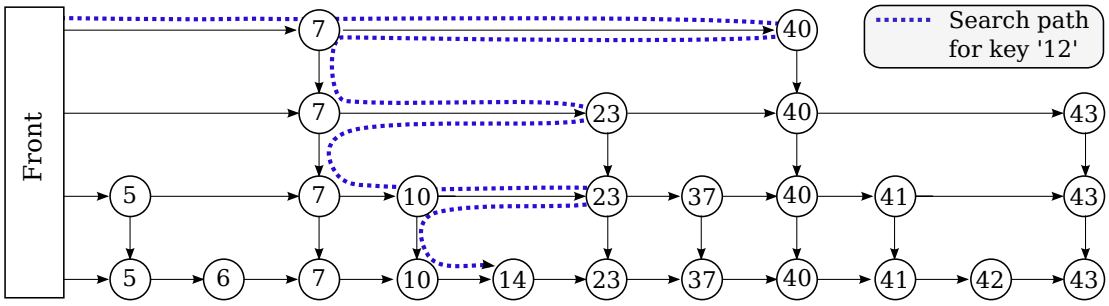


Figure 5.1: A skip list, with the search path for key 12 illustrated.

Name	Description
$\text{lookup}(x)$	Given key x, find the skip list node in L_1 containing x.
$\text{insert}(x)$	Insert key x into the skip list.
$\text{delete}(x)$	Delete x from the skip list, if it exists.
$\text{join}(L_a, L_b)$	Given two skip lists L_a and L_b such that $x_a < x_b$ for all keys $x_a \in L_a$ and $x_b \in L_b$, join L_a and L_b into a single skip list with all keys from both L_a and L_b.
$\text{split}(L, x)$	Given skip list L and key x, split L into two skip lists $L_{\leq} := \{y : y \in L, y \leq x\}$ and $L_{>} := \{y : y \in L, y > x\}$.

Table 5.1: The skip list operations that our uniquely represented implementation will support.

Let U be the set of elements we wish to store. The idea behind skip lists is assign a random positive integer *level* to each element $u \in U$, such that $\mathbf{Pr}[\text{level}(u) = k]$ decreases exponentially in k, and for each k such that some element has $\text{level}(u) \geq k$, maintain a list L_k on all elements u with $\text{level}(u) \geq k$. We also assume there is a special element front that is in every list and is smaller than all other elements, so as to be at the front of every list. Note that each list $L_k = \langle u_1, u_2, \ldots, u_t \rangle$ partitions

the elements into contiguous subsets

$$\{\{u : u_i \leq u < u_{i+1}\} : 1 \leq i < t\} \cup \{\{u : u_t \leq u\}\}$$

and the partition of L_{k-1} refines[1] the partition of L_k. Additionally, we maintain pointers from each element $u \in L_k$ to $u \in L_{k-1}$. Let u_i be the *representative* of $\{u : u_i \leq u < u_{i+1}\}$. When searching for an element u, we start at the highest level list L_k and search for the representative v_k of the level k partition set containing u, then starting from v_k in L_{k-1} search for the representative v_{k-1} of the level $k-1$ partition set containing u, and so on until reaching the representative $v_1 = u$ of the level one partition set containing u.

The original skip list paper [Pug90] sets element levels by repeatedly flipping a biased coin up to $(\beta - 1)$ times, for some parameter β which bounds the maximum allowable level. The level of the element is then set to one plus the number of coin flips to obtain an outcome of 'heads'. If all $(\beta - 1)$ coin flips turn up 'tails,' then the level is set to β. The distribution over levels is thus parameterized by the coin bias $p \in (0, 1)$ and a bound β on the maximum allowable level. Specifically, it is

$$\mathbf{Pr}[\text{level}(u) = k] = \begin{cases} p^{-(k-1)}(1-p) & \text{if } 1 \leq k < \beta \\ p^{-(\beta-1)} & \text{if } k = \beta \\ 0 & \text{if } k < 1 \text{ or } k > \beta \end{cases}$$

Using this distribution to sample the levels independently for each node, all the operations in Table 5.1 can be implemented in $O(\log n)$ time [Pug90, PMP90].

Inspecting Figure 5.1, it is fairly easy to see that the pointer structure of the skip list is uniquely represented if the levels are generated using a hash function. Pugh [Pug90] recognized this, and notes various advantages of having uniquely represented (with respect to the pointer structure) skip lists:

> "If we combine this idea [of unique representation] with an applicative (i.e., persistent) probabilistically balanced data structure and a scheme such as hashed–consing [All78] which allows constant-time structural equality tests of applicative data structures, we get a number of interesting properties, such as constant-time equality tests for the representations of sequences. This scheme also has a number of applications for incremental computation [PT89]. Since skip lists are somewhat awkward to make applicative, a probabilistically balanced tree scheme is used."

[1] A partition $\{A_1, \ldots, A_r\}$ of U is said to refine a partition $\{B_1, \ldots, B_r\}$ of U if for all A_i there is a B_j such that $A_i \subseteq B_j$.

Using n-wise independent hash functions, such as those of Östlin and Pagh [OP03] or Dietzfelbinger and Woelfel [DW03], we can obtain skip lists that are uniquely represented with respect to their pointer structure, and have the same running time guarantees as those in [Pug90]. Given a hash function h from keys to $[1 : r]$ for some parameter $r \geq n$, we set the level of u to be the largest value of k such that $r - h(u) \leq r \cdot 2^{-(k-1)}$. This implementation of skip lists is similar in many respects to the treaps of Section 4.5 on page 55. Unlike the case with the treaps, however, the skip list has several nodes with exactly the same key. This complicates the task of mapping the nodes of the skip list into the RAM memory. For skip lists there is a fairly simple solution — we can label the copy of v in list L_k with the numeric value of the string $v \oplus k$, where $\oplus$ is the concatenation operator. For a more general way of generating hashable labels, see Section 5.2.

With these observations, we are now ready to prove Theorem 9. Labeling the nodes using the numeric value of $v \oplus k$ results in distinct labels. Using a uniquely represented hash table such as the one from Section 3.1 on page 27 implies that each constant time operation in the normal skip list operations (e.g., dereferencing or updating a pointer, creating or deleting a node) corresponds to an expected constant time operation for our uniquely represented implementation (e.g., hash table lookup, changing a key, inserting or deleting a node from the hash table). Since we use independent random bits for the hash tables and the skip list node levels, we can use Lemma 1 on page 60 to show that our running times are expected $O(f(n))$ whenever they are $f(n)$ for skip lists. Using the running time guarantees for skip lists provided by Pugh [Pug90], we conclude that our implementation supports lookups, insertions, deletions, splits, and joins is expected $O(\log n)$ time, where there are n keys. As for the space, it is easy to prove that in expectation there are at most 2 copies of any given node, using the level-generating scheme described above. Thus there are at most $2n$ nodes in expectation, each occupying a constant number of words of space. Since the hash table from Section 3.1 on page 27 requires only $(1 + \epsilon)N$ slots to store N elements, this implies we need only $O(N)$ words of space in expectation to store up to N skip list elements. Moreover, if the levels are generated by an n-wise independent hash function, we can prove strong measure-concentration results for the total number of nodes. For example, a straightforward application of Azuma-Hoeffding inequality (see [AS00] or [MR95] for a treatment) proves that with high probability the skip list has a total of $2n + O(n^{1/2}(\log n)^{3/2})$ nodes.

5.2 Generating Hashable Labels: The Hash–Consing Technique

In the previous section we encountered skip lists, which may have multiple nodes with the same key. To allocate these nodes to memory locations, we need to provide them with distinct labels in a strongly history-independent manner. In the case of skip lists, there is a simple way to do this. However, in later sections we will have need of a general, principled way of generating hashable labels. Our solution to the more general labeling problem is based on the technique of *hash–consing* [Got74, All78].

5.2.1 Labels via Subtrees

Before discussing the use of hash–consing to generate labels, we will describe the related idea of generating labels via subtrees. Suppose we have a binary tree $T = (V, E)$ rooted at $r \in V$, where each node v has an associated value $x_v \in \mathbb{N}$, and we wish to assign a unique integer labels to each node in V. Furthermore, suppose we wish the labeling scheme to be strongly history independent. To simplify the exposition, we may assume that the children of each node v are ordered in T. Such trees are *serializable*, in the sense that there is a unique string representation for each such tree using the grammar

$$\text{tree} ::= \text{empty} \,|\, \text{node}(x, \langle \text{tree}, \text{tree} \rangle)$$

where $x \in \mathbb{N}$. For example,

$$\text{node}(2, \langle \text{node}(1, \langle \text{empty}, \text{empty} \rangle), \text{node}(3, \langle \text{empty}, \text{empty} \rangle) \rangle)$$

is the three node tree with a root r that has two children a and b, and $x_r = 2, x_a = 1$, and $x_b = 3$. Assuming we can convert such strings to integers, this suggests the following labeling scheme: *label v with the integer corresponding to the string representation of T_v*, where T_v is the subtree of T rooted at v.

This labeling scheme has the property that u and v get the same label if and only if the string representations of T_u and T_v are equal, which only occurs if T_u and T_v are isomorphic as node labeled rooted trees. Under a functional programming paradigm where the subtrees are immutable, it will be acceptable to consider u and v to be the same node, and store them accordingly. In other words, we may

define an equivalence relation $\approx$ such that $u \approx v$ if T_u is isomorphic to T_v. Formally, $u \approx v$ if

- Both u and v are leaves and $x_u = x_v$, or
- Neither u nor v is a leaf, $x_u = x_v$, and for all i, $u_i \approx v_i$ where u_i is the i^{th} child of u and v_i is the i^{th} child of v.

We then label the nodes such that u and v get the same labels if and only if $u \approx v$.

5.2.2 The Hash–Consing Technique

One significant disadvantage of the above labeling scheme is that the labels themselves are huge. Note that we only generate at most n labels, and the space of possible node-labeled subtrees on up to n nodes is rather large. This suggests that hashing the subtrees into the integer range $[r]$ for some suitable $r = \text{poly}(n)$, to obtain compact representations of the trees. The *hash–consing* technique does this in a particular way, and labels a node v by first labeling its children and then hashing on the labels of its children and its associated value to obtain its label. It requires a hash function h that takes as input a list of values. See Figure 5.2.

```
hash-cons-label(node u)
  If (u is a leaf)
    Return h(x_u);
  Else
    For each child u_i of u
      Set l_i to hash-cons-label(u_i);
    Let t be the number of u's children.
    Return h(l_1, l_2, ..., l_t; x_u);
```

Figure 5.2: Psuedocode for a hash-consing based labeling scheme.

Hash-consing was originally devised as a space saving mechanism. Ershov [Ers58] showed how to quotient trees by the equivalence relation $\approx$ described above by traversing the tree from the leaves up. By collapsing the equivalence classes into single nodes, the tree can be represented by a directed acyclic graph with potentially many fewer nodes. Goto [Got74] exploited this fact by implementing a memory allocator for the LISP programming language that labeled all cons-cells and avoided creating duplicates by testing whether a newly created cons-cell had the same label as a previously generated one. Hash–consing has also been imple-

mented for the Standard ML of New Jersey [AG93] and Objective Caml [FC06] programming languages. Related ideas have also been used to efficiently store certain infinite trees [Mau00].

An alternate way to label the nodes of the tree would be to use a *trie*, and label a node v with the root to v path. This approach has some disadvantages relative to hash-consing.

1. In a tree T, rotating about edge $\{u, v\}$ will alter the trie derived labels of every node in the subtrees T_u and T_v, but will only alter the hash–consing derived labels of the ancestors of u and v (including u and v themselves).

2. Trie derived labels, unlike hash–consing derived labels, do not expose isomorphisms in the underlying subtrees that allow us to save space.

3. Hash–consing effortlessly extends to the case of directed acyclic graphs, where there are multiple "roots" (i.e., zero indegree nodes) and possibly multiple paths with the same endpoints. Simply use the out-neighbors of u, $\Gamma_{\text{out}}(u) := \{v : (u, v) \in E\}$, in lieu of children, and use nodes with out-degree zero in lieu of leaves. This yields the following definition of $\approx$ for directed acyclic graphs. Nodes u and v satisfy $u \approx v$ if

   - Both u and v have zero out-degree and $x_u = x_v$, or
   - Neither u nor v has zero out-degree, $x_u = x_v$, and for all i, $u_i \approx v_i$ where u_i is the i^{th} element of $\Gamma_{\text{out}}(u)$ and v_i is the i^{th} child of $\Gamma_{\text{out}}(v)$.

   Labeling schemes based on tries do not easily extend to directed acyclic graphs.

We can also relax the assumption that the out-edges of each node u are ordered. Simply find the labels of all nodes in $\Gamma_{\text{out}}(u) := \{v : (u, v) \in E\}$, and order them by their labels.

Dynamic Updates. The update rules for hash–consing derived labels are quite simple. If a node v is altered, such that its children or its associated value change, then every node u such that v is reachable from u must have its label recomputed. In a rooted tree (without parent pointers), this means that whenever a node v is altered, all of its ancestors must have their labels recomputed. Unfortunately, this can have some implications for performance. For example, in the context of uniquely represented skip lists and uniquely represented treaps, this labeling

scheme rules out expected constant time finger insertions and finger deletions but still allows for expected $O(\log n)$ time operations. So, how can we efficiently relabel all the ancestors of an altered node v in a tree without parent pointers? One way is to traverse and temporarily store the root to v path P during any operation. This allows us to update the labels of all nodes along P in expected $O(|P|)$ time. See Figure 5.3 for a recursive procedure that does this when given the root and P as input. A simple iterative version of this procedure also exists.

```
hash-cons-label(node v, path P)
    Let t be the number of v's children, and let l_j be the label of v's j^th child.
    If (v has a child v_i ∈ P)
        Set l_i to hash-cons-label(v_i, P).
        Return h(l_1, l_2, ..., l_t; x_v)
    Else v has no children in P
        Return h(l_1, l_2, ..., l_t; x_v)
```

Figure 5.3: Psuedocode for a updating the hash-consing based labels in a tree, given a path P from the root to the altered node.

In certain special cases we may retain the parent points and still use hash–consing to generate labels. This is safe so long as whenever $u \approx v$, $u.\mathsf{parent} = v.\mathsf{parent}$. It is fairly easy to see that this is the case with skip lists. In fact, in a skip list no two nodes u and v satisfy $u \approx v$. Treaps have this property as well, since no two treap nodes have the same associated value.

Example: Skip Lists. With these observations, we are ready to show how to use hash–consing to generate labels for skip list nodes such that Theorem 9 remains true (omitting the treatment of hash collisions resulting in non-distinct labels, which we discuss below). Simply use the standard skip list operations from [Pug90] (and the obvious implementations of split and join), together with the hash–consing labeling scheme described above and a uniquely represented hash table such as the one from Section 3.1 on page 27. For the purposes of computing the labels, we treat the skip list L as a binary tree $T(L)$ as follows[2]. Let (v, k) denote the copy of v in the list L_k of nodes at level k or higher. Then for each (v, k), if $(v, k+1)$ exists then let $(v, k+1)$ be the parent of (v, k), and otherwise let the predecessor of v in L_k be the parent of (v, k). If we use a hash function that can be evaluated in constant

[2]This binary tree is unrelated to the so-called *skip tree* data structure [Mes97].

time, we can amortize the time to update the labels against the search time for the input key for all operations except join. For joins, we can amortize the time to update the labels against the length of the search path for a key that is strictly between the keys in the input skip lists, which is $O(\log n)$ in expectation. Since the hash table operations take expected constant time, and we use independent random bits for the hash table and for the skip list levels, we can use Lemma 1 on page 60 to bound the total time for lookups, insertions, deletions, splits, and joins at expected $O(\log n)$ time, where there are n keys.

Detecting Hash Collisions. Using a hash of T_v rather than T_v to label v creates the possibility of a *hash collision*, such that two nodes u and v with $u \not\approx v$ receive the same label. Since we are using the labels as identifiers, this creates the unacceptable situation of identifying two distinct nodes. Fortunately, we can detect such collisions as follows. Whenever u is assigned the same label as existing node v, check that their associated values are the same (i.e., $x_u = x_v$) and that the labels of out-neighbors $\Gamma_{\mathrm{out}}(u)$ and $\Gamma_{\mathrm{out}}(v)$ are equal as lists (i.e., for all i, if u_i and v_i are the i^{th} elements of $\Gamma_{\mathrm{out}}(u)$ and $\Gamma_{\mathrm{out}}(v)$ respectively, then u_i and v_i have the same label). If so, then $u \approx v$ and so u and v should receive the same label. Otherwise, if we assume no hash collision has occurred before, then $u \not\approx v$ and a hash collision has occurred. Thus we may detect the *first* hash collision in this manner with expected constant time overhead per operation, even if we cannot detect subsequent collisions.

Dealing with Hash Collisions. If a hash collision occurs, we must relabel at least some of the nodes. There are several possible ways to do this that still allow us to support expected $O(\log n)$ time operations, and we list some of them below.

1. Sample a fresh labeling hash function and relabel all nodes with it. Note that, as with uniquely represented dynamic perfect hash table, we cannot sample random bits "on demand" and must instead maintain a permutation π on a suitable hash family, and use the first hash function in π that is collision-free on the current set of keys. See Section 3.2 for details.
2. Give up on using a hash table for memory allocation. Instead, define a total ordering on the nodes, and store them in order in the memory array. One potential total ordering can be obtained via the lexicographic order on strings obtained from serializing the subtrees T_v for all nodes v, as in the full subtree labeling scheme of Section 5.2.1.

3. Store the set S of nodes involved in some collision (i.e., those nodes u such that there exists v such that $u \not\approx v$ and u and v receive the same label from the original hash function h), and relabel them in some principled fashion. One possible method is to maintain a part of the label space specifically for such nodes, say the interval $[a : b]$, and sort the nodes $v \in S$ according to a value y_v generated from the labels of the children of v and the value x_v associated with v. If v is the i^{th} node in S sorted by y values, then give v label $a + i$. Compute the labels of nodes $v \notin S$ as before, ignoring how the labels of $\Gamma_{\text{out}}(v)$ are generated.

Of these, perhaps the third is the most practical. Note that in all of these cases, if the tree is altered in such as way as the original hash function does not generate any label collisions, then by the unique representation property we must revert the data structure to using the original hash-consing derived labels. However, the probability of a collision can be made as small as $O(1/n^c)$ for any user-specified constant c, and we can accommodate these constraints while maintaining the expected $O(\log n)$ time guarantees on the operations. The details are similar to Section 3.2, and we omit them.

Bounding the Probability of Collisions. Let h be the hash function used to generate the labels, and suppose h maps keys to $[r]$, for some parameter $r \in \mathbb{N}$. We will prove the following.

Proposition 3. *If h is pairwise independent hash function from keys to $[r]$, then using h to generate labels for the nodes of a directed acyclic graph via the hash–consing scheme described above results in a label collision with probability $O(n^2/r)$, where there are n nodes in the graph.*

Proof: We claim that if there is a collision on labels, then there exist nodes $u \not\approx v$ that collide (i.e., are given the same label) such that u and v give different arguments to h to obtain their labels. To prove this, select any two nodes $u \not\approx v$ that collide. If they additionally give identical arguments to h to obtain their labels, then they must have the same number of out-neighbors (i.e., $|\Gamma_{\text{out}}(u)| = |\Gamma_{\text{out}}(u)|$), and they must have the same associated value (i.e., $x_u = x_v$). Thus, by definition of $\approx$, they must have out-neighbors u_i and v_i with such that $u_i \not\approx v$ yet u_i and v_i have the same label. If u_i and v_i give different arguments to h to obtain their labels, then we are done. Otherwise, just apply the same argument to u_i and v_i. Each time we apply this argument, we advance along a chain in the graph, and since the graph is

acyclic, the longest chain has length at most n. Thus this process must terminate at the desired colliding nodes that give different arguments to h to obtain their labels.

Let X be the set of arguments given to h to obtain all the labels for nodes in the graph. Since the graph has n nodes, $|X| \leq n$. Suppose h is at least pairwise independent. Then using the above claim and the analysis of Section 4.3 on page 51, the probability of a collision can be bounded by $O(n^2/r)$. ∎

5.3 The Treap Partitioning Technique

In the sections that follow, we will have need of a uniquely represented partitioning scheme that partitions a dynamic ordered set U into contiguous subsets of size at most β, where β is a user-defined parameter. Formally, the *ordered set partitioning* abstract data type, when initialized with some fixed value for β, should store a partition $\{U_i : i \geq 0\}$ of an ordered set U such that

- (Contiguous sets): For all $i \neq j$, either $\max\{u \in U_i\} < \min\{u \in U_j\}$ or $\max\{u \in U_j\} < \min\{u \in U_i\}$.
- (Size bound): For all i, $|U_i| \leq \beta$.

The ordered set partitioning abstract data type also supports the following operations.

- $\text{insert}(u)$: Insert u into the set U.
- $\text{finger-insert}(u, v)$: Given a pointer to the predecessor v of u in U, insert u into the set U.
- $\text{finger-delete}(u)$: Given a pointer to u, delete it from U.
- $\text{find}(u)$: Return a *representative* of U_i, namely an element $v \in U_i$. This operation must maintain the invariant that at all times, for all partition sets U_i and for all $u, u' \in U_i$, $\text{find}(u) = \text{find}(u')$.

Without loss of generality, we may also store U in a doubly linked list, and support expected constant time successor and predecessor queries.

A uniquely represented implementation of this abstract data type must satisfy the additional constraint that the representative of each partition set U_i must be a

function only of the current partition, and not of the historical sequence of operations that generated it. Our uniquely represented implementation, which we call *treap partitioning*, first appeared in [BG07] and works as follows.

5.3.1 The Construction

Given a treap T and an element $x \in T$, let its *weight* $w(x, T)$ be the number of its descendants, including itself. For a parameter s, let

$$\mathcal{L}_s[T] := \{x \in T : w(x, T) \geq s\} \cup \{\text{root}(T)\}$$

be the *weight s partition leaders* of T[3]. We will often refer to these nodes simply as *leaders*. For every $x \in T$ let $\ell(x, T)$ be the least (deepest) ancestor of x in T that is a partition leader. Here, each node is considered an ancestor of itself. We will call a node x a *follower* of its leader $\ell(x, T)$. The weight s partition leaders partition the treap into the sets $\{\{y \in T : \ell(y, T) = x\} : x \in \mathcal{L}_s[T]\}$, each of which is a contiguous block of keys from T consisting of the followers of some leader. It is not hard to see that each set in the partition has at most $2s - 1$ elements, since a leader can have at most $s - 1$ followers less than it and at most $s - 1$ followers greater than it in key order. To ensure each partition set has size at most β, we set $s = \lceil \frac{\beta}{2} \rceil$. The representative of the partition set $\{y \in T : \ell(y, T) = x\}$ is x. Figure 5.4 on the next page depicts the treap partitioning.

We implement treap partitioning by storing the set U in a treap, where each node v has a key field storing an element of U that induces an ordering on treap nodes in the obvious way: $u < v$ if and only if $u.\text{key} < v.\text{key}$. Also, the treap priority for a node u is generated by hashing $u.\text{key}$, and the treap nodes are hashed into memory using their keys. Each node v additionally has a leader field which stores the representative of the set it is in, namely $\ell(v, T).\text{key}$, the key of the deepest leader node that is an ancestor of v. Thus, the treap partitioning scheme yields the partition $\{\{y.\text{key} : y \in T, \ell(y, T) = x\} : x \in \mathcal{L}_s[T]\}$.

For a key x, we let $\text{node}(x)$ denote the treap node containing x, if it exists. We will prove the following.

Theorem 10. *There exists a uniquely represented implementation of the ordered set partitioning abstract data type, whose memory is statically allocated and stores elements of a fixed size, such that insertions take expected $O(\log n)$ time, and fast finger*

[3]For technical reasons we include $\text{root}(T)$ in $\mathcal{L}_s[T]$ ensuring that $\mathcal{L}_s[T]$ is nonempty.

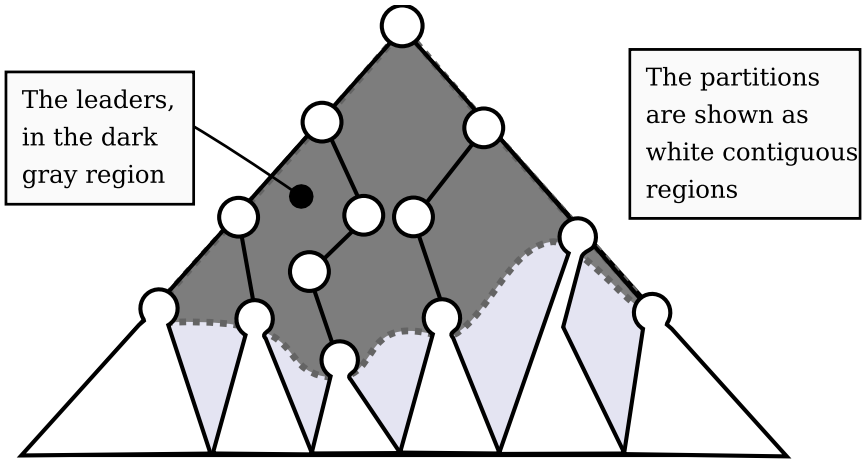


Figure 5.4: A depiction of the treap partitioning scheme.

insertions, fast finger deletions, and find operations take expected $O(1)$ *time. Furthermore, this implementation requires only* $O(\log N)$ *random bits, and requires only* $O(N)$ *space to store an ordered set partition of capacity* N.

Let T be the treap storing the partition. In addition to the leader fields, we must store some additional information to support fast finger insertions and deletions. One approach is to store the size of the subtree T_v for all nodes v. Unfortunately this would require updating the subtrees sizes for each ancestor of a newly inserted node, thus guaranteeing the insert operation will take expected $\Omega(\log n)$ time. Instead we will store the subtree size information for some subset of these nodes. Define the *leader frontier* $\mathcal{F}$ as the elements of $\mathcal{L}_s[T]$ which have more than one element in their partition set (or equivalently, that have at least one child with weight less than s). To maintain unique representation, each node v will have a size field, and we will maintain an invariant on what the contents of these fields must be. We will first describe an invariant in Section 5.3.2 that leads to expected $\Theta(\log \beta)$ time finger insertions and deletions as opposed to expected $O(1)$ time as promised by Theorem 10. We then describe a modified invariant with modified operations in Section 5.3.3 that support finger insertions and deletions in expected $O(1)$ time. We also provide detailed correctness proofs for both versions.

5.3.2 The Basic Version

In this section we will use the following invariant.

The basic size field invariant:
For all v with $|T_v| < s$, $v.\mathsf{size} = |T_v|$. Otherwise $v.\mathsf{size} = \infty$.

The operations may then be implemented as follows.

Insertion: To insert a key, create a new node u with that key, then search for the predecessor v of u via treap search, and finally perform a fast finger insertion of u using v.

Finger Insertion: Suppose we have created a node u with the key to be inserted and have its predecessor v and now now wish to insert it. Proceed with a treap fast finger insert of u. We must be careful to maintain the basic size field invariant. Thus, as we rotate u up from its initial starting point – the leaf it would occupy if it had the lowest priority of all keys – we update the size fields during the rotations. Once u has reached its proper location, if necessary we continue up the u to root path, incrementing the size fields until we reach a node w with $|T_w| \geq s$.

Next, we must update the leader fields. Set $u.\mathsf{leader}$ appropriately. It is possible that inserting u resulted in the "promotion" of a node w to leadership status. We can detect this by determining if the child w of $\mathrm{node}(\mathrm{node}(u.\mathsf{parent}).\mathsf{leader})$ that is an ancestor of u has $|T_w| = s$, using the size fields of the children of w. In this event, traverse w's subtree setting the leader fields of all of its descendants to $w.\mathsf{key}$. Then set size field of w and its parent to ∞.

If u is itself a leader, find its predecessor a and successor b. Proceed up the a-to-root and b-to-root paths looking for the deepest nodes with subtree size at least s, denoted $\ell(a)$ and $\ell(b)$. Let a' and b' be the children of $\ell(a)$ and $\ell(b)$ that are ancestors of a and b, respectively. Update the leader fields of all of the descendants of a' (to $\ell(a).\mathsf{key}$) and b' (to $\ell(b).\mathsf{key}$) via subtree traversals. Also update the leader fields of $\ell(a)$ and $\ell(b)$ if necessary. Finally, set $u.\mathsf{leader} = u.\mathsf{key}$.

Finger Deletion: Deletion is similar to insertion. Let u be the node to be deleted. (Note that given a key to be deleted, we can find its corresponding node in expected constant time using the underlying hash table.) Rotate u down to a leaf position, updating the subtree size information appropriately on each rotation. Make sure to account for the fact that u will ultimately be deleted from the treap when updating these size fields.

If initially $|T_u| < s$, and $|T_{\mathrm{node}(u.\mathsf{leader})}| > s$ (i.e., deleting u does not result in any "demotions" of nodes from leaders to non-leaders), just delete u.

If initially $|T_u| < s$, and $|T_{\mathrm{node}(u.\mathsf{leader})}| = s$ (so that u's leader must be demoted), traverse $T_{\mathrm{node}(u.\mathsf{leader})}$, setting the leader field of each node in that subtree to the key of the parent of $\mathrm{node}(u.\mathsf{leader})$.

If u was initially a leader, maintain a list L of nodes x such that we rotated on edge $\{x, u\}$ when rotating u down to a leaf position. Delete u from the treap. Find the lowest priority node v in L that has $|T_v| \geq s$. For each node $x \in L$, in order of increasing priority, if x is a descendant of v and has a child x' with $x'.\mathsf{leader} \neq v.\mathsf{key}$, update the leader field of each node in $T_{x'}$ to $v.\mathsf{key}$. Also set $x.\mathsf{leader} = v.\mathsf{key}$ For each ancestor x of v in L, set $x.\mathsf{leader} = x.\mathsf{key}$. Also, if x has a child y of with $|T_y| < s$ then for each such child y with $|T_y| < s$ test if $y.\mathsf{leader} \neq x.\mathsf{key}$. If so, do a traversal of T_y and update the leader field of each node in that subtree to $x.\mathsf{key}$.

Find: Given an input key, find the corresponding node u using the underlying hash table, then simply search for the node with key $u.\mathsf{leader}$ in the hash table storing T in expected $O(1)$ time.

Proof of Correctness. We break the proof of correctness into the following three propositions.

Proposition 4. *The above data structure is a correct uniquely represented implementation of the ordered set partitioning abstract data type, assuming the basic size field invariant is maintained, and that the leader fields are set according to the specification in Section 5.3.1.*

Proof: The data structure is clearly uniquely represented, assuming the operations maintain the alternative size field invariant and the correct leader fields. The treap is uniquely represented. The size fields are a deterministic function of the treap together with the constraint imposed by the alternative size field invariant. The induced partitioning $\{\{y \in T : \ell(y, T) = x\} : x \in \mathcal{L}_s[T]\}$ is also a deterministic function of the treap, as are the correct contents of the leader fields. Everything is thus a function of the current abstract data type state and the RAM's random bits, and is uniquely represented. ∎

Note that if the basic size field invariant is maintained and the leader fields are set correctly, then the find operation is clearly correct. The remainder of this section

is devoted to showing that the insertion and deletion operations described above do in fact maintain the alternative size field invariant and the correct leader fields.

Proposition 5. *Insertions are implemented in such a way as to maintain the correct setting of the size and leader fields.*

Proof: Suppose we insert a node u containing a new key. The basic size field invariant is fairly easy to maintain during rotations. Rotating on edge $\{x, y\}$ only affects the subtree size of T_x and T_y, and we can update the field size as normal, using the modified arithmetic that $z + \infty = \infty$ for all quantities z.

So consider the task of updating the leader fields. The two cases that $|T_u| < s$ (both with and without a node promotion) are straightforward. Thus we focus on the case that u is a leader.

By Lemma 2 on page 90, inserting u can only change the leaders of nodes v within distance s from u. Proceeding from the predecessor a of u to its post-insertion leader $\ell(a)$, we claim that the subtree $T_{\ell(a)}$ contains all of nodes within distance s of u that are less than u. If this is not the case, there must be a node x within distance s from u not in $T_{\ell(a)}$, which can only happen if x has higher priority than $\ell(a)$. If x has higher priority than $\ell(a)$, it must be an ancestor of $\ell(a)$. If $\ell(a) < x$, then this contradicts the fact that $\ell(a)$ is an ancestor of a. If $x < \ell(a)$, then this contradicts the fact that $|T_{\ell(a)}| \geq s$, since it can only contain keys between x and u exclusive.

Thus $T_{\ell(a)}$ contains all of nodes within distance s of u that are less than u, and their leader fields should all be set to $\ell(a)$. While this can be done in $O(s)$ time, the operation specifies updating only $\ell(a)$.leader and the leader fields of nodes in the subtree of a', the child of $\ell(a)$ that is an ancestor of a. Let a'' be the other child of $\ell(a)$. We claim that the leader fields of its descendants should remain unchanged. This is because the insertion of u leaves $T_{a''}$ unaffected, and can only decrease $|T_{\ell(a)}|$ which is at least s after the insertion, and thus must have been at least s beforehand. Therefore the insertion correctly updates the leader fields of all nodes within distance s from u that are less than u. By a symmetric argument, the insertion correctly updates the leader fields of all nodes within distance s from u that are greater than u. Since we also set u.leader correctly, Lemma 2 on page 90 guarantees that we have updated all of the leader fields correctly. ■

Proposition 6. *Deletions are implemented in such a way as to maintain the correct setting of the size and leader fields.*

Proof: Suppose we delete a node u. As with insertions, the basic size field invariant is fairly easy to maintain during rotations. We thus omit the correctness proof for it and focus on the leader fields.

The two cases that $|T_u| < s$ (both with and without a node demotion) are straightforward. Thus we focus on the case that u is a leader.

We claim that the only way a node x can have its leader changed during the deletion of u is to have an ancestor on the rotation path of u consisting of the nodes $\{y : \text{we rotated on } \{u, y\} \text{ during this operation}\}$. This is because a change of leader for x can occur only due to one of following two scenarios.

1. The set of ancestors of x changes (i.e., an ancestor a of x changes parent during a rotation on its parent).
2. The subtree size of an ancestor y of x changes (which can only occur during a rotation involving y).

Next we claim that if a node x in a subtree of some node y on the rotation path of u has a change in leader, then so does the child of y that is an ancestor of x. Suppose x's change in leader is due to the first case above. That is, suppose an ancestor a of x has its parent changed to y during a rotation involving y. It is not hard to see that changing a's parent to y resulted in a change of leader for x, then a is not a leader so $|T_a| < s$, which implies that the leader of every other node in T_a changes as well. Next, suppose x's change in leader is due to the second case above. In the second case, let a be the child of y that is an ancestor of x. Again, a cannot be a leader, so the change of leader experienced by x also occurs for all nodes in T_a.

The above claim implies that to find all the nodes whose leader fields need to be changed, we need only find all the nodes whose leader fields need to be changed among the children of nodes in the list L of rotated nodes. The delete operation does this explicitly for nodes whose leader field should be set to v.key, where v is the lowest priority node in L that has $|T_v| \geq s$. For all other nodes $w \in L$, namely those with priority greater than v, any node x whose leader field should be set to w.key satisfies the following property: w is the deepest node in L that is an ancestor of x. Thus in this case, the child of w that is an ancestor of x also has its leader field set incorrectly, and we detect this. Thus the deletion operation correctly detects all nodes that require updates to their leader field, and updates them appropriately. ■

5.3.3 Achieving Expected Constant Time Updates

Though we have not proved it, the running time of the finger insert and delete operations in Section 5.3.2 are dominated by the need to update the size fields of all ancestors v of u with $|T_v| < s$ in the common case that u is not a leader. In expectation there will be $\Theta(\log \beta)$ such nodes. Using the size field invariant below, we can reduce the number of nodes whose size field we must update in the common case that u is not a leader to at most 3.

The size field invariant: For all v that are children of nodes in $\mathcal{F}$, $v.\mathsf{size} = |T_v|$. For all v that are ancestors of some node in $\mathcal{F}$, $v.\mathsf{size} = \infty$. For all remaining nodes v, $v.\mathsf{size} = \mathsf{null}$.

Standard insertion and find operations proceed exactly as in Section 5.3.2. The fast finger updates proceed as follows.

Finger Insertion: Suppose we have created a node u with the key to be inserted and have its predecessor v and now now wish to insert it. Proceed with a treap fast finger insert of u. For now, make no changes to the size field of any node.

Next, we must update the size and leader fields. Determine if u is a leader by testing if either of the following conditions is true:

1. u has a child v with $v.\mathsf{size} = \infty$.
2. u has a child v with $v.\mathsf{size} = s - 1$.

We will first consider the case that u is not a leader. It is possible that inserting u resulted in the "promotion" of a node w to leadership status. Determine if this is the case as follows. Set the leader field of u to equal the leader field of its parent node. Find the child of $\mathrm{node}(u.\mathsf{leader})$ that is an ancestor of u, which we denote by u'. Then there is a promotion if and only if $u'.\mathsf{size} = s - 1$. If there is no promotion and $u' \neq u$, then merely increment $u'.\mathsf{size}$. If there is no promotion and $u' = u$, then set $u.\mathsf{size}$ equal to one plus the maximum value of the size fields among u's children, and set the size fields of u's children to null. If u' is promoted, set $u'.\mathsf{size} = \infty$ and traverse $T_{u'}$ changing the leader fields of all the nodes therein to $u'.\mathsf{key}$, and simultaneously compute the size of the subtrees of T rooted at the children of u'. Update the size fields of the children of u' to their subtree sizes accordingly.

If u is itself a leader, find its predecessor a and successor b. Proceed up the a-to-root and b-to-root paths looking for the deepest nodes with subtree size at least s, denoted $\ell(a)$ and $\ell(b)$. We can find $\ell(a)$ and $\ell(b)$ in expected $O(s)$ time without using any of the size fields. (For example, to find $\ell(a)$ proceed up the a-to-u path P, and compute the subtree size of a node $v \in P$ by traversing the subtree of the child of v not in P to compute its size, and combining this information with the previously computed subtree size of v's child in P. Of course, we may stop in the middle of a traversal as soon as we detect that the subtree is large enough to ensure v is a leader.)

Let a' and b' be the children of $\ell(a)$ and $\ell(b)$ that are ancestors of a and b, respectively. Update the leader fields of all of the descendants of a' (to $\ell(a)$.key) and b' (to $\ell(b)$.key) via subtree traversals. Also update the size and leader fields of $\ell(a)$ and $\ell(b)$ if necessary. Set the size fields of a' and b' to $|T_{a'}|$ and $|T_{b'}|$, respectively. Also, for each node v that is either in the a-to-a' path or is a child of a node that is (with the exception of a'), set v.size $=$ null. Likewise, for each node v that is either in the b-to-b' path or is a child of a node that is (with the exception of b'), set v.size $=$ null. Finally, set u.leader $= u$.key.

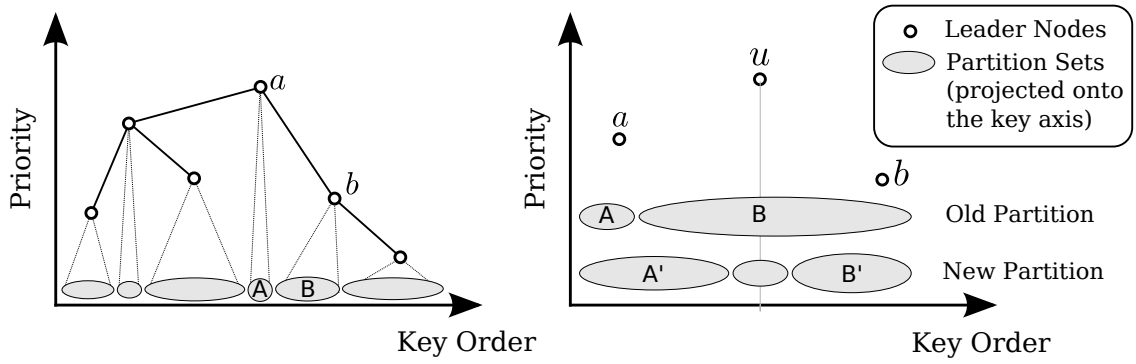
Finger Deletion: Let u be the node to be deleted. (Note that given a key to be deleted, we can find its corresponding node in expected constant time using the underlying hash table.) Add the size fields of u's leader's children to determine if $|T_{\text{node}(u.\text{leader})}| > s$. Rotate u down to a leaf position.

If u was not initially a leader (i.e., u.size $< \infty$), then there are two cases to consider. Either deleting u will result in a demotion of some node, or not. If $|T_{\text{node}(u.\text{leader})}| > s$, then no demotion occurs. In this case, simply decrement the size field of of child of u's leader that is an ancestor of u, and delete u from the treap. Otherwise the leader of u is demoted when u is deleted. Let $\ell(u) := \text{node}(u.\text{leader})$ be the leader of u. In this case, set $\ell(u)$.size $= s-1$, and set the size fields of the children of $\ell(u)$ to be null. Then traverse $T_{\ell(u)}$, setting the leader field of each node in that subtree to the key of the parent of $\ell(u)$. Finally, delete u from the treap.

If u was initially a leader, maintain a list L of nodes x such that we rotated on edge $\{x, u\}$ when rotating u down to a leaf position. Keep L sorted in order of depth. Delete u from the treap. Find the deepest node v in this list with $|T_v| \geq s$ in $O(s)$ using tree traversals, as discussed in the finger insert operation. Set the size fields of the children of v to their subtree sizes, computed via tree traversal. Set the size field of all ancestors of v in L to ∞. For all children w of ancestors of v in L whose size field is null, set w.size $= |T_w|$

as computed via a traversal of T_w. Next, for each node w that is a descendant of v in L, or is a child of a descendant of v in L, with the exception of v and its children, set $w.\mathsf{size} = \mathsf{null}$.

That takes care of the size fields. For the leader fields, we proceed as in Section 5.3.2. For each descendant w of v in L, in order of increasing priority, if w has a child w' with $w'.\mathsf{leader} \neq v.\mathsf{key}$, update the leader field of each node in $T_{w'}$ to $v.\mathsf{key}$. Also set $w.\mathsf{leader} = v.\mathsf{key}$. For each ancestor w of v in L, set $w.\mathsf{leader} = w.\mathsf{key}$. Also, if there is a child y of w with $|T_y| < s$ (as indicated by $y.\mathsf{size} < \infty$), then for each such child y with $|T_y| < s$ test if $y.\mathsf{leader} \neq w.\mathsf{key}$. If so, do a traversal of T_y and update the leader field of each node in that subtree to $w.\mathsf{key}$.



An alternate Illustration of Treap Partitioning. Leader nodes provide the boundaries for the partition sets.

A depiction of how the partitions may change when a high priority node u is inserted. At most one old partition set is split.

Figure 5.5: An illustration of an update to a Treap Partitioning instance.

Proof of Correctness. Assume that the finger insertion correctly determines if u is a leader or if u causes the promotion of another node or if u does not alter the set of leaders. Further assume that the deletion operation correctly determines if u is a leader or if deleting u causes a demotion of some node. Given these assumptions, it is not hard to prove that the fast finger insertion and deletion operations described above update the leader fields exactly as in the basic version of Section 5.3.2, so the proofs of correctness in that section apply to them with respect to the leader fields. We will thus prove that these assumptions hold, and that the size fields are updated correctly, so as to maintain the size field invariant.

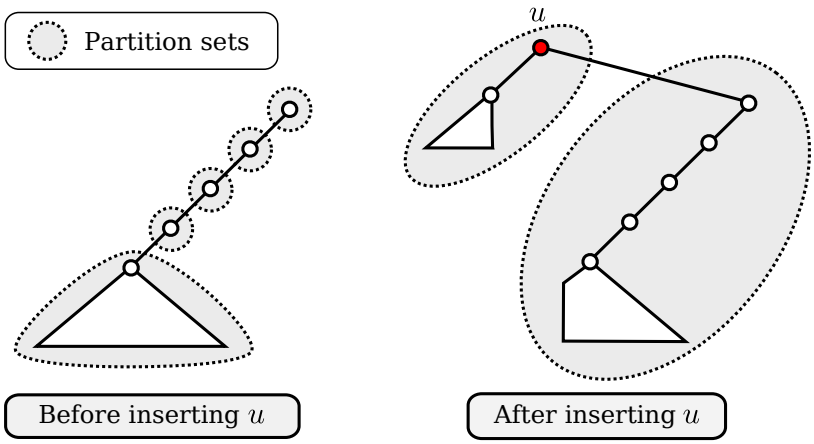


Figure 5.6: An illustration of an update to a Treap Partitioning instance, showing how a single operation may lead to several partition sets being merged together.

Proposition 7. *The finger insertion operation described above correctly determines if u is a leader, and correctly determines if inserting u causes a promotion of some node.*

Proof: We start with the task of detecting if u is a leader. Fix a treap T', and insert node u to obtain treap T. Node u is inserted into T' by inserting it at the leaf position $\text{leaf}(u, T')$ it would have occupied if it had the lowest priority of all keys, and rotating u up to its correct location. By assumption, the nodes of T' previously satisfied the size field invariant, and the size fields were not altered as u was rotated up. Let v be the child of u such that $\{u, v\}$ was the final edge rotated on as we rotated u up. (If v does not exist, then u is a leaf, and hence obvious not a weight s leader if $s > 1$). We claim $|T_u| = |T'_v| + 1$. To simplify the proof of this claim, let us add two infinite priority nodes to T' — one less than all nodes in T and one greater than all nodes of T. (This simply adds a parent and a grandparent for the root of T'.) Let a (respectively, b) be the minimum distance node from v that is less than (respectively, greater than) v and has higher priority than v. Then $|T'_v| = \text{dist}_{T'}(a, b) - 1$, where $\text{dist}_{T'}(a, b)$ is the distance between a and b among the set of keys stored in T'. Moreover, the lower priority node in $\{a, b\}$ is the parent of v in T'. Therefore, the lower priority node in $\{a, b\}$ is the parent of u in T, so a and b both have higher priority than u. Since u has higher priority than all nodes strictly between a and b,

$$|T_v| = \text{dist}_T(a, b) - 1 = \text{dist}_{T'}(a, b) = |T'_v| + 1.$$

Thus, u is a leader if and only if $|T'_v| \geq s-1$, which occurs if and only if $v.\mathsf{size} \geq s-1$ (where for all integers m we let $m < \infty$ and $\mathsf{null} < m$ by convention). Of course, this is precisely what the operation tests for.

Next we consider the task of determining if inserting u resulted in the "promotion" of a node w to leadership status. This can only occur if u is not a leader, since the promoted node w must be an ancestor of u and if u is a leader in T, then

$$s \leq |T_u| \leq |T_w| - 1 = |T'_w|$$

which contradicts the fact that w was not a leader in T'.

So assume that u is not a leader. Let $\ell(u, T)$ be the leader of u in T. If a node was promoted by the insertion of u then it must have been $\ell(u, T)$, and $\ell(u, T)$ was promoted if and only if $|T'_{\ell(u,T)}| = s - 1$.

We first show that if there is a promotion, then we will detect it. From $|T'_{\ell(u,T)}| = s - 1$ we infer that the parent v of $\ell(u, T)$ in T' was a weight s leader in T'. Also, since v is an ancestor of u, $|T_v| = |T'_v| + 1$ so v is a leader in T and furthermore, every node in the subtree of T' rooted at $\ell(u, T)$ has v as its leader in T'. Moreover, since u is not a leader, its parent node must be a descendant of its leader $\ell(u, T)$, so $v = \mathrm{node}(\mathrm{node}(u.\mathsf{parent}).\mathsf{leader})$ and the operation correctly detects the promotion by finding the child u' of v that is an ancestor of u, and determining that u' is promoted if $u'.\mathsf{size} = s - 1$.

Next, assume that u is not a leader and there was no promotion. If there is not a promotion, then $\ell(u, T)$ is the leader of u's parent node in T'. Thus $\ell(u, T)$ equals $\mathrm{node}(\mathrm{node}(u.\mathsf{parent}).\mathsf{leader})$, and so the children of that node have subtree sizes less than $s - 1$ by assumption, and the operation correctly determines this. ∎

Proposition 8. *The deletion operation described above correctly determines if u is a leader, and correctly determines if deleting u causes a demotion of some node.*

Proof: Clearly, u is a leader if and only if $u.\mathsf{size} = \infty$ prior to the deletion. If a node is demoted, then it is $\ell(u, T) = \mathrm{node}(u.\mathsf{leader})$. This occurs if and only if $|T_{\ell(u,T)}| = s$, which we can determine by summing the size fields of the children of $\ell(u, T)$, using $m + \infty = \infty$ for all integers m. ∎

Proposition 9. *The finger insertion and deletion operations described above correctly update the size fields.*

Proof: Let u be the node that we either insert or delete. In the case that u is not a leader, updating the size fields to maintain the size field invariant is fairly straightforward, so we omit the correctness proof for that case.

So suppose u is a leader. To maintain the size field invariant, we need only worry about three groups of nodes:

1. Nodes whose leadership status changes during the operation.
2. Children of nodes whose leadership status changes during the operation.
3. Nodes whose subtree size changes during the operation.

Any node that is not in one of these groups should not have its size field changed in response to the operation.

First consider a $\text{finger-delete}(u)$ operation. This operation stores a temporary list L containing nodes on the rotation path of u. We claim that the nodes in the three groups above are a subset of nodes in L and their children. Observe that for any node v, there are only two ways $|T_v|$ can change during the deletion of u. The first is that v is involved in a rotation about edge $\{u, v\}$ during the deletion (i.e., $v \in L$). The second is that v was an ancestor of u. Clearly, the only way the leadership status of v can change is via a change in $|T_v|$. Finally, note that since u is leader in T, its ancestors have subtree sizes of at least $s + 1$, and will not be demoted by by the deletion of u. Thus leadership status does not change and their size fields should remain unaltered at ∞. It is not hard to verify that the operation correctly sets the size field of all nodes in L as well as all children of nodes in L.

Next consider a $\text{finger-insert}(u)$ operation. As with deletion, the changes in leadership status require changes in subtree size, and changes in subtree size occur only for nodes on the rotation path P of u. Also, since u is leader in T, its ancestors have subtree sizes of at least $s + 1$, and thus were leaders before the insertion of u. It follows that the nodes in the three groups above are a subset of nodes in P and their children.

We next claim that P is a subset of the nodes in the a-to-u path P_a and the nodes in the b-to-u path P_b, where a and b are the predecessor and successor of u, respectively. This can be proved by induction. Let $\hat{P}_a$ and $\hat{P}_b$ denote the a-to-u path and b-to-u path after the insertion of u is completed. We allow P_a and P_b denote the corresponding paths at any point in time, including in the middle of the insertion. Our induction hypothesis is that after k rotations of u, there are non-negative integers k_a and k_b such that

- $k_a + k_b = k$.
- If $k_a > 0$ then $a \in T_u$ and the a-to-u path consists of the first k_a nodes in $\hat{P}_a$.
- If $k_b > 0$ then $b \in T_u$ and the b-to-u path consists of the first k_b nodes in $\hat{P}_b$.
- The a-to-u and b-to-u paths consist entirely of nodes in the rotation path P.

We omit the details.

We have show that we need only consider $P_a \cup P_b$ and the children of nodes in $P_a \cup P_b$. We claim that $\ell(a)$ and $\ell(b)$ and their ancestors were leaders before the insertion of u, and so their size fields remain unchanged. This is because inserting a node u with higher priority that a node v can either split T_v, thus reducing its size, or leave it alone. Furthermore, the children of ancestors of $\ell(a)$ and $\ell(b)$ (except for the children of $\ell(a)$ and $\ell(b)$ themselves) were not involved in rotations, and thus their size fields remain unchanged as well. Thus, for the purposes of updating the size fields, we only need to consider elements in the a-to-$\ell(a)$ and b-to-$\ell(b)$ paths, together with their children. These are precisely the nodes the operation updates, and it is easy to prove that it sets these fields correctly. ■

5.3.4 Performance Analysis and Some Additional Facts

Before proving Theorem 10 on page 78, we will prove several useful lemmata.

Lemma 2. *Fix any treap T. Inserting or deleting a node u can alter the assignment of nodes to their weight s leaders in T for at most $2s$ other nodes in T, namely those within distance s of u.*

Proof: We will prove this lemma by showing how to find the leader of any node x while inspecting only those nodes y at distance at most s from x. This guarantees that the existence or non-existence of other nodes at distance greater than s from x can have no effect on who the leader of x is.

Let $\mathrm{dist}(u, v)$ denote the distance between u and v in key-space. Thus $\mathrm{dist}(u, v) = k$ if there are $k + 1$ keys between u and v, inclusive. Fix a treap T on elements U. Let $A := \{y : y \in U, y \leq x, \text{ and } \mathrm{dist}(x, y) \leq s\}$ and $B := \{y : y \in U, y > x, \text{ and } \mathrm{dist}(x, y) \leq s\}$ be the keys within distance s of x. Let a and b be the maximum priority elements of A and B, respectively. Without loss of generality, suppose a has a higher priority than b. By the definition of a treap, all the keys in $A \cup B$ must be descendants of a, and all the keys in B must be descendants of b. If we assume that $|A| = |B| = s$, then we may infer that a and b are weight s leaders in T. Furthermore, the path from x to the root of T must go through b. The leader of x is thus the deepest node on the path from x to b with weight at least s, and this node must lie between a and b, inclusive.

If $|A| < s$ and $|B| = s$, we may safely add a dummy node that is less than all elements of A in key order and has infinite priority, and apply the previous argument. Likewise, if $|B| < s$ and $|A| = s$, we may safely add a dummy node that

is greater than all elements of B in key order and has infinite priority, and apply the previous argument. Finally, if $|A| < s$ and $|B| < s$, then there are no nodes at distance greater than s from x, so there is nothing to prove. ∎

Lemma 3. *Fix any treap T. Inserting or deleting a node u can alter the size fields (set according to the size field invariant) for at most $2s$ other nodes in T, namely those within distance s of u.*

The proof of Lemma 3 is very similar to the proof of Lemma 2, so we omit it.

Lemma 4. *Let $s \in \mathbb{Z}^+$ and let T be a treap of size n with priorities generated by an 11-wise independent hash function h from keys to $[r]$ for some $r \geq n^3$. Then $\mathbf{Pr}[|T_v| = k] = O(1/k^2)$ for any $1 \leq k < n$, $\mathbf{Pr}[|T_v| = n] = O(1/n)$, and for any $s \geq 1$, $\mathbf{Pr}[|T_v| \geq s] = O(1/s)$, so each node is a weight s partition leader with probability $O(1/s)$.*

Proof: It is easy to prove that probability of a collision on priorities is at most $\binom{n}{2}/r < 1/2n$. In this case, we use the trivial bound $|T_v| \leq n$ to show that

$$\mathbf{Pr}[w(x) = k] \leq \mathbf{Pr}[w(x) = k \mid \text{no collisions}] + 1/2n$$

So assume there are no collisions on priorities. Let S be the set of points in T. A useful lemma due to Mulmuley [Mul94] implies that a collection $\mathcal{X}$ of random variables satisfies the d-max property (see Definition 9 on the next page) if $\mathcal{X}$ is $(3d + 2)$-wise independent, and thus if h is the priority generating hash function and U is any set of keys, $\{h(u) : u \in U\}$ satisfies the d-max property for $d \leq 3$.

Now consider the probability that $|T_v| = k$ for some k. Relabel the nodes $\{1, 2, \ldots, n\}$ by their key order. Note that $|T_v| = k$ if and only if the maximal interval I containing v such that v has the maximum priority in I has size $|I| = k$. If $k = n$, then by the 1-max property $\mathbf{Pr}[|T_v| = n] = O(1/n)$. So suppose $k < n$, and further suppose $I = [a : b] := \{a, a + 1, \ldots, b\}$. Then either $a - 1$ has priority higher than v, or $a - 1$ does not exist, and similarly with $b + 1$. If $a - 1$ does not exist and $|I| = k$, then $h(b + 1) > h(v) > \max\{h(i) : i \in I \text{ and } i \neq v\}$, which occurs with probability $O(1/k^2)$ by the 2-max property. The case that $b + 1$ does not exist is analogous. If both $a - 1$ and $b + 1$ exist, then $\min\{h(a - 1), h(b + 1)\} > h(v) > \max\{h(i) : i \in I \text{ and } i \neq v\}$. By the 3-max property, this occurs with probably $O(1/k^3)$.

Taking a union bound over all intervals $I \subset [1 : n]$ of length k that contain v, we obtain

$$\mathbf{Pr}[|T_v| = k] = O(1/k^2).$$

Given this fact, it is straightforward to show that $\mathbf{Pr}[|T_v| \geq s] = O(1/s)$ for all $s \geq 1$, which completes the proof. ■

Definition 9 (d-max property)**.** *A finite collection $\mathcal{X}$ of random variables has the d-max property iff there is some constant c such that for any subset of $\mathcal{X}$ and for any enumeration $X_1, X_2, \ldots, X_m$ of the elements of that subset, we have*

$$\mathbf{Pr}[X_1 > X_2 > \cdots > X_d > \max\{X_i \ : \ i > d\}] \leq c/m^{\underline{d}}$$

where $m^{\underline{d}} := \prod_{i=0}^{d-1}(m-i)$.

If the priorities are generated via a truly random hash function without collisions, then it is relatively easy to work out the appropriate constants for Lemma 4. For any node v and integer k such that $1 \leq k < n$, $\mathbf{Pr}[|T_v| = k] \leq 3/k^2$. Furthermore, if v has at least k nodes less than it and greater than it, then $\mathbf{Pr}[|T_v| = k] = \frac{2}{(k+1)(k+2)}$.

Lemma 5. *Let $s \in \mathbb{Z}^+$ and let T be a treap of size at least s with priorities generated by an 11-wise independent hash function from keys to $[N^3]$, where N is an upper bound on the size of T. Let T' be the treap induced on the weight s partition leaders in T. Then the probability that inserting a new element into T or deleting an element from T alters the structure of T' is at most c/s for some absolute constant c.*

Proof: We consider insertions first. There are only two ways the insertion of u into T could change T'. The first is that u may become a leader. Lemma 4 guarantees that this occurs with probability $O(1/s)$. The other possibility is that u may cause an "overflow", resulting in the promotion of a node that formerly had exactly $s-1$ descendants to leadership status. By Lemma 2, if an overflow occurs it must occur in a node within distance s of u. There are at most $2s$ such nodes, and by Lemma 4 each has a $O(1/s^2)$ probability of having exactly $s-1$ descendants prior to the insertion of u. Taking a union bound over these $2s$ possibilities, the probability of an overflow is thus $O(1/s)$.

Now consider deletions. Let S be the set of nodes in T. By the unique representation property, deleting u returns T to the unique machine state with key set $S \setminus \{u\}$. The probability that this deletion alters T' is precisely equal to the probability that inserting u into a treap storing $S \setminus \{u\}$ alters T'. However we have already proved above that the probability of the latter event is $O(1/s)$. ■

Lemma 6. *Let $s \in \mathbb{Z}^+$ and let T be a treap of size at least s with priorities generated by a 12-wise independent hash function from keys to $[N^3]$, where N is an*

upper bound on the size of T. Let T' be the result of adding an element to T. Then $\mathbf{E}[|\mathcal{L}_s[T] \triangle \mathcal{L}_s[T']|] \leq c/s$ for some absolute constant c. Here $X \triangle Y := (X - Y) \cup (Y - X)$. Similarly, if deleting an element from T yields treap T'', then $\mathbf{E}[|\mathcal{L}_s[T] \triangle \mathcal{L}_s[T'']|] \leq c/s$ for some absolute constant c.

Proof: By unique representation, it suffices to prove the lemma for insertions, as each deletion can be interpreted as "undoing" some corresponding insertion. So consider the insertion of x into T to yield treap T'. Let $X := |\mathcal{L}_s[T] \triangle \mathcal{L}_s[T']|$, $X^+ = |\mathcal{L}_s[T'] \setminus \mathcal{L}_s[T]|$ and let $X^- = |\mathcal{L}_s[T] \setminus \mathcal{L}_s[T']|$, so that $X = X^- + X^+$.

We first claim that $X^+ \leq 1$ unconditionally. Note that if y is an ancestor of x in T', then $|T'_y| = |T_y| + 1$. Otherwise, $|T'_y| \leq |T_y|$ as inserting x either leaves T_y unaffected or "cuts away" nodes from T_y. For example, if $y < x$ and y has lower priority than x, then inserting x will cut away all nodes in T_y greater than x; these nodes will not be present in T'_y. Thus, if $X^+ > 0$ then $\mathcal{L}_s[T'] \setminus \mathcal{L}_s[T]$ consists of ancestors of x, possibly including x. However, if $x \in \mathcal{L}_s[T'] \setminus \mathcal{L}_s[T]$, then all of its proper ancestors y satisfy $|T'_y| \geq s + 1$, so $|T_y| \geq s$ and y was a leader in T. So $x \in \mathcal{L}_s[T'] \setminus \mathcal{L}_s[T]$ implies $X^+ = 1$. If $x \notin \mathcal{L}_s[T'] \setminus \mathcal{L}_s[T]$, it is clear that there is at most one ancestor y of x such that $|T_y| < s$ and $|T'_y| \geq s$, so $X^+ \leq 1$ in this case as well.

Next we claim that $\mathbf{E}[X^- \mid x \notin \mathcal{L}_s[T']] = 0$. Each element $y \in \mathcal{L}_s[T] \setminus \mathcal{L}_s[T']$ must have had its subtree size reduced when x was inserted. Note that only nodes on the rotation path for x (from the leaf position x would have occupied if it had priority $-\infty$ to its current location) can have their subtree sizes reduced. All of these nodes are descendants of x in T'. If x is not a leader, which implies that none of its descendants are either. Thus $X^- = 0$ in this case.

Finally we claim that $\mathbf{E}[X^- \mid x \in \mathcal{L}_s[T']] = O(1)$. Lemma 2 implies that there are only elements within distance s of x can lose their leadership status in response to the insertion of x. However, by Lemma 4 each of these elements is a leader in T with probability $O(1/s)$, so only $O(1)$ of these elements are leaders in T in expectation. (We can apply Lemma 4 independent of the priority of x since fixing the priority of x induces an 11-wise independent hash family on the remaining priorities.) This immediately yields a bound on X^- conditioned on $x \in \mathcal{L}_s[T']$.

We bound $\mathbf{E}[X] = \mathbf{E}[X^+] + \mathbf{E}[X^-]$ as follows. Since $X^+ \leq 1$ unconditionally, $\mathbf{E}[X^+] = \mathbf{Pr}[X^+ > 0] \leq c/s$ from Lemma 5. Furthermore, $\mathbf{E}[X^- \mid x \notin \mathcal{L}_s[T']] = 0$, so that $\mathbf{E}[X^-] = \mathbf{E}[X^- \mid x \in \mathcal{L}_s[T']] \cdot \mathbf{Pr}[x \in \mathcal{L}_s[T']]$. From our remarks above and Lemma 4, this is bounded by c'/s for some absolute constant c'. ∎

Lemma 7. *Let $s \in \mathbb{Z}^+$ and let T be a treap of size n with priorities generated by an 11-wise independent hash function h from keys to $[r]$ for some $r \geq n^3$. Then the expected size of a partition in the weight s treap partition of T is $\Omega(s)$.*

Proof: Let X be a random variable equal to the number of weight s leaders in T. By Lemma 4, for any node v, $\mathbf{Pr}[|T_v| \geq s] = O(1/s)$, so by linearity of expectation $\mathbf{E}[X] = \sum_v \mathbf{Pr}[|T_v| \geq s] \leq cn/s$ for some suitable constant c. Moreover, for any fixed set of leaders, the average size of a partition is exactly n/X. By Jensen's inequality (reproduced as Theorem 11 below, see [Ste04] for a treatment), for any positive random variable such as X, $\mathbf{E}\left[\frac{1}{X}\right] \geq \frac{1}{\mathbf{E}[X]}$. Thus the expected average size of a partition is at least $\frac{n}{\mathbf{E}[X]} \geq \frac{s}{c}$. ∎

Theorem 11 (Jensen's Inequality [Jen06])**.** *Fix a convex function $f : [a, b] \to \mathbb{R}$ and a finite sequence of nonnegative numbers $\{p_i\}_{i=1}^n$ whose sum is one. Fix $x_i \in [a, b]$ for $i = 1, 2, \ldots, n$. Then $f(\sum_{i=1}^n p_i x_i) \leq \sum_{i=1}^n p_i f(x_i)$.*

We now commence with the proof of Theorem 10, whose statement may be found on page 78.

Proof of Theorem 10: We are only using 11-wise independent hash functions to generate priorities, so $O(\log N)$ random bits suffice. The fact that the data structure requires only $O(N)$ space to store an ordered set partition of capacity N follows from the space efficiency of our uniquely represented bounded treap (see Section 4.5). The expected $O(\log n)$ time insertion follows from the $O(\log n)$ expected depth of a random treap, assuming fast finger search takes $O(\log n)$ expected time. Clearly, find operations take expected $O(1)$ time, and could in fact be implemented in worst-case constant time using a dynamic perfect hash table to map the treap nodes onto the RAM memory. It remains only to prove that fast finger insertions and deletions take expected $O(1)$ time.

We begin with fast finger insertions. The high-level approach is to show that either inserting u takes expected $O(1)$ time if u does not changes the set of leaders, and takes expected $O(s)$ time if u changes the set of leaders. Since the latter event occurs with probability $O(1/s)$ by Lemma 5, the overall time bound is still constant in expectation.

So consider a fast finger insertion of u. The actual insertion (i.e., the fast finger search and rotations) takes expected $O(1)$ time [SA96]. The tricky part is maintaining the additional state, namely the leader and size fields. If u's proper location is below the frontier $\mathcal{F}$ and no node is promoted to a leader by the addition of u, then it is easy to see that the insertion takes expected $O(1)$ time.

If a node u' is promoted, then the time to perform the insertion is dominated by the time to update the leader fields of the descendants of u' and the time to compute the sizes of the subtrees rooted at the children of u' to update their size fields properly. We can do both tasks at once, in expected $O(s)$ time, by performing a traversal on $T_{u'}$. Lemma 5 bounds the probability that a node is promoted by $O(1/s)$, so this scenario contributes expected constant time to the running time.

The third and final possibility is that u becomes a leader. (Note that we can detect if u is leader or generated a promotion or neither of the above in expected constant time.) We find the predecessor a and successor b of u in expected constant time using fast finger search. As noted, we then can find $\ell(a)$ and $\ell(b)$ in expected $O(s)$ time using a non-standard but still fairly simple tree traversal. Let a' and b' be the children of $\ell(a)$ and $\ell(b)$ that are ancestors of a and b, respectively.

Updating the leader fields of all of the descendants of a' and b' takes expected $O(s)$ time. Updating the size and leader fields of $\ell(a)$, $\ell(b)$, and u takes constant time. Setting the size fields of a' and b' to $|T_{a'}|$ and $|T_{b'}|$ takes constant time, assuming we computed these subtree sizes around during the search for $\ell(a)$ and $\ell(b)$. Alternately, we can compute them in expected $O(s)$ time via subtree traversals. Finally, the nodes whose size fields must be set to null are all descendants of a' or b', and thus they take only expected $O(s)$ time to update. The whole process takes $O(s)$ time (after the rotations on u are complete). Lemma 5 bounds the probability that u is a leader by $O(1/s)$, so this scenario also contributes only expected constant time to the running time. The total running time is thus $O(1)$ in expectation.

Next we consider fast finger deletions. We can detect if u is leader or will generate a demotion or neither of the above in expected constant time.

If u is not a leader and its deletion will cause no demotion, then the operation clearly takes expected constant time. As with insertions, the actual deletion (i.e., rotating u down to a leaf position and pruning it) takes expected $O(1)$ time [SA96].

If u is not a leader but causes a demotion of a node $\ell(u)$, the operation takes expected constant time to update the size fields of $\ell(u)$ and its children, and takes expected $O(s)$ time to update the leader fields of each node in $T_{\ell(u)}$ to the key of the parent of $\ell(u)$. Lemma 5 bounds the probability that u causes a demotion $O(1/s)$, so this scenario also contributes only expected constant time to the running time.

The third scenario is that u is a leader. It takes expected $O(s)$ time to find the deepest node v in the list L of rotated nodes with $|T_v| \geq s$. Computing $|T_w|$ for each child w of v to update w.size takes expected $O(s)$ time. Updating the size and leader fields of all ancestors of v in L takes expected $O(|L|)$ time, which we amortize

against the rotations of u. The same amortization applies to updating the size fields of all descendants of v in L, and the children of such nodes. Thus updating the size fields takes expected $O(s)$ time in the case that u is a leader. It remains to bound the time to update the leader fields. Note that we only update the leader fields of either single nodes or entire subtrees. Furthermore, when we proceed to update a subtree, every node in the subtree initially has its leader field set incorrectly, as per the remarks in the proof of Proposition 6 on page 82. Therefore, updating the leader fields of k nodes takes expected $O(k+|L|)$ time. We amortize the $O(|L|)$ term against the cost to do the rotations, and apply Lemma 2 to prove that $k \leq 2s+1$. Thus the operation takes expected $O(s)$ time (ignoring the costs amortized against the rotations of u) if u is a leader. Lemma 5 bound the probability of this scenario by $O(1/s)$, so this scenario contributes expected $O(1)$ time to the running time, for an overall running time guarantee of expected $O(1)$ time. ■

We also note that if the relative priorities are drawn from an random permutation, then the number of weight s leaders is $O(n/s)$ with high probability, as the following lemma states.

Lemma 8. *Let T be a random treap with relative priorities determined by a random permutation selected uniformly at random. Let n be the number of nodes in T and fix $s \leq n$. Then $|\mathcal{L}_s[T]| = O(n/s)$ with probability $1 - O\left(\exp\{-\frac{2n}{s(s+1)}\}\right)$.*

Proof: Lemma 4 states that a node is a weight s partition leader with probability $O(1/s)$, so we may immediately infer that the expected number of weight s leaders is $O(n/s)$. In fact, since the priorities are fully random, it is not hard to show that the probability of any node being in $\mathcal{L}_s[T]$ is at most $3/(s-1)$.

While this is encouraging, we aim to prove high probability bounds. Our task is complicated by the fact that the events $\{v \in \mathcal{L}_s[T]\}$ are dependent. Let X_v be the indicator random variable for the event $\{v \in \mathcal{L}_s[T]\}$ (i.e., $X_v = 1$ if this event occurs, and zero otherwise). Then $X = \sum_{v \in T} X_v$ equals the size of $\mathcal{L}_s[T]$. We cannot simply use Chernoff bounds (see Theorem 12) on X. Instead, we partition X as $X = \sum_i Y_i$ for dependent Y_i that are each the sum of independent random variables in $\{X_v : v \in T\}$. Lemma 2 on page 90 implies that the random variables in $\{X_u : u \in W\}$ are independent so long as the minimum distance (in the key space) between any two elements of W is at least $s+1$. Relabel the keys $u_1, u_2, \ldots, u_n$, with $u_1 < u_2 < \ldots < u_n$. Define $W_i := \{u_{(i+j(s+1))} : j \in \mathbb{N}\} \cap [1, n]$ for each i such that $1 \leq i \leq s$, and let

$$Y_i := \sum_{u \in W_i} X_u.$$

Applying the Chernoff bound variant of Theorem 12 to each Y_i yields

$$\mathbf{Pr}[Y_i \geq 3\mathbf{E}[Y_i]] \leq \exp\left(-\mathbf{E}[Y_i]\right)$$

(Note that we make no attempt to optimize constants here.) We next apply Theorem 13 to the Y_i's to obtain

$$\mathbf{Pr}[X \geq 3\mathbf{E}[X]] \leq \max_i \left(\exp\left(-\mathbf{E}[Y_i]\right)\right)$$

It is not hard to show that $\mathbf{E}[Y_i] = \frac{2n}{(s+1)^2} + \Theta(1)$, using the following facts.

- For all i, $\mathbf{Pr}[|T_{u_i}| = n] = \frac{1}{n}$.
- For all i such that $s < i < n - s$, and for all k such that $s \leq k < n$, $\mathbf{Pr}[|T_{u_i}| = k] = \frac{2}{(k+1)(k+2)} = 2\left(\frac{1}{k+1} - \frac{1}{k+2}\right)$.
- For all i such that $i \leq s$ or $n - s \leq i \leq n$, and for all k such that $s \leq k < n$, $\mathbf{Pr}[|T_{u_i}| = k] = \lambda\frac{2}{k(k+1)(k+2)} + \frac{1}{k(k+1)} \geq \frac{1}{k(k+1)}$, where $\lambda := \min\{i-1, n-i\}$.

Therefore

$$\mathbf{Pr}[X \geq 3\mathbf{E}[X]] \leq O\left(\exp\left(-\frac{2n}{s(s+1)}\right)\right).$$

In particular

$$\mathbf{Pr}\left[X \geq \frac{9n}{s-1}\right] \leq O\left(\exp\left(-\frac{2n}{s(s+1)}\right)\right).$$

■

There are several Chernoff bounds, though we find the following one particularly convenient.

Theorem 12. *A Chernoff Bound (Theorem 5 of [CL06]). Let $\{X_i : 1 \leq i \leq n\}$ be independent* binary *random variables with $\mathbf{Pr}[X_i = 1] = p_i$ and $\mathbf{Pr}[X_i = 0] = 1 - p_i$. Fix any positive $a_1, \ldots, a_n$ and let $Y := \sum_i a_i X_i$, let $\mu := \mathbf{E}[Y] = \sum_i a_i p_i$, let $\beta := \sum_i a_i^2 p_i$, and let $a := \max_i\{a_i\}$. Then*

$$\mathbf{Pr}[Y \leq \mu - \lambda] \leq \exp\left(-\lambda^2/2\beta\right) \quad (5.1)$$

$$\mathbf{Pr}[Y \geq \mu + \lambda] \leq \exp\left(-\lambda^2/2(\beta + a\lambda/3)\right) \quad (5.2)$$

Siegel proved the following useful theorem (see Result 2.1 and Theorem 3 of [Sie95b]).

Theorem 13. *[Sie95b]. Let $X = X_1 + X_2 + \cdots + X_k$ be the sum of k dependent random variables. Let $a = a_1 + a_2 + \cdots + a_k$ be partitioned so that the Chernoff-Hoeffding estimates for $\mathbf{Pr}[X_i - \mathbf{E}[X_i] \geq a_i]$ are at most C for all $1 \leq i \leq k$. Then $\mathbf{Pr}[X - \mathbf{E}[X] \geq a] \leq C$.*

5.4 Dynamic Ordered Sets & Dictionaries

The *dynamic ordered sets* abstract data type maintains an subset $S \subseteq U$ from an ordered universe U and supports the following operations.

- $\text{lookup}(x)$: determine if $x \in S$.
- $\text{insert}(x)$: insert x into S.
- $\text{delete}(x)$: delete x from S.
- $\text{pred}(x)$: For any $x \in U$, return the predecessor of x, $\max\{y \in S : y < x\}$.

The related *dynamic dictionary* abstract data type also maintains auxiliary data with each element, which can be updated via an insert operation and inspected via a lookup operation. It is not hard to see that the two problems are essentially equivalent, assuming the auxiliary data is a constant number of words (e.g., a pointer to, or a label of, some other object). We will thus focus on the dynamic ordered sets problem, and will consider two classes of inputs.

1. $U = \{0, 1, \ldots, n^{O(1)}\}$ and uses the standard ordering on the integers.
2. U is an arbitrary set with an arbitrary permutation σ inducing a total ordering $<_\sigma$ on U, and a black-box *comparator* f such that for all $x, y \in U$, $f(x, y)$ returns true if $x <_\sigma y$ and false otherwise.

Let $n := |S|$ be the number of elements we are currently storing. In the first case, the van Emde Boas structure [vEBKZ77] described by Mehlhorn and Naher [MN90a] requires only $O(n)$ space and supports lookup and predecessor operations in $O(\log \log n)$ worst-case time and insertions and deletions in expected $O(\log \log n)$ time. The same guarantees can be obtained using the *y-fast tries* of Willard [Wil83, Wil84]. It turns out that this is optimal up to constant factors for any data structure using only $n(\log n)^{O(1)}$ space [PT06], though data structures that use $n^{\Theta(1)}$ space may be slightly faster [BF02]. In the second case, a standard binary search tree such as AVL trees [AVL62] will support all operations in worst case $O(\log(n))$ time while using only linear space. This is optimal by the $\Omega(n \log n)$ sorting lower bound for a universe with a comparator[4]. We will achieve the same

[4]This folklore result is proved by modeling any sorting algorithm as a decision tree T branching on the output of the comparator. The tree must have at least $n!$ leaves corresponding to the $n!$ possible orderings on U, and since the comparator only provides a bit of information per query T must be a binary tree. It follows that its average depth is $\Omega(n \log_2(n))$.

performance guarantees, up to randomization, for uniquely represented implementations of the dynamic ordered set abstract data type, as described in Theorems 14 and 15.

5.4.1 Integer Keys

Theorem 14. *A uniquely represented ordered dictionary on n keys from the domain $U = \{1, 2, \ldots, n^c\}$ for any constant c can support lookup and predecessor in $O(\log\log n)$ worst-case time, and insertion and deletion in $O(\log\log n)$ expected time, while using $O(n)$ data space*[5].

We will prove Theorem 14 by modifying the y-fast trie [Wil83] to make it uniquely represented. First we review the construction of y-fast tries.

y-Fast Tries. Suppose we are given a set $S \subseteq U$ of keys, where $x \in S$ has associated data $d(x)$. We treat each $x \in U$ as a bit string, and store each prefix p of each element $x \in S$ in a dynamic perfect hash table H. Each element x has $d(x)$ stored with it, and each proper prefix p has $\min\{x : x \in S,\ p \text{ is a prefix of } x\}$ and $\max\{x : x \in S,\ p \text{ is a prefix of } x\}$ stored with it. We also maintain all elements in a linked list L, sorted in order. This implementation requires $O(n \log |U|)$ space, and can support insertions and deletions in expected $O(\log |U|)$ time. It can also support predecessor queries in $O(\log\log |U|)$ time as follows. Given x, find the longest prefix p of x such that p is in the hash table H. This can be done via a binary search on the length of p (in bits) in $O(\log\log |U|)$ time. If $p \cdot 1$ is a prefix of x, then $\max\{x' : x' \in S,\ p \cdot 0 \text{ is a prefix of } x'\}$ is the predecessor of x. Otherwise, $p \cdot 0$ is a prefix of x, and $\min\{x' : x' \in S,\ p \cdot 1 \text{ is a prefix of } x'\}$ is the successor of x, which is adjacent to the predecessor of x in L. In either case, we can find the predecessor in $O(1)$ time after finding p.

To reduce the space usage to $O(n)$ and the update time to expected $O(\log\log |U|)$, Willard used *indirection*. This technique involves partitioning S into $\Theta(n/\log |U|)$ groups of size $O(\log |U|)$, and selecting a single representative from each group. The representatives are stored in the structure described above using $O(n)$ space. Each group is stored in a balanced binary search tree, so that updates and predecessor queries within each tree take $O(\log\log |U|)$ time.

We now proceed with the proof of Theorem 14.

[5] See Definition 6 on page 26.

Proof: We use the uniquely represented dynamic perfect hash tables of Section 3.2 instead of conventional dynamic perfect hash tables. It is not hard to see that in this case, the simple version of the data structure (that may use $\Theta(n \log |U|)$ space) is uniquely represented. The set of prefixes and keys stored, along with the values stored with them, is a function of the dictionary contents and the random bits only, and is history independent.

For the indirection, we can use the treap partitioning scheme of Section 5.3 with partition size parameter $\beta = \Theta(\log |U|)$. The partition leaders can be stored as described above, and each group (i.e., each partition set) can be stored in a uniquely represented binary search tree such as may be found in Section 4.5. (The different structures can be have memory dynamically allocated to them as described in Chapter 8.) This will maintain unique representation and assuming there are only $O(n/\log |U|)$ leaders the resulting structure will use $O(n)$ space. By Lemma 8, this assumption holds with very high probability (i.e., $1 - \exp\{-\Omega(n/\log^2 |U|)\}$). To ensure the data structure uses $O(n)$ data space with certainty rather than with very high probability, we can select a permutation on hash functions for the treap partitioning scheme as we do for the dynamic perfect hash table of Section 3.2. As with the dynamic perfect hash table, the cost to reconstruct everything using the lowest index hash function adds only $o(1)$ expected time to the operations. We omit the details.

Finally, we bound the running time. If there are only $O(n/\log |U|)$ leaders, the running times for lookups and predecessor queries are the same as conventional y-fast tries. Consider the insertion or deletion of an element x. If the operation does not change the set of leaders, then we need only find the group that x belongs to (using a predecessor query this takes $O(\log\log |U|)$ time), and insert it into the corresponding uniquely represented binary search tree (in expected $O(\log\log |U|)$ time). If the operation does change the set of leaders, as happens with probability $O(1/\log |U|)$ by Lemma 5 on page 92, it changes the leaders of at most $O(\log |U|)$ keys by Lemma 2. Inserting these keys into the appropriate binary search trees thus costs expected $O(\log |U| \log\log |U|)$, so this part of the operation takes $O(\log\log |U|)$ time in expectation. Additionally, some of these keys may have to be added or removed from the dynamic perfect hash table, along with all of their prefixes. This takes expected $O(\log |U|)$ time per key that is added or removed from the set of leaders. Lemma 6 on page 92 proves that the expected number of keys that are added or removed from the set of leaders is $O(1/\log |U|)$ per update, so this part of the operation takes only $O(1)$ time in expectation. ∎

Using van Emde Boas Structures. We note that the construction described by Mehlhorn and Naher [MN90a] based on the van Emde Boas structure [vEBKZ77, vEB77] can also be converted into a uniquely represented solution to the ordered dictionary problem. This is the approach taken in [BG07]. The basic idea is to use the uniquely represented dynamic perfect hash tables of Section 3.2 instead of conventional hash tables in the algorithm described by Mehlhorn and Naher [MN90a], and to use treap partitioning for indirection (to reduce the space usage from $O(n \log\log |U|)$ to $O(n)$). It is also necessary to dynamically resize the hash tables and devise a means to label the hash tables in a uniquely represented manner for purposes of memory allocation. While this is all certainly possible, y-fast tries appear to offer a slightly simpler solution.

5.4.2 Keys with a Comparator

In the case that the keys come from a general universe with a comparator we use a variant of treaps [SA96]. We assume every element has a unique label that can be used by a hash function.

Theorem 15. *A uniquely represented ordered dictionary on n keys from a totally ordered domain U can support lookup and predecessor in $O(\log n)$ worst-case time, and insertion and deletion in $O(\log n)$ expected time, while using $O(n)$ data space.*

Proof: We store all elements in a treap using a uniquely represented hash table to locate the elements in memory. The space bound is thus an immediate consequence of Theorem 7. We use a random hash function to generate priorities. From Lemma 4.8 of [SA96] we have $\mathbf{Pr}[\mathrm{depth}(x) \geq 1 + 2c \ln n] < 2(n/e)^{-c \ln(c/e)}$. For a set of keys S, we will call a priority function f *good* if the corresponding treap on S using f has depth at most, say, $8 \ln n$, and *bad* otherwise. In the unlikely event that the initial priority function is bad, we would ordinarily just respond by generating a new random priority function and reconstructing the tree with the new priorities. To maintain the property of unique representation, however, we must be careful. As with dynamic perfect hashing (see Section 3.2), on initialization we select a random permutation on a suitable family of priority functions π^{prio}, and iterate through $\{\pi_i^{\text{prio}} \ : \ i = 0, 1, 2, \ldots\}$ until we find the first priority function which is good for the current keys. As with hashing, after deleting a key we will need to reconstruct the treap using each π_i^{prio} in increasing order of i until we find the first one which is good for the current set of keys. However since the probability that

we need reconstruct the treap t times is $O(n^{-\lambda t})$ with $\lambda = 4\ln(4/e) \approx 1.545$, the expected reconstruction cost is $o(1)$. ∎

5.5 B-Treaps: A Uniquely Represented Alternative to B-Trees

B-trees were invented by Bayer and McCreight [BM72], and are often used to organize information on magnetic disk drives so as to minimize disk I/O operations. B-trees are typically analyzed in an *external memory model* of computation.

An External Memory Model. For concreteness, we describe a variant of the parallel disk model of Vitter [Vit06], with one processor and one disk. This model focuses on the performance bottleneck due to disk I/Os, and thus measures performance in terms of them[6]. In it, there are two types of memory: internal and external. Internal memory is modeled as in a RAM, as a large one dimensional array of data items. External memory is modeled as a very large one dimensional array of *blocks* of data items. A block is a sequence of β data items, where β is a parameter of the model called the *block transfer size*. The external memory can read (or write) a single block of data items to (or from) internal memory during a single I/O. Other parameters include the problem size, n, and the internal memory size m, both measured in units of data items. It is typically assumed that $m = \omega(\beta)$, so that any constant number of blocks can be stored in internal memory. Note that we state the performance guarantees of B-treaps in terms of B-treap nodes read or altered instead of the number of blocks read or written. When each B-treap node fits in a block, these measures are equal up to constants.

B-trees layout. B-trees exploit the external memory model by storing at each node v a set of keys $X_v = \{x_1, x_2, \ldots, x_k\}$ in sorted order at each node v, and maintaining pointers $\{p_i : 0 \leq i \leq k\}$ to children of v, such that p_i points to a child u of v storing a set keys between x_i and x_{i+1}. (Here, we define x_0 to be the largest key in some ancestor of v that is less than x_1, if it exists, and otherwise x_0 is a key less than every other key. Similarly, we define x_{k+1} to be the smallest key in some ancestor of v that is greater than x_k, if it exists, and otherwise x_k is a key

[6]In current hardware, disk I/Os are roughly 10^6 times slower than internal memory accesses.

greater than every other key.) Moreover, the subtree rooted at the node p_i points to contains all keys between x_i and x_{i+1}. The main advantage of B-trees is that if the sets of keys stored at a node are always of size at least α, then the depth of the tree is at most $\log_\alpha(n)$, as opposed to the $\log_2(n)$ lower bound for the depth of any binary tree on n nodes. Often, α is set to n^ϵ for some constant $\epsilon > 0$, so that the depth of the tree is constant. More information on B-trees and their variants (e.g., the B^+-tree and the B^*-tree) can be found in almost any data structures textbook (see e.g., [AHU83, CLRS01, Vit06]) or in the survey of Comer [Com79].

We will not attempt to create a uniquely represented variant of a B-tree. Instead we will construct a uniquely represented tree that is a suitable replacement. We call the resulting data structure a *B-treap*, for "bushy-treap[7]." It will support the following operations.

- $\mathrm{insert}(x)$: insert key x into the B-treap.
- $\mathrm{delete}(x)$: delete key x from the B-treap.
- $\mathrm{lookup}(x)$: determine if x is present in the B-treap, and if so, return a pointer to it.

It is easy to associate auxiliary data with the keys, though for simplicity of exposition we will assume there is no auxiliary data being stored. We will prove the following result.

Theorem 16. *There exists a uniquely represented implementation of a bounded B-treap whose memory is statically allocated and stores keys of a fixed size, such that if the B-treap contains n keys and stores $\Theta(\alpha)$ keys in each node, where $\alpha = \Omega\left((\ln(n))^{1/(1-\epsilon)}\right)$ for some $\epsilon > 0$, then* lookup, insert, *and* delete *each touch at most $O(\frac{1}{\epsilon}\log_\alpha(n))$ B-treap nodes in expectation. Furthermore, in this case the B-treap has depth $O(\frac{1}{\epsilon}\log_\alpha(n))$ with high probability, and if $\alpha = O(n^{\frac{1}{2}-\delta})$ for some $\delta > 0$, then with high probability this implementation requires only linear space to store the B-treap.*

We will devote the remainder of this section to proving Theorem 16. The organization, implementation, and analysis of B-treaps are all discussed in the following sections.

[7]According to Comer [Com79], "The origin of 'B-tree' has never been explained by the authors [Bayer and McCreight]. [...] 'balanced,' 'broad,' or 'bushy' might apply."

5.5.1 B-Treaps Organization

Fix a parameter $\alpha > 2$. Our construction will store at most $2\alpha - 1$ keys in any given node. See Figure 5.7 for a depiction of a B-treap. Suppose we wish to store a set of keys U. We first describe how the B-treap is organized, and then discuss how to implement the operations.

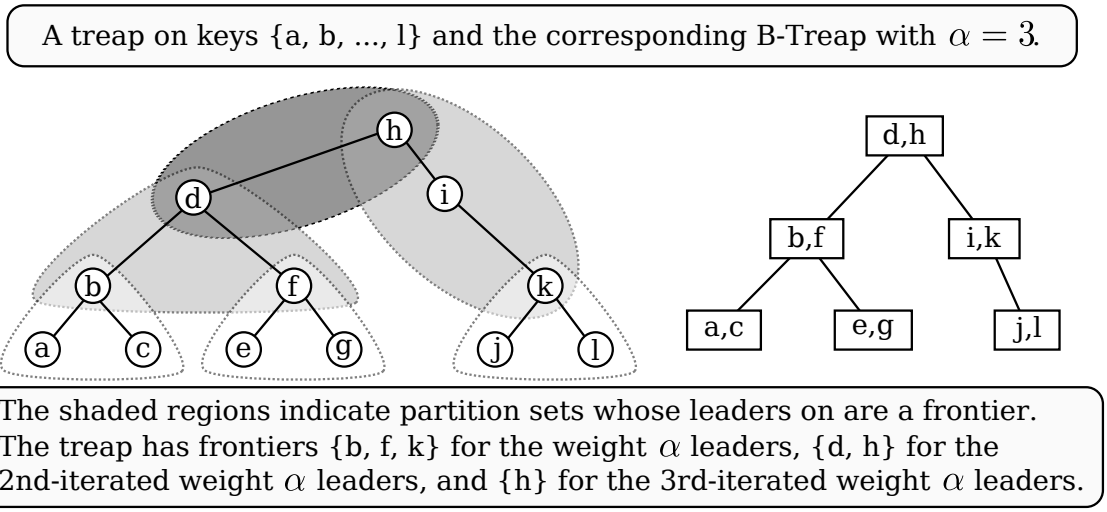


Figure 5.7: A depiction of a B-treap.

First consider a uniquely represented treap T from Section 4.5 storing U. We first describe the organization of the B-treap informally, and then give a formal description. We make use of a refinement of the treap partitioning scheme of Section 5.3. Let $\mathcal{L}_\alpha[T]$ be the *weight α leaders* of T, and let $\ell(u)$ be the leader of u in $\mathcal{L}_\alpha[T]$. We will make use of the following definition.

Definition 10 (Frontier)**.** *For a set of nodes $S \subseteq U$, let the* frontier of S*, denoted $\mathcal{F}[S]$, be the nodes in S that have at least one child not in S.*

We will store the followers in the partition sets whose leaders are in the frontier, one per node of the B-treap. That is, for each $v \in \mathcal{F}[\mathcal{L}_\alpha[T]]$, we create a node for the B-treap and store $\{u : \ell(u) = v\} \setminus \{v\}$ in it. These will be the leaves of the B-treap. We then remove all the followers (i.e., elements of $U \setminus \mathcal{L}_\alpha[T]$), and perform the same procedure on the remaining portion of T. A B-treap leaf storing key set $\{u : \ell(u) = v\} \setminus \{v\}$ has as its parent the node containing key v. We repeat the

process until T has less than α nodes left, in which case all the remaining nodes are assigned to the root node of the B-treap.

Formally, the B-treap $\bar{T}$ is organized as follows, to maintain what we will call the *B-treap organization invariant*. Fix the random bits of the RAM. Given a set of keys $S \subseteq U$, let $\mathrm{treap}(S)$ be the uniquely represented treap on key set S. As before, let $\mathcal{L}_\alpha[T]$ be the weight α leaders of T. We will need the following definitions.

Definition 11 (Iterated Leaders of T)**.** *The i^{th}–iterated weight α leaders of T, denoted $\mathcal{L}^i_\alpha[T]$, are defined inductively as follows.*

- *If $i = 0$, $\mathcal{L}^i_\alpha[T]$ is the set of all keys in T.*
- *If $i \geq 1$ and $|\mathcal{L}^{i-1}_\alpha[T]| > 1$, then $\mathcal{L}^i_\alpha[T] = \mathcal{L}_\alpha[\mathrm{treap}(\mathcal{L}^{i-1}_\alpha[T])]$.*
- *If $i \geq 1$ and $|\mathcal{L}^{i-1}_\alpha[T]| \leq 1$ then $\mathcal{L}^i_\alpha[T] = \emptyset$.*

Furthermore, let $\ell^i(u)$ be the deepest ancestor of u in T that is an i^{th}–iterated weight α leader of T.

Definition 12 (Rank)**.** *The* rank *of a node v in T, denoted $\mathrm{rank}(v)$, is the maximum integer k such that $v \in \mathcal{L}^k_\alpha[T]$. The rank of a tree is the rank of its root.*

Let $k = \mathrm{rank}(T)$ be the rank of the root of T. We will store the keys in $\mathcal{L}^{k-1}_\alpha[T]$ as well as the root of T at the root of the B-treap $\bar{T}$. For $i = k-1$ to 1 in decreasing order, for each node $v \in \mathcal{F}[\mathcal{L}^i_\alpha[T]]$, construct a node $\bar{v}$ in $\bar{T}$ corresponding to v with a key set consisting of the followers of v in the i^{th} level treap partition, excluding v. Formally, the key set of $\bar{v}$ is $\{u \ : \ u \neq v, \ell^i(u) = v, \text{ and } u \in \mathcal{L}^{i-1}_\alpha[T]\}$. Finally, make $\bar{v}$ a child of the node in $\bar{T}$ corresponding to $\ell^{i+1}(v)$. Note that it is possible for a node v to be in two different frontiers, so that $v \in \mathcal{F}[\mathcal{L}^i_\alpha[T]]$ and $v \in \mathcal{F}[\mathcal{L}^j_\alpha[T]]$ for $i < j$. In this case, we create a B-treap node corresponding to each instance in which v is in some frontier. In other words, we create a B-treap node corresponding to (v, i) and another one corresponding to (v, j). In this case, the key set of the former is $\{u \ : \ u \neq v, \ell^i(u) = v, \text{ and } u \in \mathcal{L}^{i-1}_\alpha[T]\}$ and the key set of the latter is $\{u \ : \ u \neq v, \ell^j(u) = v, \text{ and } u \in \mathcal{L}^{j-1}_\alpha[T]\}$. In the future, we will simply refer to the B-treap node corresponding to v, since it can always be inferred from the context which copy is meant.

We have described above how to assign keys to B-treap nodes. In fact, the B-treap will not store merely a set of keys in each node, rather it will store the corresponding treap nodes, with left and right fields for the left and right child of

the current node. The idea is that each node of the B-treap will store a portion of the treap, which will allow us to perform the B-treap operations while touching relatively few (i.e., $O(\log_\alpha n)$) B-treap nodes. In particular, the treap node set stored at each B-treap node will induce at most two connected components in the treap. Finally, if a treap node v stored in B-treap node $\bar{v}$ has a child u that is stored in a different B-treap node $\bar{u}$, we store the label of $\bar{u}$ with v. Thus each treap node should have two more fields left_b-treap_child and right_b-treap_child. (Alternately, we may store two additional bits with each node indicating whether the left and right fields correspond to a "local" treap nodes stored in the same B-treap node or to another B-treap node. Given the B-treap node $\bar{u}$ containing, e.g., v's left child in T, we can find v's left child in T by scanning the treap nodes stored in $\bar{u}$ and selecting the highest priority node less than v.) These B-treap node labels may be generated in several ways. For example, we may use the minimum key in the set of treap nodes stored in the B-treap node. Alternately, we may use the *hash-consing* technique in Section 5.2.2.

By storing small regions (i.e., (nearly) connected subgraphs) of the treap in each B-treap nodes, and storing abstract pointers (in the form of labels) corresponding to treap edges that cross from one region to another, we can search the B-treap for a key by using the underlying treap that it stores. In effect, we have described how to simulate the treap with the B-treap. However, this is not enough information to dynamically maintain the B-treap organization invariant. For that, we must implement several layers of treap partitioning.

5.5.2 Iterated Treap Partitioning

As before, let $k = \mathrm{rank}(T)$. We wish to dynamically maintain k treap partitioning instances simultaneously on the same treap T, where the i^{th} instance stores the weight α partition of $\mathrm{treap}(\mathcal{L}_\alpha^{i-1}[T])$. Surprisingly, this is indeed possible. Consider the basic treap partitioning scheme of Section 5.3.3, particularly the basic size field invariant of Section 5.3.2:

> For all v with $|T_v| < \alpha$, v.size $= |T_v|$. Otherwise v.size $= \infty$.

Note that the size fields of leaders are set to ∞ by convention, because they are there and must be set to some default to ensure unique representation. We could have just as easily set it to be any other value that is distinguished from the subtree sizes. For the iterated treap partitioning scheme, we can thus modify the size fields

to store a pair of integers and modify the invariant as follows. (Below, for a treap T and set of nodes S, we define $T \cap S$ as though T where the set of the nodes it contains.)

The size field invariant (iterated version):
For each treap node v, $v.\mathsf{size} = (\mathrm{rank}(v),\ |T_v \cap \mathcal{L}_\alpha^{\mathrm{rank}(v)}[T]|)$.

The invariant describing what the leader fields should be set to must also be modified. In particular, each node v with $v.\mathsf{size} \in \{i\} \times \mathbb{N}$ should have its leader field set to $\ell^{i+1}(v)$, its deepest ancestor in $\mathcal{L}_\alpha^{i+1}[T]$. (From $v.\mathsf{size} \in \{i\} \times \mathbb{N}$ we may easily infer that $v \in \mathcal{L}_\alpha^i[T]$, so if $i \geq 1$, v is its own j^{th}-iterated leader for all $j \leq i$.) Then v will be stored in the B-treap node corresponding to $\mathrm{node}(v.\mathsf{leader})$.

5.5.3 Implementing B-Treaps

Let $\bar{T}$ be a B-treap storing a treap T. We implement the operations as follows.

Lookup: Given an input key u, start at the root $\bar{r}$ of $\bar{T}$, find the root r of the treap T (by finding the highest priority node stored in $\bar{r}$) and proceed as in a regular treap lookup, jumping from one B-treap node to the next as necessary.

Insertion: To insert a key, create a new node u with that key and search for the leaf position $\mathrm{leaf}(u)$ that u would occupy in the treap T, if it had the lowest priority of any node. Rotate u up to its proper position in T, updating the size fields appropriately during the rotations, so as to maintain the iterated size field invariant. This can be done in constant time per rotation, and ensures that the size fields of all descendants of u are correct.

Suppose u is a leader in some partition – that is, $\mathrm{rank}(u) > 0$. As in the treap partitioning scheme of Section 5.3, find the predecessor predecessor a and successor b of u in T. Proceed up the a-to-u and b-to-u paths looking for $\ell^i(a)$ and $\ell^i(b)$ for all $i \in \{1, 2, \ldots, \mathrm{rank}(u)\}$. (Recall $\ell^i(v)$ is the deepest ancestor of v in $\mathcal{L}_\alpha^i[T]$.) These are easy to find given the information stored in the size fields. If a node v was in $\mathcal{F}[\mathcal{L}_\alpha^i[T]]$ and no longer is, then the corresponding B-treap node must be destroyed. Similarly, if a node v is now in $\mathcal{F}[\mathcal{L}_\alpha^i[T]]$ and was not previously, then a corresponding B-treap node must be created. Furthermore, whenever a node has its leader field changed, we must move it to the B-treap node corresponding to its new leader.

Let a'_i and b'_i be the children of $\ell^i(a)$ and $\ell^i(b)$ that are ancestors of a and b, respectively. For each $i \in 1, 2, \ldots \operatorname{rank}(u)$, update the leader fields of all of the descendants of a'_i in $\mathcal{L}_\alpha^{i-1}[T]$ to $\ell^i(a).\mathsf{key}$ and move them to the B-treap node corresponding to $\ell^i(a)$. Do likewise for the the descendants of b'_i in $\mathcal{L}_\alpha^{i-1}[T]$, with $\ell^i(b)$ in place of $\ell^i(a)$. Make sure to destroy all B-treap nodes that become empty of treap nodes during this process.

That addresses the descendants of u with rank less than $\operatorname{rank}(u)$. The descendants of u with rank equal to $\operatorname{rank}(u)$ will have their fields set correctly unless inserting u causes some node to be promoted. We will deal with that possibility later, and will now focus on setting u's leader field correctly. To do so, we find $\ell^{\operatorname{rank}(u)+1}(u)$. As in the case with treap partitioning, inserting u may cause a "promotion" of some ancestor of u into $\mathcal{L}_\alpha^{\operatorname{rank}(u)+1}[T]$. We can determine this as in treap partitioning. Specifically, set the leader field of u to equal the leader field of its parent node. Find the child of $\operatorname{node}(u.\mathsf{leader})$ that is an ancestor of u, which we denote by u'. Then there is a promotion if and only if $u'.\mathsf{size} \in \mathbb{N} \times \{s-1\}$ and $u' \neq u$. If there is no promotion, then merely increment the second coordinate of $w.\mathsf{size}$ for each w on the path from u to u' (excluding u and including u'), and insert u into the B-treap node corresponding to $\operatorname{node}(u.\mathsf{leader})$.

If u' is promoted, create a new B-treap node $\bar{u'}$ corresponding to it. Traverse $T_{u'} \cap \mathcal{L}_\alpha^{\operatorname{rank}(u)}[T]$ move all nodes therein to $\bar{u'}$, and change their leader fields to $u'.\mathsf{key}$. Note that this correctly updates the leader fields and placements of the descendants of u with rank equal to $\operatorname{rank}(u)$. Additionally, increment the second coordinate of $w.\mathsf{size}$ for each w on the path from u to u' (excluding u and u').

Finally, we must consider the possibility that the promotion of u' into $\mathcal{L}_\alpha^{\operatorname{rank}(u)+1}[T]$ may cause an additional promotion of some node into $\mathcal{L}_\alpha^{\operatorname{rank}(u)+2}[T]$, which may in turn cause an additional promotion of some node into $\mathcal{L}_\alpha^{\operatorname{rank}(u)+3}[T]$, and so on, potentially all the way up to the root. However, the promotion of a node w into $\mathcal{L}_\alpha^k[T]$ is like an insertion of w into the weight α partition on $\mathcal{L}_\alpha^k[T]$ with the additional fact that w is a leaf of $\operatorname{treap}(\mathcal{L}_\alpha^k[T])$. Hence we can handle it as discussed above.

Deletion: Let u be the node to be deleted. First, we deal with potentially cascading demotions. We repeat the following process until the demotion or deletion of the current node w does not cause a demotion of its leader. Initialize w to u.

Proceed up the path P from w to $w_l := \operatorname{node}(w.\mathsf{leader})$, decrementing the

second coordinate of w'.size for each $w' \in P \setminus \{w, w_l\}$. Use the size fields of the children of w_l to determine if $|T_{w_l} \cap \mathcal{L}_\alpha^{\mathrm{rank}(w_l)-1}[T]| = \alpha$ before the deletion or demotion of w.

If $|T_{w_l} \cap \mathcal{L}_\alpha^{\mathrm{rank}(w_l)-1}[T]| = \alpha$ then w_l will be demoted, in which case we create a new B-treap node corresponding to the parent of w_l. Traverse $T_{w_l} \cap \mathcal{L}_\alpha^{\mathrm{rank}(w)}[T]$, move all of the nodes therein to the newly created B-treap node, and set their leader fields to the key of the parent of w_l. Also, decrement the first coordinate of w_l.size (i.e., its rank) and set its second coordinate to $\alpha - 1$.

Set $w = w_l$.

Next, rotate u down to a leaf position, updating the subtree size information appropriately on each rotation. This can be done in constant time per rotation. (Make sure to account for the fact that u will ultimately be deleted from the treap when updating these size fields.)

If initially $\mathrm{rank}(u) = 0$, then just delete u. If initially $\mathrm{rank}(u) \geq 1$, maintain a list L of nodes x such that we rotated on edge $\{x, u\}$ when rotating u down to a leaf position. Within the list, *mark* nodes whose rank changed. (Note that we can determine if $\mathrm{rank}(x)$ changes during the rotation of edge $\{x, u\}$ in constant time, given the nodes x and u and their children.) Delete u from the treap, but retain a temporary copy of its fields. Let $\mathrm{rank}(u)$ denote the initial rank of u. For $i \in \{1, 2, \ldots, \mathrm{rank}(u)\}$ in increasing order, find the deepest element v_i of $\mathcal{L}_\alpha^i[T]$ in L. For each i, if there is no B-treap node corresponding to v_i then create one. For each node $x \in L$, in order of increasing priority, if $x \in \mathcal{L}_\alpha^{i-1}[T]$ is a descendant of v_i and has a child $x' \in \mathcal{L}_\alpha^{i-1}[T]$ with x'.leader $\neq v_i$.key, update the leader field of each node in $T_{x'} \cap \mathcal{L}_\alpha^{i-1}[T]$ to v_i.key. Move all such nodes to the B-treap node corresponding to v_i. Also set x.leader $= v_i$.key and move it to the B-treap node corresponding to v_i. Throughout the whole operation, make sure to destroy all B-treap nodes that are emptied of treap nodes. If the deletion of u did not cause any demotions, then for each ancestor x of $v_{\mathrm{rank}(u)}$ in L set x.leader $= u$.leader. Also, if x has a child y of with $\mathrm{rank}(y) < \mathrm{rank}(x)$ then for each such child y test if y.leader $\neq x$.key. If so, do a traversal of $T_y \cap \mathcal{L}_\alpha^{\mathrm{rank}(y)}[T]$, update the leader field of each node in that subtree to x.key, and move each such node to the B-treap node corresponding to x.

5.5.4 The Analysis of B-Treaps

Proving Correctness

The correctness proofs for the B-treap operations are slightly more complicated than those for the treap partitioning technique of Section 5.3, however they employ almost exactly the same reasoning as may be found in Section 5.3.2 and offer no further insight. The crux of the matter is to show that the field invariants are maintained. We omit the details.

Bounding the Space Usage

We next analyze the space usage of the B-treap, and ultimately show that if the treap priorities are generated via an n-wise independent hash function and $\alpha = O(n^{\frac{1}{2}-\epsilon})$, then the space usage is linear with overwhelming probability (roughly $1 - \exp(n/\alpha^2)$). Towards this end, Proposition 10 bounds the space needed for any given B-treap node at $O(\alpha)$ words, and Lemmas 9 and 10 together bound the number of B-treap nodes at $O(n/\alpha)$. All together, this implies that a B-treap with n keys uses only $O(n)$ words of space with high probability, thus proving the space bounds of Theorem 16.

Proposition 10. *Each B-treap node $\bar{v}$ has at most $2\alpha - 1$ treap nodes stored in it.*

Proof: Assume the operations maintain the B-treap organization invariant. Fix any B-treap node $\bar{v}$. The keys stored at $\bar{v}$ must be a subset of a weight α partition of some subtree of T, and thus, as per the discussion in Section 5.3.1, each B-treap node has at most $2\alpha - 1$ treap nodes stored in it. ■

Lemma 9. *A B-treap $\bar{T}$ on $n \geq \alpha$ keys obtained from the iterated weight α partition of a random treap T uses $O(n + \alpha l)$ words of space, where l is the number of leaves in the B-treap.*

Proof: A *chain* C of T is a connected set of degree two nodes in T such that for all $u, v \in C$, u is an ancestor of v or v is an ancestor of u. Thus, each chain C can be written as $\text{ancestors}(u) \cap \text{descendants}(v)$ for some nodes $u, v \in C$, called the *endpoints* of C. It is not hard to see that any chain C of T has all of its nodes stored in at most $\left\lfloor \frac{|C|}{\alpha} \right\rfloor + 2$ nodes in the B-treap $\bar{T}$. If, for example, $|C| \geq \alpha/2$, then we may amortize the storage required for these $\left\lfloor \frac{|C|}{\alpha} \right\rfloor + 2$ nodes (each of which takes $O(\alpha)$

words of space) against the $|C|$ treap nodes at a rate of $O(1)$ words of space per treap node. We thus *mark* all B-treap nodes that store at least one treap node from any chain of T of length at least $\alpha/2$. As per our previous remarks, the marked nodes take up $O(n)$ words of space.

Next, consider the unmarked nodes. We claim that the number of unmarked nodes is at most $2l$, where l is the number of leaves in the B-treap (either marked or unmarked). To prove this, we first prove that in the B-treap, each unmarked internal node $\bar{v}$ other than the root has at least two children. Since $\bar{v}$ is an internal node, each treap node u stored in it has $\mathrm{rank}(u) \geq 1$, so that $|T_u| \geq \alpha$. Suppose the treap nodes in $\bar{v}$ have rank k. Then each $u \in \mathcal{F}[\mathcal{L}_\alpha^k[T]]$ stored in $\bar{v}$ corresponds to a child of $\bar{v}$ in $\bar{T}$. However, if there were only one such node, then $\bar{v}$ must store a chain of length at least $\alpha - 1$, contradicting the fact that $\bar{v}$ is an unmarked, internal, non-root node. This allows us to bound the number of unmarked nodes as follows. Let m be the number of marked, internal, non-root nodes in $\bar{T}$. Let i be the number of unmarked, internal, non-root nodes in $\bar{T}$. There are thus $m + l + i + 1$ nodes in the B-treap. Each marked, internal, non-root node has degree at least two, each leaf has degree one, and each unmarked, internal, non-root node has degree at least three. Using the well-known facts that in any undirected graph $G = (V, E)$, $\sum_{v \in V} \deg(v) = 2|E|$ and in a tree $|E| = |V| - 1$, we may infer

$$l + 2m + 3i + 1 \leq \sum_{v \in V} \deg(v) = 2|E| = 2(m + l + i)$$

Canceling terms yields $i + 1 \leq l$, so that the total number of unmarked nodes in the B-treap is at most twice the number of leaves l. Since each B-treap node takes $O(\alpha)$ words of space, this implies that the total space requirement for the unmarked nodes is $O(\alpha l)$ ∎

Lemma 10. *Let $\bar{T}$ be a B-treap on $n \geq \alpha$ keys obtained from the iterated weight α partition of a random treap T with relative priorities determined by a random permutation selected uniformly at random. Let l be the number of leaves of $\bar{T}$. Then $l = O(n/\alpha)$ with probability $O\left(\exp\{-\frac{2n}{\alpha(\alpha+1)}\}\right)$.*

Proof: The number of leaves in $\bar{T}$ equals $|\mathcal{F}[\mathcal{L}_\alpha[T]]|$, which is bounded by $|\mathcal{L}_\alpha[T]|$. The result thus follows from Lemma 8 on page 96. ∎

Bounding the Depth

The whole purpose of the B-tree is to reduce the depth of the search tree from, e.g., $2\log_2(n)$ (for red-black trees) or $\sim 1.44\log_2(n)$ (for AVL trees), to $\log_\alpha n$ for a suitable parameter α. As mentioned previously, α is often set to n^ϵ for some constant ϵ, to make the tree height roughly $1/\epsilon$. So it is reasonable to require any proper B-tree alternative to also have height $O(\log_\alpha(n))$. The B-treap does indeed have this property with high probability if α is sufficiently large, though proving this fact involves invoking some subtle inductive probabilistic conditioning.

Theorem 17. *Fix a random treap T on n nodes with relative priorities determined by a random permutation selected uniformly at random. Let $\bar{T}$ be the corresponding B-treap generated from the iterated weight α partitioning of T. If $\alpha = \Omega\left(\ln(n)^{1/(1-\epsilon)}\right)$ for some positive constant ϵ, then $\mathrm{rank}(T) = O\left(\frac{1}{\epsilon}\log_\alpha(n)\right)$ with high probability. Furthermore, since the B-treap depth is bounded by the rank of its root node, these bounds apply to the B-treap depth as well.*

Proof: Let

$$f(m,k) := \textbf{Pr}[\text{A random treap } T \text{ on } m \text{ nodes has } \mathrm{rank}(T) > k]$$

In the definition of f, we assume the treap T has priorities determined by a truly random permutation on the keys. (We do not use a hash function to determine the priorities, to avoid having to deal with collisions. Nevertheless, the probability of a collision on priorities can be made as small as $1/n^c$ for any constant c, so using hash functions does not affect the result.)

First we claim the intuitively natural statement that adding a node to the treap cannot improve the B-treap depth "on average." More precisely, we claim that for any m and k, $f(m,r) \le f(m+1,r)$. In other words, adding a node to the treap results in a distribution over $\mathrm{rank}(T)$ that stochastically dominates (at first order) the previous distribution over $\mathrm{rank}(T)$. Perhaps the easiest way to see this is to fix a treap T on n nodes and consider adding a new node v that is greater than all others. No matter what priority v has, it cannot split any subtree of T. As such, v cannot affect the rank of any nodes that are not ancestors of it. Furthermore, inserting v can only increase the rank of its ancestors, if it changes them at all. Suppose for a contradiction that node u is the deepest ancestor of v whose rank decreased when v was inserted. Suppose u had rank k before inserting v and has rank $k' < k$ after inserting v. Since u's rank decreased, inserting v must have decreased the number of descendants of u of rank exactly $k-1$. Since inserting v can only affect the rank

of its ancestors, this means that some node w that is an ancestor of v and a proper descendant of u must have had its rank changed. Previously, the rank of w was $k-1$. If the rank of w increased to $k'' \geq k$, then w must have higher priority than u after the insertion of v; this contradicts the fact that on any path to the root the ranks are non-decreasing. If the rank of w decreased, this contradicts our choice of u as the deepest ancestor of v whose rank decreased. We conclude that the rank of the root does not decrease as we add v, and hence $f(m,r) \leq f(m+1,r)$.

Having proved $f(m,r) \leq f(m+1,r)$ for any m and k, we proceed by induction on the treap size. Specifically, we will use the following induction hypothesis, for suitable constants c_1, c_2 and c_3. We will not attempt to optimize constants here, as it adds unnecessary clutter to the proof.

$$f\left(\left(\frac{\alpha}{c_2 \ln(n)}\right)^k,\ c_1 k\right) \ \leq\ \frac{(3(\alpha+1))^k}{n^{c_3}} \tag{5.3}$$

For the basis, $k=1$, note that with less than α nodes, $\mathrm{rank}(T) = 1$ with certainty, so if $c_1 \geq 1$ and $c_2 \geq 1/\ln(n)$, $f(\frac{\alpha}{c_2 \ln(n)}, c_1) = 0$.

For the induction step, we assume the induction hypothesis is true for $k-1$ and prove it for k. Consider a treap T on $m = \left(\frac{\alpha}{c_2 \ln(n)}\right)^k$ nodes, with keys $X := \{1, 2, \ldots, m\}$. Let Y be the α nodes of T with the highest priorities. Since the priorities are random, Y is will be distributed uniformly at random on $\{V : V \subset X, |V| = \alpha\}$. Let $Y = \{y_1, y_2, \ldots, y_\alpha\}$, with $y_1 < y_2 < \cdots < y_\alpha$, and let $y_0 = 0, y_{\alpha+1} = m+1$. Define $Z_i := \{z : y_i < z < y_{i+1}\}$ for all $0 \leq i \leq \alpha$. We claim that

$$\mathrm{rank}(T) \leq \max_{0 \leq i \leq \alpha} (\mathrm{rank}(\mathrm{treap}(Z_i))) + 2 \tag{5.4}$$

Let $k := \max_{0 \leq i \leq \alpha} (\mathrm{rank}(\mathrm{treap}(Z_i)))$. Note that the only rank $k+1$ nodes in T must be in Y itself, since all the nodes in any Z_i have rank at most k. Thus T contains at most α nodes of rank $k+1$, and hence the root of T can have rank at most $k+2$.

Next we obtain a high probability bound on $\max_{0 \leq i \leq \alpha} (\mathrm{rank}(\mathrm{treap}(Z_i)))$ using Lemma 11 and the induction hypothesis. Let A_i be the event that $|Z_i| > \left(\frac{\alpha}{c_2 \ln(n)}\right)^{k-1}$, and let $A := \cup_i A_i$. Setting the parameter c in Lemma 11 to $c_2 \frac{m}{m+1}$, we then obtain

$$\mathbf{Pr}[A] \leq \frac{m}{n^{c_2 \frac{m}{m+1}}} \leq \frac{1}{n^{c_2/2-1}} \tag{5.5}$$

Note that if we condition on all the Z_i's being sufficiently small (i.e., the event $\bar{A}$), then the priorities on the nodes of each Z_i are still random, and so we may apply

the induction hypothesis to obtain

$$\forall i: \ \mathbf{Pr}[\mathrm{rank}(\mathrm{treap}(Z_i)) > c_1(k-1)] \ \leq \ \frac{(3(\alpha+1))^{k-1}}{n^{c_3}} \tag{5.6}$$

Let B be the event that there exists an i such that $\mathrm{rank}(\mathrm{treap}(Z_i)) > c_1(k-1)$. Taking another union bound, we conclude

$$\mathbf{Pr}\big[B|\bar{A}\big] \ \leq \ \frac{(3(\alpha+1))^{k-1}(\alpha+1)}{n^{c_3}} \tag{5.7}$$

Equation (5.4) then implies that

$$f\left(\left(\frac{\alpha}{c_2\ln(n)}\right)^k,\ c_1(k-1)+2\right) \ \leq \ \mathbf{Pr}[A\cup B] = \mathbf{Pr}[A] + \frac{\mathbf{Pr}\big[B|\bar{A}\big]}{\mathbf{Pr}\big[\bar{A}\big]} \tag{5.8}$$

Plugging in the bounds from Equations (5.5) and (5.7), and making some additional assumptions (namely, $\frac{1}{n^{c_2/2-1}} \leq 1/2$ and $c_2/2-1 \geq c_3$) we obtain

$$\begin{aligned}
\mathbf{Pr}[A] + \frac{\mathbf{Pr}\big[B|\bar{A}\big]}{\mathbf{Pr}\big[\bar{A}\big]} &\leq \frac{1}{n^{c_2/2-1}} + \frac{(3(\alpha+1))^{k-1}(\alpha+1)}{n^{c_3}} \cdot \frac{1}{1-\frac{1}{n^{c_2/2-1}}} && (5.9)\\
&\leq \frac{1}{n^{c_2/2-1}} + \frac{(3(\alpha+1))^{k-1}(\alpha+1)}{n^{c_3}} \cdot 2 && (5.10)\\
&\leq \frac{(3(\alpha+1))^{k}}{n^{c_3}} && (5.11)
\end{aligned}$$

In going from (5.9) to (5.10) we have used the fact that $\frac{1}{1-x} \leq 1+2x$ for $x \in [0,1/2]$ and the assumption that $\frac{1}{n^{c_2/2-1}} \leq 1/2$. In going from (5.10) to (5.11) we have used the assumption that $c_2/2-1 \geq c_3$.

Note that if $c_1 \geq 2$, then $c_1(k-1)+2 \leq c_1 k$, and thus we completed the induction step by proving that

$$f\left(\left(\frac{\alpha}{c_2\ln(n)}\right)^k,\ c_1\cdot k\right) \ \leq \ \frac{(3(\alpha+1))^k}{n^{c_3}}.$$

Let $d(n) := \frac{\ln(n)}{\ln(\frac{3(\alpha+1)}{c_2\ln(n)})}$. The fact that $f(m,r) \leq f(m+1,r)$ for all m and r, implies that

$$f(n,\ c_1\lceil d(n)\rceil) \leq \frac{(3(\alpha+1))^{\lceil d(n)\rceil}}{n^{c_3}}$$

Assume that α is sufficiently large so that there exists a constant $\epsilon > 0$ such that $\alpha^\epsilon = \frac{3(\alpha+1)}{c_2 \ln(n)}$. In this case, $d(n) = \frac{1}{\epsilon}\log_\alpha(n)$, and $(3(\alpha+1))^{d(n)} \leq \alpha^{2d(n)} = n^{\frac{2}{\epsilon}}$, and so

$$f\left(n,\ c_1\left\lceil\frac{1}{\epsilon}\log_\alpha(n)\right\rceil\right) \leq n^{\frac{2}{\epsilon}-c_3}$$

Suppose we wish to make this probability as small as $1/n^\lambda$, so that $c_3 \leq \lambda - \frac{2}{\epsilon}$. Let us review the assumptions we have made so far.

- $c_1 \geq 2$.
- $c_3 \leq c_2/2 - 1$.
- $1/n^{(c_2/2-1)} \leq 1/2$, so that $c_2 \geq 2 + 2\frac{\ln(2)}{\ln(n)}$.
- $3(\alpha+1) \leq \alpha^2$, so that $\alpha \geq 4$.
- $\alpha^\epsilon = \frac{3(\alpha+1)}{c_2\ln(n)}$, so that $\alpha = (c_2\ln(n)/3)^{1/(1-\epsilon)}$ for some constant $\epsilon > 0$.

To complete the proof, it suffices to set $c_3 = \lambda + 2/\epsilon$, $c_2 = 2(c_3+1)$, and $c_1 = 2$. Then we obtain the following result. If $\alpha \geq \left(\frac{2}{3}(\lambda + \frac{2}{\epsilon} + 1)\ln(n)\right)^{1/(1-\epsilon)}$, then

$$f\left(n,\ 2\left\lceil\frac{1}{\epsilon}\log_\alpha(n)\right\rceil\right) \leq \frac{1}{n^\lambda}$$

For a more concrete example, for any positive λ, if $\alpha \geq \left(\frac{1}{3}(2\lambda+10)\ln(n)\right)^2$ then

$$f\left(n,\ \lceil 4\log_\alpha(n)\rceil + 1\right) \leq \frac{1}{n^\lambda}$$

■

Lemma 11. *Fix $\alpha, m \in \mathbb{N}$ with $\alpha < m$. Let $X = \{1, 2, \ldots, m\}$. Select a subset Y of X uniformly at random from $\{V : V \subset X, |V| = \alpha\}$. Let $Y = \{y_1, y_2, \ldots, y_\alpha\}$, with $y_1 < y_2 < \cdots < y_\alpha$, let $y_0 = 0, y_{\alpha+1} = m+1$, and let $\Delta = \max_i\{y_{i+1} - y_i - 1\}$, where i ranges from zero to α. Then for all c and n,* $\mathbf{Pr}\left[\Delta > c\frac{m+1}{\alpha}\ln(n)\right] \leq \frac{m}{n^c}$.

Proof: Clearly, there are $\binom{m}{\alpha}$ possible choices of Y. Fix any $k \in \mathbb{N}$. If $\Delta > k$, then there must be an interval $[a:b] \subset X$ containing exactly k integers such that for all i, $y_i \notin [a:b]$. For any fixed values of a and b, this can occur in $\binom{m-k}{\alpha}$ different ways. Taking the union bound over all $m-k+1$ choices of $[a:b]$, we obtain

$$\mathbf{Pr}[\Delta > k] \leq \frac{(m-k+1)\cdot\binom{m-k}{\alpha}}{\binom{m}{\alpha}}$$

Using $\binom{n}{k} = \frac{n!}{(n-k)!k!}$ and simplifying yields

$$\begin{aligned}
\mathbf{Pr}[\Delta > k] &\leq (m-k+1)\prod_{i=0}^{\alpha-1}\left(1-\frac{k}{m-i}\right) && (5.12)\\
&\leq (m-k+1)\exp\left(-\sum_{i=0}^{\alpha-1}\frac{k}{m-i}\right) && (5.13)\\
&\leq (m-k+1)\exp\left(-k\int_{x=m-\alpha+1}^{m+1}\frac{1}{x}\,\mathrm{d}x\right) && (5.14)\\
&= (m-k+1)\exp\left(-k\ln\left(\frac{m+1}{m-\alpha+1}\right)\right) && (5.15)\\
&= (m-k+1)\left(\frac{m-\alpha+1}{m+1}\right)^{k} && (5.16)\\
&\leq (m-k+1)\exp\left(-\frac{k\alpha}{m+1}\right) && (5.17)
\end{aligned}$$

Above we have used the fact that $1-x \leq e^{-x}$ for all $x \in \mathbb{R}$. Setting $k = c\frac{m+1}{\alpha}\ln(n)$ and some simple algebra then completes the proof. ∎

Bounding the Running Time

We measure the running time of the B-treap operations in terms of the number of B-treap nodes they inspect or alter. This is in line with the computational model of B-trees, in which the time to access a B-tree node on disk far exceeds the time to subsequently copy its contents into main memory and perform "reasonable" (e.g., linear time) operations on the contents.

By this measure, lookups clearly inspect only $\mathrm{depth}(\bar{T})$ B-treap nodes. Inserting u may require inspecting up to $\mathrm{depth}(\bar{T})$ nodes to find $\mathrm{leaf}(u)$. After that, if P is the rotation path of u from $\mathrm{leaf}(u)$ to its proper location in treap T, then the operation might modify all the B-treap nodes containing treap nodes in P, as well as B-treap nodes corresponding to elements of $F := P \cap (\cup_{i\geq 1}\mathcal{F}[\mathcal{L}_\alpha^i[T]])$. Moreover, it is not hard to see that these are the only B-treap nodes that need to be updated. The number of B-treap nodes storing treap nodes in P is of course bounded by $\mathrm{depth}(\bar{T})$. Bounding $|F|$ is trickier. Note that $\mathbf{E}[|F|] = O(1)$, since $|F| \leq |P|$ is bounded by the number of rotations in P, and $\mathbf{E}[|P|] = O(1)$ in a random treap [SA96]. The same argument holds true for deletions, so we have proven the following result.

Lemma 12. *In a B-treap $\bar{T}$, the lookup operation inspects $\mathrm{depth}(\bar{T})$ nodes, and insertions and deletions inspect or modify $\mathrm{depth}(\bar{T}) + O(1)$ nodes in expectation.*

Combining Lemma 12 with Theorem 17 then proves the running times bounds of Theorem 16.

5.5.5 Empirical Observations

Figure 5.8 on the following page depicts how the B-treap depth grows as a function of its size. These data were obtained from simulations in which the treap priorities were provided by a pseudorandom number generator. For each size, the depth was averaged over ten rounds, however this tended not to matter as the depth was often the same in all ten rounds and would always be within ± 1 of the average. As the figure shows, the depth is empirically bounded by $1.5 \log_\alpha(n)$.

Figure 5.9 depicts the observed distribution of B-treap size, as measured by the number of B-treap nodes, for a B-treap storing 10^5 elements. As with the previous simulations, the treap priorities were provided by a pseudorandom number generator. As we would expect from our analysis in Section 5.5.4, the space usage appears to be tightly concentrated about the mean, which is approximately $1.5\frac{n}{\alpha}$ B-treap nodes in the figures. Since each B-treap node has space for $2\alpha - 1$ treap nodes, these modest experiments suggest that we can typically store n elements in the space required for about $3n$ treap nodes. If this space utilization is judged unacceptably low, there are various ways the it might be improved at the cost of increasing the average number of I/Os per operation. For example, for any $k \in \mathbb{N}$, we may divide each block into k equally sized parts, hash block parts rather than full blocks into external memory, and allocate $\lceil kq/(2\alpha - 1) \rceil$ block parts to a B-treap node with q treap nodes.

5.6 Dynamic Order Maintenance

The *order-maintenance* problem involves creating a data structure that stores a total ordering σ while supporting the following operations.

- $\mathrm{insert}(x, y)$: insert new element y right after x in σ.
- $\mathrm{delete}(x)$: delete element x from σ.

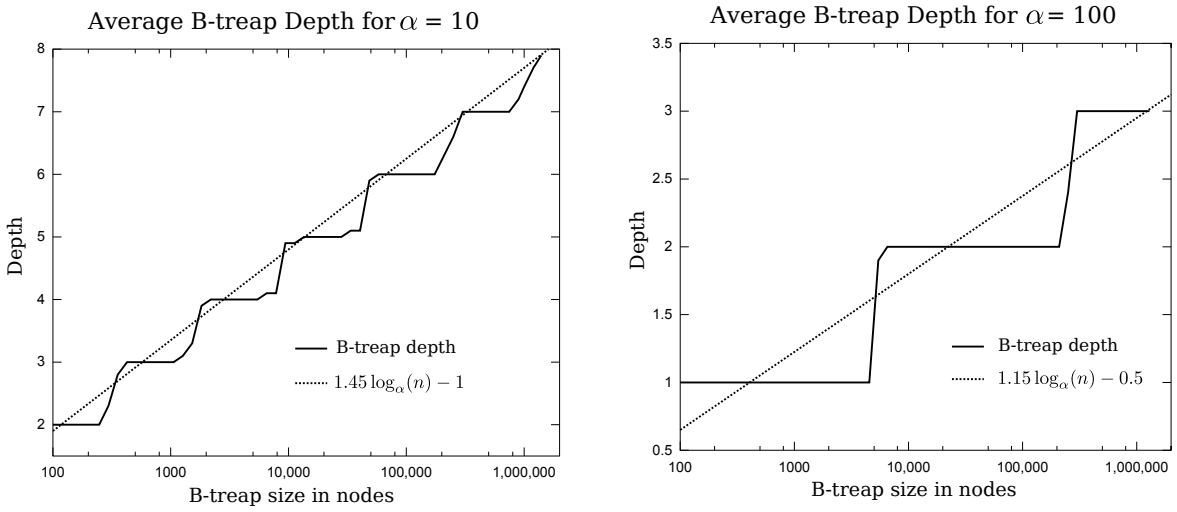


Figure 5.8: B-treap depth as a function of n with $\alpha = 10$ and $\alpha = 100$, averaged over 10 runs. Here, the depth is the number of edges in the longest root to leaf path.

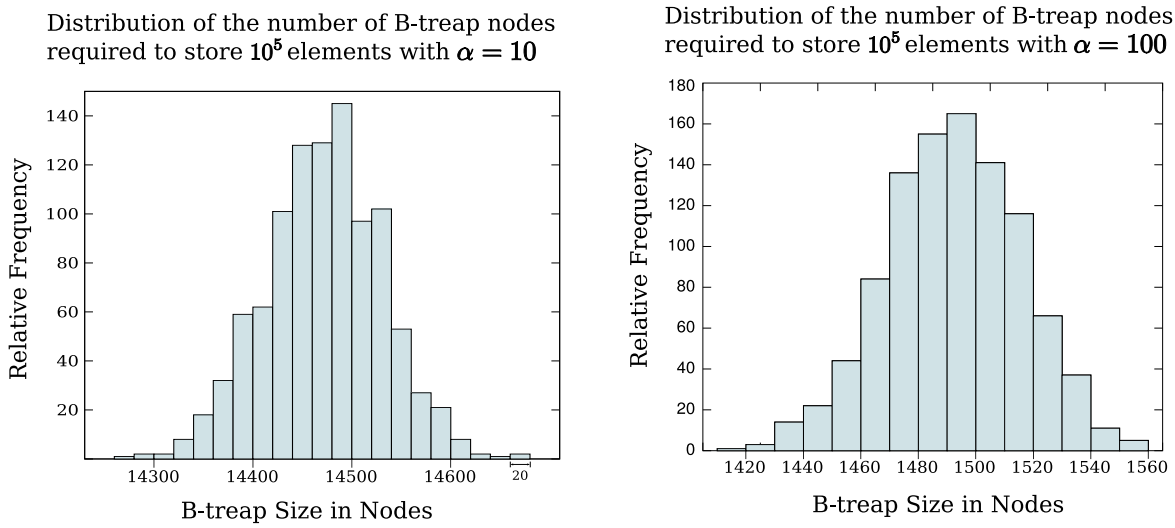


Figure 5.9: The observed distributions on the number of nodes in a B-treap on $n = 10^5$ nodes, with $\alpha = 10$ and $\alpha = 100$, using 1000 samples.

- $\mathrm{compare}(x, y)$: determine if x precedes y in σ.

Let n be the number of elements in σ. Dietz and Sleator [DS87] developed a data structure that supports all three operations in worst case $O(1)$ time. (Note that the inputs are assumed to be pointers to the elements in the data structure.) Bender *et al.* [BCD$^+$02] later developed simpler data structures with the same performance guarantees. Both structures depend significantly on the history and we see no simple modifications to make them uniquely represented. In this section we describe a randomized uniquely represented data structure for order maintenance that takes worst-case $O(1)$ time for compare and expected $O(1)$ time for updates. We assume each element has a unique label $l \in U$ that can be hashed. We can use the labels, for example, to place the elements into our uniquely represented hash table.

Bender *et al.* describe a general technique for the *list-labeling problem* based on a certain class of weight-balanced trees. The list-labeling problem is to maintain a dynamic list (with insertions and deletions) along with a mapping from elements in the list to integer labels in the range $[0, u)$ such that the ordering in the list matches the ordering of the labels. The list-labeling problem can be used to implement order-maintenance by using the integer labels for comparison. We say that the *weight* $w(x)$ of a node x in a tree is the number of its descendants (including itself), and the *weight cost* of an operation is the sum of the weights of all modified nodes in the tree plus the running time for the operation. We then have:

Theorem 18 ([BCD$^+$02])**.** *Any balanced-tree structure with (amortized) weight cost $f(n)$ for insertions, maximum degree d, and depth h yields a strategy for list labeling with (amortized) cost $O(f(n))$ and tags from universe $[0, d^h)$.*

For a binary tree, the idea of the technique is to label each node x with the binary representation of the path from the root to x, where left branches yield 0 and right branches 1. Since internal nodes might have path labels that are prefixes of other nodes' path labels, all paths can be terminated with an additional 1 giving the desired (lexicographic) ordering. It is straightforward to show that Theorem 18 also applies for expected weight costs, and the expected weight cost for updates to a treap is known to be $O(\log n)$ [SA96]. Furthermore our treaps from Section 4.5 are uniquely represented. This yields a uniquely represented data structure for the list-labeling problem that supports $O(\log n)$ expected time updates and $O(1)$ time comparisons with high probability (since labels have $O(\log n)$ bits with high probability).

To make updates $O(1)$ amortized time, previous work used a two level structure in which the top level stores a partition of the elements into $\Theta(n/\log n)$ sets of size $O(\log n)$, such that updates to the top level take $O(\log n)$ time and are relatively rare, and updates to the second level take $O(1)$ time (see e.g., [DS87]). Unfortunately the bottom level technique is highly dependent on history. To achieve unique representation (and history independence) we use two levels based on Theorem 18 and a third level using state transitions with table lookup; this uses our hashing scheme in an interesting way.

Theorem 19. *There exists a uniquely represented order maintenance data structure that supports worst-case $O(1)$ time comparisons and expected $O(1)$ time updates, and uses linear data space.*

Proof: We dynamically resize the data structure by reconstructing it from scratch whenever the number of elements crosses a threshold in $\{\lceil \alpha \cdot 2^k \rceil \ : \ k \in \mathbb{Z}^+\}$, where α is chosen uniformly at random from $[1, 2]$. This allows us to assume we have an upper bound, N, on the current number of elements, n, being stored, such that $N/2 \leq n \leq N$. To bound the expected reconstruction cost for each update, note that the probability of any operation crossing a threshold is $O(1/n)$, constructing the necessary lookup tables (defined below) takes $o(n^\epsilon)$ time for any $\epsilon > 0$, and reinserting all the elements takes expected $O(n)$ time (assuming updates take $O(1)$ expected time), so this adds only $O(1)$ time in expectation to any update.

So assume we know N, and $N/2 \leq n \leq N$. We make use of the *treap partitioning* technique described in Section 5.3. Let $s = \lceil \log N \rceil$ and $s' = \lceil \log \log N \rceil$. We will store all elements in a uniquely represented treap T as described in Section 4.5, using an n-wise independent hash function h to generate priorities. Furthermore, T will be partitioned at weight s as described in Section 5.3.3 and again at weight s'. Let the *frontier* of S, denoted $\mathcal{F}[S]$, be defined as in Definition 10 on page 104. Let $\mathcal{L}_\alpha[T]$ be the weight α leaders of T, as defined in Section 5.3.1 on page 78. It is relatively straightforward to modify the treap partitioning scheme so that the same treap can store both of the above partitions simultaneously. The modification is trivial if each node is provided with a set of size and leader fields for each partition. However, it is not difficult to manage with only one set of size and leader fields if we use the following invariants.

Size fields: For all v that are children of nodes in $\mathcal{F}[\mathcal{L}_s[T]] \cup \mathcal{F}[\mathcal{L}_{s'}[T]]$, $v.\mathsf{size} = |T_v|$. For all other nodes, $v.\mathsf{size} = \mathsf{null}$.

Leader fields: For all $v \in \mathcal{L}_{s'}[T]$, $v.\mathsf{leader}$ is set to the key of the weight s leader of v. All other nodes v have $v.\mathsf{leader}$ set to the key of their weight s' leader.

We will additionally maintain a path field with each node. For an ancestor u of v, we encode the path from u to v as a binary string, with left branches encoded as 0 and right branches encoded as 1 as in a binary trie. A basic implementation sets the path field according to the following invariant.

The basic path field invariant: For all $v \in \mathcal{L}_s[T]$, v.path stores the path from $\text{root}(T)$ to v. For all $v \in \mathcal{L}_{s'}[T] \setminus \mathcal{L}_s[T]$, v.path stores the path from the v's weight s leader to v. For all $v \notin \mathcal{L}_{s'}[T]$, v.path stores the path from the v's weight s' leader to v.

To compare nodes x and y, we first compare their weight s leaders using their path fields. If x and y have the same weight s leader, compare their weight s' leaders, again using their path fields. Finally, if x and y have the same weight s' leader, compare them directly using their path fields. (Note that to achieve worst case $O(1)$ comparisons we must be able to compare two path fields (which are numbers in $[0 : 2^d - 1]$, where d is the depth of the top-level treap) in constant time. Thus, we require that $d = O(w)$, where $w = \Omega(\log N)$ is the word size of our machine. We will ensure the treap has sufficiently small depth as we did in the proof of Theorem 15 on page 101, using π^{prio}. As before, the additional expected cost to do this is $o(1)$.) It is possible to show that this construction yields expected $O(\log\log\log n)$ time updates, and we could add levels to achieve expected $O(\log^* n)$ time updates. To get constant time we use table lookup rather than list labeling for the third level. For this more sophisticated implementation we use the following invariant.

The path field invariant: For all $v \in \mathcal{L}_s[T]$, v.path stores the path from $\text{root}(T)$ to v. For all $v \in \mathcal{L}_{s'}[T] \setminus \mathcal{L}_s[T]$, v.path stores the path from the v's weight s leader to v. For all $v \notin \mathcal{L}_{s'}[T]$, v.path $=$ null.

The idea of the third-level is to maintain the $l \leq 2s' = 2\lceil \log\log N \rceil$ nodes per second level partition in a uniquely represented hash table of size $t = \Theta(\log\log N)$. Specifically, each $v \notin \mathcal{L}_{s'}[T]$ is stored in a hash table corresponding to its weight s' leader. (Thus only nodes that are weight s' followers of nodes in $\mathcal{F}[\mathcal{L}_{s'}[T]]$ are stored in hash tables.) These nodes still have their treap node fields, however they now also have hash table locations associated with them which we can use for comparisons. We use these locations for comparisons as follows. The hash table by itself does not define the ordering among the elements, so we represent each possible ordering among the occupied hash-table locations as a distinct *state*. The

number of possible states is at most $\sum_{x=0}^{t} \binom{t}{x} \cdot x! = \sum_{x=0}^{t} \frac{t!}{(t-x)!} \leq \sum_{x=0}^{t} t^x \leq 2 \cdot t^t$. Since t is $O(\log \log N)$, the number of states is $o(N^\epsilon)$ for all $\epsilon > 0$. We can thus represent the state in a single word of memory which we store with each table. To allocate space for the uniquely represented hash tables we can use another uniquely represented hash table with the partition leader as part of the key. For example, given a hash table H with t slots corresponding to node v, we could create labels for each element of $\{(v.\mathsf{key}, 0), (v.\mathsf{key}, 1), \ldots, (v.\mathsf{key}, t-1)\}$, and allocate space for the i^{th} slot of H using the uniquely represented memory allocator (which is essentially one large uniquely represented hash table) with the label for $(v.\mathsf{key}, i)$.

Each state defines a function $q : [t] \times [t] \to \{<, >, \mathsf{undefined}\}$, where $q(a, b)$ returns whether the element stored at location a comes before or after the element at location b or undefined if either location is unoccupied. This function can be represented as a lookup-table with t^2 two-bit entries. The total space for representing all functions is therefore $o(N^\epsilon (\log \log N)^2)$ bits, which is $o(n^\delta)$ for all $\delta > \epsilon$. To implement the $\mathrm{compare}$ operation between elements that fall in the same table (second-level partition) we find the location of each in the table and use q to compare the locations. If two elements appear in different second-level partitions, we compare their weight s' or weight s partition leaders using their path fields as described above. This takes worst case constant time if we can compare two path fields in worst case constant time, and can find the weight s' and weight s leaders of a given node v in worst case constant time. We have discussed above how to ensure the former condition by guaranteeing that the path fields are $O(\log n)$ bits. The latter condition can be ensured by using a uniquely represented dynamic perfect hash table such as the one from Section 3.2 for memory allocation.

We also need to define state transition tables for updates in a second level partition. The insertion of an element into a hash table can be broken down into a sequence of (possibly zero) swaps, followed by an insertion into an empty slot. Deletion is symmetric. We therefore only need state transitions for insertion into an empty slot, deletion from a slot, and swapping of two slot. Each can be represented as a table with t^2 entries, with one such table per state. For example, when inserting key $\mathbf{k}$ into an empty slot, the transition is specified by the slot y to insert $\mathbf{k}$ into, as well as the slot (if it exists) containing the key that immediately precedes $\mathbf{k}$, among those stored in the relevant hash table. As with the comparison function q, these tables will use $o(n^\delta)$ bits.

The overall scheme for an $\mathrm{insert}(x, y)$ can be outlined as follows. Insert y into the treap partitioning instances as described in Section 5.3 using fast finger insertion. Additionally, create and destroy hash tables corresponding to each node that

is added or removed from $\mathcal{F}[\mathcal{L}_{s'}[T]]$, respectively. Move each node v to the appropriate hash table whenever its weight s' leader changes, and update the state of each hash table using the state transition tables as appropriate. This additional work can be amortized against the normal treap partitioning operations and does not change the running time for the insertion, namely expected $O(1)$ time. Finally, maintain the correct settings of the path fields. This involves changing the path fields of each node in $T_v \cap \mathcal{L}_{s'}[T]$ (respectively, $T_v \cap \mathcal{L}_s[T]$) whenever some $v \in \mathcal{L}_{s'}[T] \setminus \mathcal{L}_s[T]$ (respectively, $v \in \mathcal{L}_s[T]$) has its parent changed during a rotation. Using subtree traversals, this can be done in expected time linear in the number of nodes requiring updates. If the inserted node y causes a change in $\mathcal{L}_s[T]$, the expected time to update the path fields is $O(s)$, since the expected time to perform an insertions in treaps is $O(\log n)$ even when the cost to rotate about an edge $\{u, v\}$ is $\Theta(\max\{|T_u|, |T_v|\})$ [SA96], and $s = \Theta(\log(n))$. By Lemma 5, this occurs with probability at most $O(1/s)$, so this adds at most expected constant time to the operation. The same reasoning applies to the work caused if the inserted node y causes a change in $\mathcal{L}_{s'}[T]$. The overall insertion time is thus $O(1)$ in expectation.

The overall scheme for an $\mathrm{delete}(x)$ is similar. Delete x from the treap partitioning instances as described in Section 5.3 using using fast finger deletion. As with insertions, create and destroy hash tables corresponding to nodes that are added or removed from $\mathcal{F}[\mathcal{L}_{s'}[T]]$, respectively. Move each node v to the appropriate hash table whenever its weight s' leader changes, and update the state of each hash table using the state transition tables as appropriate. Maintain the correct settings of the path fields as discussed for insertions. The running time analysis for deletions is very similar to that for insertions, so we omit it.

Finally we consider the space usage. Each element is stored in a treap node with a constant number of additional fields. So the elements of $\mathcal{L}_{s'}[T]$ take $O(|\mathcal{L}_{s'}[T]|)$ words of space to store. The remaining nodes are stored in hash tables. As we have described the construction, all of these nodes are stored in hash tables of size $t = \Theta(s')$. Lemma 8 on page 96 implies that the number of such hash tables will be $O(n/s')$ with probability $1 - O(\exp\{-\frac{2n}{s'(s'+1)}\})$. Thus the total space usage (aside from the random bits used) will be $O(n)$ with extremely high probability. To make the space usage $O(n)$ with certainty, we may use hash tables that dynamically resize to store the second level partitions of nodes in $\mathcal{F}[\mathcal{L}_{s'}[T]]$. In this case we would require state transition tables and state functions of type $[t'] \times [t'] \to \{<, >, \mathsf{undefined}\}$ for each potential hash table size $t' \leq t$. However this can increase the space to store these tables by at most a factor of $t = \Theta(\log \log N)$, so the space to store all such tables and functions is still $o(n^\delta)$ for all $\delta > 0$. ∎

5.7 Dynamic Ordered Subsets

The *dynamic ordered-subsets* problem is to maintain a total ordering $(L, <)$ and a collection of subsets of L, denoted $\mathcal{S} = \{S_0, \ldots, S_{q-1}\}$, while supporting the order maintenance operations from Section 5.6 on L and the following ordered dictionary operations on each subset in $\mathcal{S}$:

- $\mathrm{insert}(x, S_k)$: insert $x \in L$ into set S_k.
- $\mathrm{delete}(x, S_k)$: delete x from S_k.
- $\mathrm{pred}(x, S_k)$: For $x \in L$, return $\max\{e \in S_k : e < x\}$.

Typically, the number of sets, $q = |\mathcal{S}|$, is assumed to be part of the input during initialization. Alternately, we can support operations $\mathrm{create\text{-}set}(k)$ and $\mathrm{delete\text{-}set}(\mathrm{S_k})$, which creates a new empty subset S_k with integer label k, or delete S_k, respectively.

Dietz [Die89] first describes this problem in the context of fully persistent arrays, and gives a solution yielding $O(\log\log m)$ expected amortized time operations, where $m := |L| + \sum_{i=0}^{q-1} |S_i|$ is the total number of element occurrences in subsets. Mortensen [Mor03a] describes a solution that supports updates to the subsets in expected $O(\log\log m)$ time, and all other operations in $O(\log\log m)$ worst case time. We will describe a uniquely represented data structure for this problem that supports the order maintenance operations on L in expected $O(1)$ time and the dynamic ordered dictionary operations on the subsets in expected $O(\log\log m)$ time. We use a somewhat different approach than Mortensen [Mor03a], and our solution is more self-contained. Furthermore, our results improve on Mortensen's results by supporting insertion into and deletion from L in $O(1)$ instead of $O(\log\log m)$ time.

Theorem 20. *Let $m := |\{(x, k) : x \in S_k\}| + |L|$. There exists a uniquely represented data structure for the ordered subsets problem that uses expected $O(m)$ space, supports all order maintenance operations in expected $O(1)$ time, and all other operations in expected $O(\log\log m)$ time.*

We devote the rest of this section to proving Theorem 20. To construct the data structure, we start with a uniquely represented *order maintenance* data structure on L, which we will denote by $L^{\leq}$ (see Section 5.6). Whenever we are to compare two elements, we simply use $L^{\leq}$.

In the construction of $L^{\leq}$ (see Section 5.6, and also [BG07]) the elements of the order are treap partitioned twice using the technique of Section 5.3, at weight $s_L =$

$\Theta(\log |L|)$ and again at weight $\Theta(\log\log |L|)$. The partition sets at the finer level of granularity with more than one element are then stored in uniquely represented hash tables. In the rest of the exposition we will refer to the treap on all of L as $T(L^{\leq})$. The set of weight s_L partition leaders of $T(L^{\leq})$ is denoted by $\mathcal{L}[T(L^{\leq})]$, and the treap on these leaders by $T(\mathcal{L}[L^{\leq}])$.

The other main structure that we use is a treap $\mathcal{T}$ containing all elements from the set $\hat{L} = \{(x,k) : x \in S_k\} \cup \{(x,0) : x \in \mathcal{L}[T(L^{\leq})]\}$. Treap $\mathcal{T}$ is depicted in Figure 5.10. It is partitioned by weight $s = \Theta(\log m)$ partition leaders. Each of these leaders is labeled with the path from the root to it (0 for left, 1 for right), so that the label of each v is the binary representation of the root to v path. The subtreap of $\mathcal{T}$ on $\mathcal{L}[\mathcal{T}]$ thus forms a trie. We also keep a hash table H that maps path labels to nodes. It is important that only the leaders are labeled since otherwise insertions and deletions would require $O(\log m)$ time. We maintain a pointer from each node of $\mathcal{T}$ to its leader. In addition, we maintain pointers from each $x \in \mathcal{L}[T(L^{\leq})]$ to $(x,0) \in \mathcal{T}$. As usual, all of the pointers that we use are actually abstract pointers, that is, labels used to for memory allocation as discussed in Section 5.2.

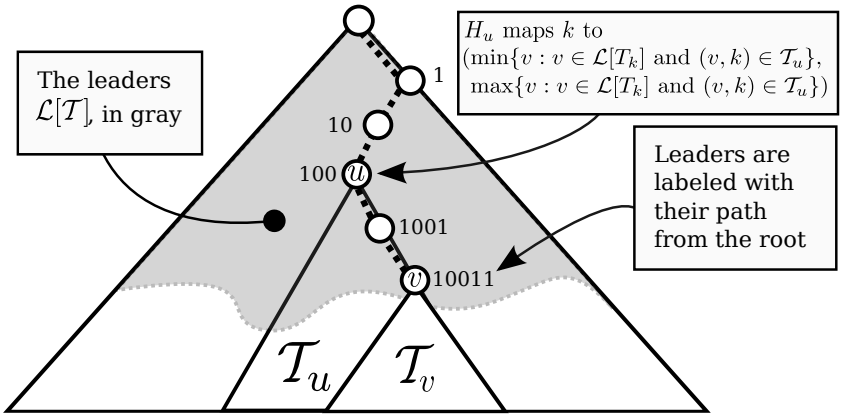


Figure 5.10: Ordered subsets treap $\mathcal{T}$ storing $\hat{L}$.

We store each subset S_k in its own treap T_k, also partitioned by weight $s = \Theta(\log m)$ leaders. When searching for the predecessor in S_k of some element x, we use $\mathcal{T}$ to find the leader ℓ in T_k of the predecessor of x in S_k. Once we have ℓ, the predecessor of x can easily be found by searching in the partition of S_k associated with leader ℓ, which is either $\{\ell\}$ or is stored in an $O(\log m)$-sized subtree of T_k rooted at ℓ. To guide the search for ℓ, we store at each node v of $\mathcal{T}$ the minimum and maximum T_k-leader labels in the subtree rooted at v, if any. Since we have

multiple subsets we need to find predecessors in, we actually store at each v a *mapping* from each subset S_k to the minimum and maximum leader of S_k in the subtree rooted at v. For efficiency, for each leader $v \in \mathcal{T}$ we store a hash table H_v, mapping $k \in [1:q]$ to the tuple $(\min\{u \ : \ u \in \mathcal{L}[T_k] \text{ and } (u,k) \in \mathcal{T}_v\}, \ \max\{u \ : \ u \in \mathcal{L}[T_k] \text{ and } (u,k) \in \mathcal{T}_v\})$, if it exists. Recall $\mathcal{T}_v$ is the subtreap of $\mathcal{T}$ rooted at v. The high-level idea is to use the hash tables H_v to find the right "neighborhood" of $O(\log m)$ elements in T_k which we will have to update (in the event of an update to some S_k), or search (in the event of a predecessor or successor query). Since these neighborhoods are stored as treaps, updating and searching them takes expected $O(\log \log m)$ time. We summarize these definitions, along with some others, in Table 5.2.

$w(x,T)$	number of descendants of node x of treap T
$\ell(x,T)$	the partition leader of x in T
$\mathcal{L}[T]$	weight $s = \Theta(\log m)$ partition leaders of treap T
T_k	treap containing all elements of the ordered subset S_k, $k \in [1:q]$
$T(L^{\leq})$	the treap on L
$T(\mathcal{L}[L^{\leq}])$	the subtreap of $T(L^{\leq})$ on the weight $s = \Theta(\log m)$ leaders of $T(L^{\leq})$
$\hat{L}$	the set $\{(x,k) : x \in S_k\} \cup \{(x,0) : x \in \mathcal{L}[T(L^{\leq})]\}$
$\mathcal{T}$	a treap storing $\hat{L}$
H	hash table mapping label $i \in \{0,1\}^*$ to the key of the leader node in $\mathcal{T}$ with label i
H_v	hash table mapping $k \in [1:q]$ to the tuple (if it exists) $(\min\{u \ : \ u \in \mathcal{L}[T_k] \wedge (u,k) \in \mathcal{T}_v\}, \ \max\{u \ : \ u \in \mathcal{L}[T_k] \wedge (u,k) \in \mathcal{T}_v\})$
I_x	for $x \in L$, a fast ordered dictionary (see Section 5.4) mapping each $k \in \{i : x \in S_i\}$ to (x,k) in $\mathcal{T}$
J_x	for $x \in \mathcal{L}[T(L^{\leq})]$, a treap containing $\{u \in L : \ell(u, T(L^{\leq})) = x \text{ and } \exists\, i : u \in S_i\}$

Table 5.2: Some useful notation and definitions of various structures we maintain for the ordered subsets implementation.

We use Lemma 5 on page 92 to bound the number of changes to the set of partition leaders. It bounds the probability that inserting or deleting an element changes the set of weight s leaders by $O(1/s)$.

Note that each partition set has size at most $O(\log m)$. The treaps T_k, J_x and $\mathcal{T}$, and the dictionaries I_x from Table 5.2 are stored explicitly. We also store the

minimum and maximum element of each $\mathcal{L}[T_k]$ explicitly. We use a total ordering for $\hat{L}$ as follows: $(x,k) < (x',k')$ if $x < x'$ or if $x = x'$ and $k < k'$.

Order Maintenance Insert & Delete Operations: These operations remain largely the same as in the order maintenance structure of [BG07]. We assume that when $x \in L$ is deleted it is not in any set S_k. The main difference is that if the set $\mathcal{L}[T(L^{\leq})]$ changes we will need to update the treaps $\{J_v : v \in \mathcal{L}[T(L^{\leq})]\}$, $\mathcal{T}$, and the tables $\{H_v : v \in \mathcal{L}[\mathcal{T}]\}$ appropriately.

Note that we can easily update H_v in time linear in $|\mathcal{T}_v|$ using in-order traversal of $\mathcal{T}_v$, assuming we can test if x is in $\mathcal{L}[T_k]$ in $O(1)$ time. To accomplish this, for each k we can store $\mathcal{L}[T_k]$ in a hash table. Thus using Theorem 6 on page 59 we can see that all necessary updates to $\{H_v : v \in \mathcal{T}\}$ take expected $O(\log m)$ time. Clearly, updating $\mathcal{T}$ itself requires only expected $O(\log m)$ time. Finally, we bound the time to update the treaps J_v by the total cost to update $T(\mathcal{L}[L^{\leq}])$ if the rotation of subtrees of size k costs $k + \log m$, which is $O(\log m)$ by Theorem 6. This bound holds because $|J_v| = O(\log m)$ for any v, and any tree rotation on $T(L^{\leq})$ causes at most $3s$ elements of $T(L^{\leq})$ to change their weight s leader. Therefore only $O(\log m)$ elements need to be added or deleted from the treaps $\{J_v : v \in T(\mathcal{L}[L^{\leq}])\}$, and we can batch these updates in such a way that each takes expected amortized $O(1)$ time. However, we need only make these updates if $\mathcal{L}[T(L^{\leq})]$ changes, which by Lemma 5 occurs with probability $O(1/\log m)$. Hence the expected overall cost is $O(1)$.

Predecessor & Successor: Suppose we wish to find the predecessor of x in S_k. (Finding the successor is analogous.) Let $\mathrm{pred}(z,T)$ be the predecessor of z in the set or treap T, and let $\mathrm{succ}(z,T)$ be the successor of z in T. If $x \in S_k$ we can test this in expected $O(\log\log m)$ time using I_x. So suppose $x \notin S_k$. We will first find the predecessor $\mathrm{pred}((x,k),\mathcal{T})$ of (x,k) in $\mathcal{T}$ as follows. (We can handle the case that $\mathrm{pred}((x,k),\mathcal{T})$ does not exist by adding a special element to L that is smaller than all other elements and is considered to be part of $\mathcal{L}[T(L^{\leq})]$). First search I_x for the predecessor k_2 of k in $\{i : x \in S_i\}$ in $O(\log\log m)$ time. If k_2 exists, then $\mathrm{pred}((x,k),\mathcal{T}) = (x,k_2)$. Otherwise, let y be the leader of x in $T(L^{\leq})$, and let y' be the predecessor of y in $\mathcal{L}[T(L^{\leq})]$. Then either $\mathrm{pred}((x,k),\mathcal{T}) \in \{(y',0),(y,0)\}$ or else $\mathrm{pred}((x,k),\mathcal{T}) = (z,k_3)$, where $z = \max\{u : u < x \text{ and } u \in J_y \cup J_{y'}\}$ and $k_3 = \max\{i : z \in S_i\}$. Thus we can find $\mathrm{pred}((x,k),\mathcal{T})$ in expected $O(\log\log m)$ time using fast finger search for y', treap search on the $O(\log m)$ sized treaps in $\{J_v : v \in \mathcal{L}[T(L^{\leq})]\}$, and the fast dictionaries $\{I_x : x \in L\}$.

We will next find the predecessor $\mathrm{pred}(x, \mathcal{L}[T_k])$ or the successor $\mathrm{succ}(x, \mathcal{L}[T_k])$ of x in $\mathcal{L}[T_k]$. Note that if we have $\mathrm{pred}(x, \mathcal{L}[T_k])$ we can find $\mathrm{succ}(x, \mathcal{L}[T_k])$ quickly via fast finger search, and vice-versa. Note that if $\mathrm{pred}((x,k), \mathcal{T}) \in \mathcal{L}[T_k] \times \{k\}$, then $\mathrm{pred}((x,k), \mathcal{T}) = \mathrm{pred}(x, \mathcal{L}[T_k])$ and we can test this in constant time if we store the subtree sizes in the treaps T_k. So assume $\mathrm{pred}((x,k), \mathcal{T}) \notin \mathcal{L}[T_k] \times \{k\}$. We start from $\mathrm{pred}((x,k), \mathcal{T})$ and consider its leader $\ell(\mathrm{pred}((x,k), \mathcal{T}))$ in $\mathcal{T}$. We first binary search on the path P from the root of $\mathcal{T}$ to $\ell(\mathrm{pred}((x,k), \mathcal{T}))$ for the deepest node u' such that $\mathcal{T}_{u'}$ contains at least one node from $\mathcal{L}[T_k] \times \{k\}$. (In particular, this means that this subtreap also contains a node (u,k) where u is either the predecessor or successor of x in $\mathcal{L}[T_k]$. If no node on the path has this property, then S_k is empty.) The binary search is performed on the length of the prefix of the label of $\ell(\mathrm{pred}((x,k), \mathcal{T}))$. Given a prefix α, we look up the node p with label α using H, and test whether $\mathcal{T}_p$ contains at least one node from $\mathcal{L}[T_k] \times \{k\}$ using H_p. If so, we increase the prefix length. Otherwise we decrease it.

Next use $H_{u'}$ to obtain $u_{\min} = \min\{u : u \in \mathcal{L}[T_k] \text{ and } (u,k) \in \mathcal{T}_{u'}\}$ and $u_{\max} = \max\{u : u \in \mathcal{L}[T_k] \text{ and } (u,k) \in \mathcal{T}_{u'}\}$. Note that either (1) $u_{\max} < x$, or (2) $u_{\min} > x$, or (3) $u_{\min} < x < u_{\max}$.

In the first case, we claim that $u_{\max}$ is the predecessor of x in $\mathcal{L}[T_k]$. To prove this, note that the predecessor of x in $\mathcal{L}[T_k]$ is the predecessor of $\mathrm{pred}((x,k), \mathcal{T})$ in $\mathcal{L}[T_k]$ under the extended ordering for $\hat{L}$. Now suppose for a contradiction that $z \neq u_{\max}$ is the predecessor of $\mathrm{pred}((x,k), \mathcal{T})$ in $\mathcal{L}[T_k]$. Thus $u_{\max} < z \leq \mathrm{pred}((x,k), \mathcal{T})$. Clearly, $z \notin \mathcal{T}_{u'}$, for otherwise we obtain a contradiction from the definition of $u_{\max}$. However, it is easy to see that every node that is not in $\mathcal{T}_{u'}$ is either less than $u_{\max}$ or greater than $\mathrm{pred}((x,k), \mathcal{T})$, contradicting the assumption that $u_{\max} < z \leq \mathrm{pred}((x,k), \mathcal{T})$. In the second case, we claim that $u_{\min}$ is the successor of x in $\mathcal{L}[T_k]$. The proof is analogous to the first case, and we omit it.

Thus, in the first two cases we are done. Consider the third case. We first prove that $u' = \ell(\mathrm{pred}((x,k), \mathcal{T}))$ in this case. Note that $u_{\min} < x < u_{\max}$ implies $u_{\min} \leq \mathrm{pred}((x,k), \mathcal{T}) < u_{\max}$. Since there is an element of $\mathcal{L}[T_k] \times \{k\}$ on either side of $\mathrm{pred}((x,k), \mathcal{T})$ in $\mathcal{T}_{u'}$, the child u'_c of u' that is an ancestor of $\mathrm{pred}((x,k), \mathcal{T})$ must have an element of $\mathcal{L}[T_k] \times \{k\}$ in its subtree. From the definition of u', we may infer that u'_c was not on the $\ell(\mathrm{pred}((x,k), \mathcal{T}))$ to root path in $\mathcal{T}$. Thus u' is the deepest node on this path, or equivalently $u' = \ell(\mathrm{pred}((x,k), \mathcal{T}))$. Given that $u' = \ell(\mathrm{pred}((x,k), \mathcal{T}))$ and $u_{\min} \leq \mathrm{pred}((x,k), \mathcal{T}) < u_{\max}$, we next prove that at least one element of $\{u_{\min}, u_{\max}\}$ is within distance $s = \Theta(\log m)$ of $\mathrm{pred}((x,k), \mathcal{T})$ in $\mathcal{T}$. Note that because $u'_c \notin \mathcal{L}[\mathcal{T}]$, its subtree $\mathcal{T}_{u'_c}$ has size at most s. If u'_c is the left child of u', then $u_{\min}$ is in this subtree as is $\mathrm{pred}((x,k), \mathcal{T})$. The distance between

$u_{\min}$ and $\text{pred}((x,k),\mathcal{T})$ is thus at most $s = \Theta(\log m)$ in $\mathcal{T}$. If u'_c is the right child of u', then we may argue analogously that $u_{\max}$ and $\text{pred}((x,k),\mathcal{T})$ are both in $\mathcal{T}_{u'_c}$ and thus $u_{\max}$ is within distance s of $\text{pred}((x,k),\mathcal{T})$.

Once we have obtained a node (u_{near},k) in $\mathcal{T}$ such that $u_{\text{near}} \in \mathcal{L}[T_k]$ and (u_{near},k) is within distance $d = O(\log m)$ of $\text{pred}((x,k),\mathcal{T})$ in $\mathcal{T}$, we find the predecessor $\text{pred}(x,\mathcal{L}[T_k])$ and successor $\text{succ}(x,\mathcal{L}[T_k])$ of x in $\mathcal{L}[T_k]$ via fast finger search on T_k using u_{near}. Note that the distance between u_{near} and the nearest of $\{\text{pred}(x,\mathcal{L}[T_k]),\text{succ}(x,\mathcal{L}[T_k])\}$ is at most d. This is because if $u_{\text{near}} \le \text{pred}(x,\mathcal{L}[T_k])$, then every element e of $\mathcal{L}[T_k]$ between u_{near} and $\text{pred}(x,\mathcal{L}[T_k])$ must have a corresponding node $(e,k) \in \mathcal{T}_{u'_c}$ between (u_{near},k) and $\text{pred}((x,k),\mathcal{T})$, and there are at most d such nodes. Similarly, if $u_{\text{near}} \ge \text{succ}(x,\mathcal{L}[T_k])$, then every element e of $\mathcal{L}[T_k]$ between $\text{succ}(x,\mathcal{L}[T_k])$ and u_{near} must have a corresponding node $(e,k) \in \mathcal{T}_{u'_c}$ between $\text{pred}((x,k),\mathcal{T})$ and (u_{near},k), and there are at most d such nodes. Note that finding the predecessor and successor of x in T_k given a handle to a node in T_k at distance d from x takes $O(\log(d))$ time in expectation [SA96]. In this case, $d = O(\log m)$ so this step takes $O(\log\log m)$ time in expectation.

Once we have found $\text{pred}(x,\mathcal{L}[T_k])$ and $\text{succ}(x,\mathcal{L}[T_k])$, the predecessor and successor of x in $\mathcal{L}[T_k]$, we simply search their associated partitions of S_k for the predecessor of x in S_k. These partitions are both of $O(\log m)$ size and can be searched in expected $O(\log\log m)$ time. The total time to find the predecessor is thus $O(\log\log m)$ in expectation.

Ordered Subsets Insert and Delete: $\text{delete}(x,S_k)$ is analogous to $\text{insert}(x,S_k)$, hence we focus on $\text{insert}(x,S_k)$. Suppose we wish to add x to S_k. First, if x is not currently in any sets $\{S_i : i \in [1:q]\}$, then find the leader of x in $T(L^{\le})$, say y, and insert x into J_y in expected $O(\log\log m)$ time. Next, insert x into T_k as follows. Find the predecessor w of x in S_k, then insert x into T_k in expected $O(1)$ time starting from w to speed up the insertion.

Find the predecessor w' of (x,k) in $\mathcal{T}$ as in the predecessor operation, and insert (x,k) into $\mathcal{T}$ using w' as a starting point. If neither $\mathcal{L}[T_k]$ nor $\mathcal{L}[\mathcal{T}]$ changes, then no modifications to $\{H_v : v \in \mathcal{L}[\mathcal{T}]\}$ need to be made. If $\mathcal{L}[T_k]$ does not change but $\mathcal{L}[\mathcal{T}]$ does, as happens with probability $O(1/\log m)$, we can update $\mathcal{T}$ and $\{H_v : v \in \mathcal{L}[\mathcal{T}]\}$ appropriately in expected $O(\log m)$ time by taking time linear in the size of the subtree to do rotations. If $\mathcal{L}[T_k]$ changes, we must be careful when updating $\{H_v : v \in \mathcal{L}[\mathcal{T}]\}$. Let $\mathcal{L}[T_k]$ and $\mathcal{L}[T_k]'$ be the leaders of T_k immediately before and after the addition of x to S_k, and let $\Delta_k := (\mathcal{L}[T_k] - \mathcal{L}[T_k]') \cup (\mathcal{L}[T_k]' - \mathcal{L}[T_k])$.

Then we must update $\{H_v : v \in \mathcal{L}[\mathcal{T}]\}$ appropriately for all nodes $v \in \mathcal{L}[\mathcal{T}]$ that are descendants of (x, k) as before, but must also update H_v for any node $v \in \mathcal{L}[\mathcal{T}]$ that is an ancestor of some node in $\{(u, k) : u \in \Delta_k\}$. It is not hard to see that these latter updates can be done in expected $O(|\Delta_k| \log m)$ time. From Lemma 6 we know that $\mathbf{E}[|\Delta_k|] = O(1/\log m)$. Since the randomness for T_k is independent of the randomness used for $\mathcal{T}$, these expectations multiply, for a total expected time of $O(1)$ for this part of the operation. Finally, insert k into I_x in expected $O(\log \log m)$ time, with a pointer to (x, k) in $\mathcal{T}$.

Space Analysis. We now prove that our ordered subsets data structure uses $O(m)$ space in expectation. Table 5.2 lists the objects that that the data structure uses. The order maintenance structure uses $O(|L|)$ space. The hash table H, treap $\mathcal{T}$, the collection of treaps $\{T_k : k \in [1 : q]\}$, and the collection fast dictionaries $\{I_x : x \in L\}$ each use only $O(m)$ space. The collection $\{J_x : x \in \mathcal{L}[T(L^{\leq})]\}$ uses only $O(|L|)$ space. That leaves the space required by $\{H_v : v \in \mathcal{L}[\mathcal{T}]\}$. We claim these hash tables use $O(m)$ space in expectation. To prove this, let $X_{u,k}$ be a random variable denoting the number of hash tables in $\{H_v : v \in \mathcal{L}[\mathcal{T}]\}$ that map k to a tuple of the form $(u, *)$ or $(*, u)$, where $*$ denotes a wildcard that matches all nodes or null. (If H_v maps k the record (u, u), we may count that record as contributing two to $X_{u,k}$). The space required for $\{H_v : v \in \mathcal{L}[\mathcal{T}]\}$ is then linear in the total number of entries of all hash tables, which is $\sum_{u \in L} \sum_{k=0}^{q-1} X_{u,k}$. Clearly, if $u \notin S_k$ then $X_{u,k} = 0$. On the other hand, we claim that if $x \in S_k$ then $\mathbf{E}[X_{u,k}] = O(1)$, in which case $\mathbf{E}\left[\sum_{u \in L} \sum_{k=0}^{q-1} X_{u,k}\right] = O(m)$ by linearity of expectation. Assume $u \in S_k$. Note that if $u \notin \mathcal{L}[T_k]$, then $X_{u,k} = 0$, and $\mathbf{Pr}[u \in \mathcal{L}[T_k]] = O(1/\log m)$ (the probability of any node being a weight s leader is $O(1/s)$, which is an easy corollary of Theorem 6). Furthermore

$$\begin{aligned} \mathbf{E}[X_{u,k} \mid u \in \mathcal{L}[T_k]] &\leq \mathbf{E}[\text{depth of } (u, k) \text{ in } \mathcal{T}] \\ &= O(\log m). \end{aligned}$$

It follows that

$$\begin{aligned} \mathbf{E}[X_{u,k}] &= \mathbf{E}[X_{u,k} \mid u \in \mathcal{L}[T_k]] \cdot \mathbf{Pr}[u \in \mathcal{L}[T_k]] \\ &= O(\log m \cdot \tfrac{1}{\log m}) \\ &= O(1). \end{aligned}$$

5.8 Range Trees

Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of points in $\mathbb{R}^d$. The well studied *orthogonal range reporting* problem is to maintain a data structure for P while supporting queries which given an axis aligned box B in $\mathbb{R}^d$ returns the points $P \cap B$. The dynamic version allows for the insertion and deletion of points. Chazelle and Guibas [CG86] showed how to solve the two dimensional dynamic problem in $O(\log n \log\log n)$ update time and $O(\log n \log\log n + k)$ query time, where k is the size of the output. Their approach used fractional cascading [CG86, MN90b]. More recently Mortensen [Mor06] showed how to solve it in $O(\log n)$ update time and $O(\log n + k)$ query time using a sophisticated application of Fredman and Willard's q-heaps [FW94]. All of these techniques can be generalized to higher dimensions at the cost of replacing the first $\log n$ term with a $\log^{d-1} n$ term [dBvKOS97].

Here we present a uniquely represented solution to the problem. It matches the bounds of the Chazelle and Guibas version, except our bounds are in expectation instead of worst-case. Our solution does not use fractional cascading and is instead based on ordered subsets. Our solution is simple and avoids any explicit discussion of weight balanced trees (the required properties fall directly out of known properties of treaps). One might possibly derive a uniquely represented version based on fractional cascading, but making dynamic fractional cascading uniquely represented would require significant work[8] and is unlikely to improve the bounds.

Theorem 21. *Let P be a set of n points in $\mathbb{R}^d$. There exists a uniquely represented data structure for the orthogonal range query problem that uses expected $O(n \log^{d-1} n)$ space and $O(d \log n)$ random bits, supports point insertions and deletions in expected $O(\log^{d-1} n \cdot \log\log n)$ time, and queries in expected $O(\log^{d-1} n \cdot \log\log n + k)$ time, where k is the size of the output.*

If $d = 1$, simply use the dynamic ordered dictionaries solution [BG07] and have each element store an abstract pointer to (i.e., the label of) its successor for fast reporting. For simplicity we first describe the two dimensional case. The remaining cases with $d \geq 3$ can be implemented using standard techniques [dBvKOS97] if treaps are used for the underlying hierarchical decomposition trees, as we describe below.

We will assume that the points have distinct coordinate values; therefore, if (x_1, x_2),$(y_1, y_2) \in P$, then $x_i \neq y_i$ for all i. (There are various ways to remove

[8] A variant of Sen's approach [Sen95] might work.

this assumption, for example the composite-numbers scheme or symbolic perturbations [dBvKOS97].) We store P in a random treap T using the ordering on the first coordinate as our binary search tree ordering. We additionally store P in a second random treap T' using the ordering on the second coordinate as our binary search tree ordering, and also store P in an ordered subsets instance D using this same ordering. We cross link these and use T' to find the position of any point we are given in D. The subsets of D are $\{T_v : v \in T\}$, where T_v is the subtree of T rooted at v. We assign each T_v a unique integer label k using the coordinates of v, so that T_v is S_k in D. The structure is uniquely represented as long as all of its components (the treap and ordered subsets) are uniquely represented.

To insert a point p, we first insert it by the second coordinate in T' and using the predecessor of p in T' insert a new element into the ordered subsets instance D. This takes $O(\log n)$ expected time. We then insert p into T in the usual way using its x coordinate. That is, search for where p would be located in T were it a leaf, then rotate it up to its proper position given its priority. As we rotate it up, we can reconstruct the ordered subset for a node v from scratch in time $O(|T_v| \log \log n)$. Using Theorem 6 on page 59, the overall time is $O(\log n \cdot \log \log n)$ in expectation. Finally, we must insert p into the subsets $\{T_v : v \in T \text{ and } v \text{ is an ancestor of } p\}$. This requires expected $O(\log \log n)$ time per ancestor, and there are only $O(\log n)$ of them in expectation. Since these expectations are computed over independent random bits, they multiply, for an overall time bound of $O(\log n \cdot \log \log n)$ in expectation. Deletion is similar.

To answer a query $(p, q) \in \mathbb{R}^2 \times \mathbb{R}^2$, where $p = (p_1, p_2)$ is the lower left and $q = (q_1, q_2)$ is the upper right corner of the box B in question, we first search for the predecessor p' of p and the successor q' of q in T (i.e., with respect to the first coordinate). Please refer to Figure 5.11. We also find the predecessor p'' of p and successor q'' of q in T' (i.e., with respect to the second coordinate). Let w be the least common ancestor of p' and q' in T, and let $A_{p'}$ and $A_{q'}$ be the paths from p' and q' (inclusive) to w (exclusive), respectively. Let V be the union of right children of nodes in $A_{p'}$ and left children of nodes in $A_{q'}$, and let $\mathcal{S} = \{T_v : v \in V\}$. It is not hard to see that $|V| = O(\log n)$ in expectation, that the sets in $\mathcal{S}$ are disjoint, and that all points in B are either in $W := A_{p'} \cup \{w\} \cup A_{q'}$ or in $\cup_{S \in \mathcal{S}} S$. Compute W's contribution to the answer, $W \cap B$, in $O(|W|)$ time by testing each point in turn. Since $\mathbf{E}[|W|] = O(\log n)$, this requires $O(\log n)$ time in expectation. For each subset $S \in \mathcal{S}$, find $S \cap B$ by searching for the successor of p'' in S, and doing an in-order traversal of the treap in D storing S until reaching a point larger than q''. This takes $O(\log \log n + |S \cap B|)$ time in expectation for each $S \in \mathcal{S}$, for a total of

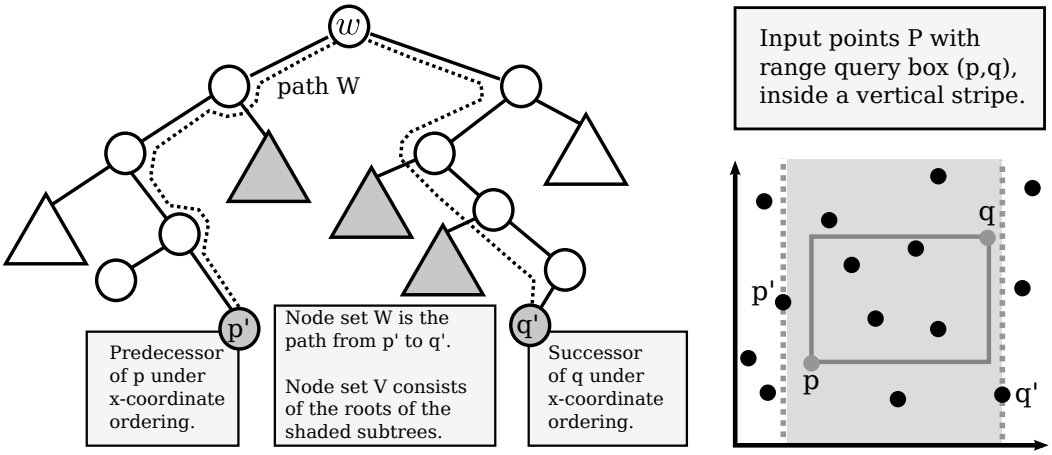


Figure 5.11: Answering 2-D range query (p, q). Path W together with the shaded subtrees contain the nodes in the vertical stripe between p' and q'.

$O(\log n \cdot \log\log n + k)$ expected time.

To bound the space required, note that each point p is stored in $\text{depth}(p) + 1$ subsets in the ordered subsets instance D, where $\text{depth}(p)$ is the depth of p in treap T. Since $\mathbf{E}[\text{depth}(p)] = O(\log n)$, we infer that $\mathbf{E}[\sum_v |T_v|] = O(n \log n)$, and so D uses $O(n \log n)$ space in expectation. Since the storage space is dominated by the requirements of D, this is also the space requirement for the whole data structure.

Extending to Higher Dimensions. We show how to support orthogonal range queries for $d \geq 3$ dimensions by reducing the d-dimensional case to the $(d-1)$-dimensional case. That is, we complete the proof Theorem 21 via induction on d, where the base cases $d \in \{1, 2\}$ are proven above. So assume Theorem 21 is true in the $(d-1)$-dimensional case. Let $P \subset \mathbb{R}^d$ be the input set of n points as before, and suppose the dimensions are labeled $\{1, 2, \ldots, d\}$. Store P in a random treap T using the d^{th} coordinate of the points to determine the binary search tree ordering, and a fresh 8-wise independent hash function to generate priorities. (Using fresh random bits implies that certain random variables are independent, and hence the expectation of their product is the product of their expectations. This fact aids our analysis considerably.) For each node $v \in T$, maintain a $(d-1)$-dimensional uniquely represented range tree data structure R_v on the points in T_v, where the point $(x_1, x_2, \ldots, x_d)$ is treated as the $(d-1)$-dimensional point $(x_1, x_2, \ldots, x_{d-1})$.

Inserting a point p then involves inserting p into T and modifying $\{R_v : v \in T\}$ appropriately. Deleting a point is analogous. We can bound the running time for these operations as follows.

Inserting p into T takes $O(\log n)$ time. We will also need to update R_v for each v that is an ancestor of p in T, by inserting p into it. Note that p has expected $O(\log n)$ depth by Theorem 6, and we can insert p into any R_v in expected $O(\log^{d-2} n \cdot \log\log n)$ time by the induction hypothesis, Thus we can update $\{R_v : v \text{ is an ancestor of } p \text{ in } T\}$ in expected $O(\log^{d-1} n \cdot \log\log n)$ time. (Here we have used the fact that the depth of p and the time to insert p into some fixed R_v are independent random variables.) Finally, inserting p into T will in general have involved rotations, thus requiring significant changes to some of the structures R_v for nodes v that are descendants of p in T. However, it is relatively easy to see that we can rotate along an edge $\{u, v\}$ and update R_u and R_v in expected time

$$O\left((|T_u| + |T_v|)\log^{d-2} n \cdot \log\log n\right)$$

using the induction hypothesis. Using Theorem 6 on page 59 with rotation cost $f(k) = O(k \log^{d-2} n \cdot \log\log n)$ then implies that these rotations take a total of $O(\log^{d-1} n \cdot \log\log n)$ expected time. (Here we rely on the fact that for any fixed u and v, $|T_u|$ and the time to update R_v are independent.) This yields an overall running time bound for insertions of expected $O(\log^{d-1} n \cdot \log\log n)$ time. The same argument applies to deletions as well.

Queries in higher dimensions resemble queries in two dimensions. Given a d-dimensional box query (p, q), we find the predecessor p' of p in T and the successor q' of q in T. Let w be the least common ancestor of p' and q' and let $A_{p'}$ and $A_{q'}$ be the paths from p' and q' (inclusive) to w (exclusive), respectively. Let V be the union of right children of nodes in $A_{p'}$ and left children of nodes in $A_{q'}$. For each $v \in A_{p'} \cup \{w\} \cup A_{q'}$, test if v is in box (p, q). This takes $O(d \log n)$ time in expectation, which is $O(\log^{d-1} n)$ for $d \geq 2$. Finally, issue a query $(\bar{p}, \bar{q})$ to each R_v for each $v \in V$, where $\bar{x}$ is the projection of x onto the first $(d-1)$ dimensions, so that if $x = (x_1, \ldots, x_d)$ then $\bar{x} = (x_1, \ldots, x_{d-1})$. The results of these queries are disjoint, each takes $O(\log^{d-2} n \cdot \log\log n + k)$ time in expectation by the induction hypothesis, and and there are $O(\log n)$ of them in expectation. Since the query times (conditioned on the set of points stored in each structure R_v) and the number of queries made are independent random variables, the total running time is $O(\log^{d-1} n \cdot \log\log n + k)$ in expectation, where k is the size of the output.

We now show that the space usage of our data structure is $O(n \log^{d-1} n)$ in expectation. As before, we proceed by induction on d. Assume that the space usage

is $O(n \log^{d-2} n)$ in expectation for a $(d-1)$-dimensional point set. The space usage is dominated by the structures $\{R_v : v \in T\}$, which by the induction hypothesis and linearity of expectation require

$$O\left(\sum_{v \in T} |T_v| \log^{d-2} |T_v|\right) \quad \text{which is} \quad O\left(\sum_{v \in T} |T_v| \log^{d-2} n\right)$$

space in expectation, where T is a random treap. Computing the expectation over the choice of random treap, the space usage is thus bounded by

$$\mathbf{E}\left[\sum_{v \in T} |T_v| \log^{d-2} n\right] = \mathbf{E}\left[\sum_{v \in T} |T_v|\right] \cdot \log^{d-2} n$$

However treaps have expected logarithmic subtree size [SA96], so $\mathbf{E}\left[\sum_{v \in T} |T_v|\right] = \sum_{v \in T} \mathbf{E}[|T_v|] = \sum_{v \in T} O(\log n) = O(n \log n)$. The total space required is therefore $O(n \log^{d-1} n)$.

5.9 Horizontal Point Location & Orthogonal Segment Intersection

Let $S = \{(x_i, x_i', y_i) : i \in [1 : n]\}$ be a set of n horizontal line segments. In the *horizontal point location problem* we are given a point $(\hat{x}, \hat{y})$ and must find $(x, x', y) \in S$ maximizing y subject to the constraints $x \leq \hat{x} \leq x'$ and $y < \hat{y}$. In the related *orthogonal segment intersection problem* we are given a vertical line segment $s = (x, y, y')$, and must report all segments in S intersecting it, namely $\{(x_i, x_i', y_i) : x_i \leq x \leq x_i' \text{ and } y \leq y_i \leq y'\}$. In the dynamic version we must additionally support updates to S. As with the orthogonal range reporting problem (see Section 5.8), both of these problems can be solved using fractional cascading and in the same time bounds [CG86] ($k = 1$ for point location and is the number of lines reported for segment intersection). Mortensen [Mor03b] improved orthogonal segment intersection to $O(\log n)$ updates and $O(\log n + k)$ queries.

We extend our ordered subsets approach to obtain the following results for horizontal point location and range reporting.

Theorem 22. *Let S be a set of n horizontal line segments in $\mathbb{R}^2$. There exists a uniquely represented data structure for the point location and orthogonal segment intersection problems that uses $O(n \log n)$ space, supports segment insertions and deletions in expected $O(\log n \cdot \log \log n)$ time, and supports queries in expected $O(\log n \cdot$*

$\log\log n + k)$ *time, where* k *is the size of the output. The data structure uses* $O(\log n)$ *random bits.*

5.9.1 The Data Structures

We will first obtain a hierarchical decomposition $\mathcal{D}$ of the plane into vertical slabs (in a manner akin to segment trees) using a random treap T on the endpoints E of segments in S. The treap T uses the natural ordering on the first coordinate to determine the binary search tree ordering. For $a, b \in \mathbb{R}^2$ with $a = (a_x, a_y)$ and $b = (b_x, b_y)$, we let $[a, b]$ denote the vertical slab $\{(x, y) : a_x \leq x \leq b_x, y \in \mathbb{R}\}$. The decomposition has as its root the whole plane; for concreteness we may imagine it as the vertical slab $[(-\infty, 0), (+\infty, 0)]$. A node $[a, b]$ in $\mathcal{D}$ has children $[a, c]$ and $[c, b]$ if $c = (c_x, c_y) \in E$ is the highest priority node in T such that $a_x < c_x < b_x$. Note that the decomposition tree $\mathcal{D}$ has nearly the same structure as T. To obtain the structure of $\mathcal{D}$ from T, it suffices to add nodes to T so that the root has degree two, and every other original node in T has degree three. It will be useful to associate each node $v \in T$ with a node $\bar{v} \in \mathcal{D}$, as follows: label the nodes of T and $\mathcal{D}$ as in a trie, and for $u \in T$ and $w \in \mathcal{D}$ let $w = \bar{u}$ iff u and w have the same label.

Each $[a, b] \in \mathcal{D}$ also has an associated subset of line segments in S, which we denote by $S_{[a,b]}$. In particular, line segment $(x, x', y) \in S$ is *associated* with $[a, b]$ if $[a, b] \subseteq [x, x']$ and for all ancestors $[a', b']$ of $[a, b]$, $[a', b'] \not\subseteq [x, x']$. Note that $s \in S$ may be associated with as many as $O(\log n)$ nodes in $\mathcal{D}$, in expectation. We store the sets $\{S_{[a,b]} : [a, b] \in \mathcal{D}\}$ in an ordered subsets structure, using the natural order on the second coordinate as our total order. As with range searching we also keep a treap T' on S ordered by the second coordinate which is used to insert new elements into the ground set L of the ordered subset structure.

To answer a point location query on a point $p = (x, y)$, first search for the narrowest slab $[a, b] \in \mathcal{D}$ with $a \leq x \leq b$. Let P be the path from this node to the root of $\mathcal{D}$. Insert y into L of the OSP instance (using T') and for each $[a, b] \in P$, search $S_{[a,b]}$ for the predecessor of y using ordered subsets. Of these $|P|$ segments, return the highest one.

To answer a segment intersection query on a vertical segment (x, y, y'), find the path P as in a point location query for (x, y). For each $[a, b] \in P$, search $S_{[a,b]}$ for the successor s_i of y, and report in order all segments in $S_{[a,b]}$ from s_i to the greatest segment at height at most y'.

To insert a segment (x, x', y), insert (x, y) and (x', y) into the treap T. As (x, y)

and (x', y) are rotated up to their correct positions, modify $\mathcal{D}$ accordingly and construct the sets $S_{[a,b]}$ of newly created slabs $[a, b]$ from scratch. We construct a set $S_{[a,b]}$ as follows. For each descendant e of a or b in T, determine if the segment with endpoint e is associated with $[a, b]$ in constant time. If so, insert the segment into $S_{[a,b]}$. In order to guarantee that this procedure correctly constructs $S_{[a,b]}$, we show the following: Every segment associated with $[a, b]$ in $\mathcal{D}$ has an endpoint that is a descendant of either a or b in T. See Claim 2 on the following page for the proof.

Finally, we may need to insert (x, x', y) into sets $S_{[a,b]}$ for slabs $[a, b]$ that were not affected by the insertions of x and x' into T and the corresponding modifications to $\mathcal{D}$. To find the sets to modify, find the path P from (x, y) to (x', y) in T, and consider $S_{[a,b]}$ for each $[a, b]$ that is a *child* of some node in $\{\bar{v} : v \in P\}$. For each, test in constant time if the new segment should be added to it, and add it accordingly. Deletion is similar.

5.9.2 The Analysis

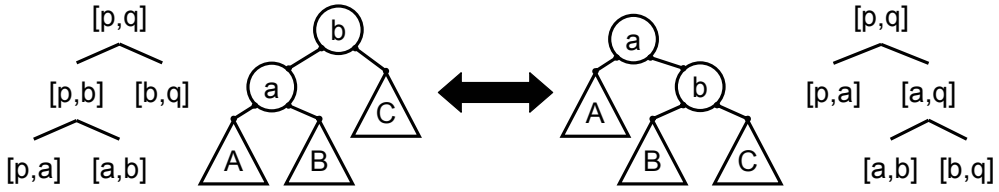


Figure 5.12: Rotation of the decomposition $\mathcal{D}$ and treap T about edge $\{a, b\}$, starting with slab $[p, q]$.

We start with the running time for queries. Note that the decomposition tree $\mathcal{D}$ has nearly the same structure as T. To obtain the structure of $\mathcal{D}$ from T, it suffices to add nodes to T so that the root has degree two, and every other original node in T has degree three. Thus the path P has expected logarithmic length and can be found in logarithmic time. Performing the $O(|P|)$ predecessor queries takes expected $O(|P| \log\log n)$ time. In a point location query, finding the maximum height result takes $O(|P|)$ time for a total of expected $O(\log n \cdot \log\log n)$ time. For a segment intersection query, if there are $k_{[a,b]}$ segments in $S_{[a,b]}$ intersecting the query segment, we can report them in expected time $O(\log\log n + k_{[a,b]})$ by either

performing fast finger searches in the treap $T_{[a,b]}$ which stores $S_{[a,b]}$ (thus finding the successor of a node in expected $O(1)$ time), or by storing pointers at each treap node $v \in T_{[a,b]}$ to its successor. The total expected time is thus $O(|P| \log\log n + \sum_{[a,b]\in P} k_{[a,b]})$. Since each segment in the answer appears in exactly one set $S_{[a,b]}$, this total is $O(\log n \cdot \log\log n + k)$.

We now analyze the running time for insertions. We first bound the cost to insert (x, y) and (x', y) into T, do rotations in $\mathcal{D}$, and create $S_{[a,b]}$ for newly created slabs. Note that this costs the same as updating T, up to constant factors, if the cost to rotate a subtree of size z is $z \log\log n$. Thus, by Theorem 6 on page 59 the update time per insertion is $O(\log n \cdot \log\log n)$ in expectation. Next we bound the cost to update $S_{[a,b]}$ for preexisting slabs $[a, b]$. Let P be the path from (x, y) to (x', y) in T. It is easy to prove using Theorem 6 that the expected length of P is $O(\log n)$. Thus the total time to make these updates is again $O(\log n \cdot \log\log n)$ in expectation. The analysis for deletions is similar.

Finally we consider the space usage. Using Claim 2 and the definition of segment association, it is not difficult to prove that a segment with endpoints e and e' can be associated with at most $|P_{e,e'}|$ slabs, where $P_{a,b}$ is the path from a to b in T. Since this is logarithmic in expectation, we conclude that each segment is stored at most $O(\log n)$ times in expectation. Since treaps and our ordered subset structure take linear space in expectation, the total space usage is thus $O(n \log n)$ in expectation.

Claim 2. *In the data structure of Section 5.9.1, every segment associated with* $[a, b]$ *in* $\mathcal{D}$ *has an endpoint that is a descendant of either* a *or* b *in* T.

Proof: Fix a segment $s = (x, x', y)$ with endpoints $e = (x, y), e' = (x', y)$. Let P be the treap path from e to e', and let T' be the subtree of T containing P and all descendants of nodes in P. Suppose for a contradiction that s is associated with $[a, b]$ but neither of its endpoints is a descendant of a or b. Thus $[a, b] \subseteq [x, x']$ and for all ancestors $[a', b']$ of $[a, b]$, $[a', b'] \not\subseteq [x, x']$. Let $a = (a_x, a_y)$ and $b = (b_x, b_y)$. Since s is associated with $[a, b]$, this implies $x < a_x \leq b_x < x'$. Note that $[a, b] \in \mathcal{D}$ implies that one of $\{a, b\}$ is a descendant of the other in T. Suppose b is a descendant of a (the other case is symmetric). We consider two cases: $a \in T'$ and $a \notin T'$.

In the first case, clearly $a \notin P$, so a must be a descendant of some node $v \in P$. Then if $c = (c_x, c_y)$ is the parent of a, then either $c_x < a_x$ or $c_x > b_x$, since otherwise $[a, b]$ would not be in $\mathcal{D}$. However, $c \in T'$, thus $x < c_x < x'$, and so either $[a, c]$ or $[c, b]$ contains $[a, b]$ and is contained in $[x, x']$, a contradiction.

In the second case, $a \notin T'$. By assumption, neither e nor e' is a descendant of

a, and $x < a_x < x'$, so there must be a treap node d with $x < d_x < a_x$ with higher priority than both a and e, and a node d' with $a_x < d'_x < x'$ with higher priority than both a and e'. However, a must be a descendant of at least one node in $\{d, d'\}$, and P must pass through ancestors of both d and d' (where each node is included among its own ancestors), contradicting the case assumption that $a \notin T'$. ∎

5.10 2-D Dynamic Convex Hull

In this section we obtain a uniquely represented data structure for maintaining the *convex hull* of a dynamic set of points $S \subset \mathbb{R}^2$. The convex hull of S is defined as the minimal convex set containing S. The *vertices* of the convex hull of S are those points which cannot be written as convex combinations of the other points in S. For our purposes, the convex hull is represented by an ordered set consisting of the vertices of the convex hull. (For concreteness we may define the ordering as starting with the point with minimum x-coordinate and proceeding in clockwise order about the centroid of S.) To ease exposition, we will refer to this representation as the convex hull, and refer to the minimal convex set containing S as the *interior* of the convex hull.

Our approach builds upon the work of Overmars & van Leeuwen [OvL81]. Overmars & van Leeuwen use a standard balanced binary search tree T storing S to partition points along one axis, and store the convex hull of T_v for each $v \in T$ in a balanced binary search tree. In contrast, we use treaps in both cases, together with the hash table of Section 3.1 for memory allocation. Our main contribution is then to analyze the running times and space usage of this new uniquely represented version, and to show that even using only $O(\log n)$ random bits to hash and generate treap priorities, the expected time and space bounds match that of the original version up to constant factors. Specifically, we prove the following.

Theorem 23. *Let $n = |S|$. There exists a uniquely represented data structure for 2-D dynamic convex hull that supports point insertions and deletions in $O(\log^2 n)$ expected time, outputs the convex hull in $O(k)$ time, where k is the number of points in the convex hull, reports if a query point is in the convex hull or in its interior in $O(\log k)$ expected time, finds the tangents to the convex hull from an exterior query point in $O(\log k)$ expected time, and finds the intersection of the convex hull with a query line in $O(\log k)$ expected time. Furthermore, the data structure uses $O(n)$ space in expectation and requires only $O(\log n)$ random bits.*

Our Approach. We will discuss how to maintain only the *upper* convex hull, the lower convex hull is kept analogously. (The upper convex hull consists of all vertices v of the convex hull such that if we move vertically upward from v we immediately exit the interior of the convex hull.) Let $U \subseteq \mathbb{R}^2$ be the universe of possible points, S be our set of points, and N be an upper bound on the number of points to be stored. We maintain a top level random treap T on the points, using an 11-wise independent hash function $h : U \to [N^3]$ to generate priorities, and using the natural ordering on the x-coordinates of the points as the key-ordering. That is, $(x, y) < (x', y')$ in the key ordering iff $x < x'$. (For simplicity, we will assume no two points have the same x-coordinate, and that no three points are collinear.) Let $V[T_v]$ denote the points in T_v. Each node v stores a point $p \in S$ as well as the *convex hull* of $V[T_v]$. This convex hull is itself stored in a modified treap on $V[T_v]$, which we call H_v. Each treap in $\{H_v : v \in T\}$ obtains key priorities from the same 8-wise independent hash function $g : U \to [N^3]$, and they all use the same key-ordering as T. We will also maintain with each node u in each H_v pointers to its predecessor and successor in H_v according to the key ordering. Abusing notation slightly, we will call these pointers $\mathrm{pred}(u)$ and $\mathrm{succ}(u)$. Maintaining these pointers during updates is relatively straightforward, so we omit the details. Note that as usual all of the pointers that we use are actually abstract pointers, that is, labels used to for memory allocation as discussed in Section 5.2.

Insertions. Suppose we are currently storing point set S, and insert point p. First we identify the leaf position l that p would occupy in T if it had priority $-\infty$, and then rotate it up to its proper position (given its priority $h(p)$). We then must recompute H_v for all v in the path P from l to the root of T. For $v \in P$ that are ancestors of p, we need only add p to H_v as described below. For each $v \in P$ that is either p and one of its descendants, we must merge the convex hulls of v's children, and then add v to the result.

Adding a Point. We first consider adding a point u to H_v, assuming that u is not already in H_v. First, we can determine if u is in the upper convex hull of $V[T_v]$ in expected $O(\log |H_v|)$ as follows. Find the nodes $a := \max\{w : w < u\}$ and $b := \min\{w : w > u\}$, which takes expected $O(\log |H_v|)$ time. Then do a *line side test* to see if u is above or on line segment (a, b). Point u is in the hull if and only if u is above or on (a, b), otherwise not. If u is not in the hull, we leave H_v unchanged. If u is in the hull, we must find the points x and y such that the upper hull is $\{w \in H_v : w \leq x\} \cup \{u\} \cup \{w \in H_v : w \geq y\}$. Once these are found, we can

split H_v at x and y, join the appropriate pieces, and add u to get the upper hull in $O(\log |H_v|)$ time.

We now discuss how to find x. Finding y is analogous. Let $\mathrm{line}(p, q)$ denote the line containing points p and q. Given a point $w < u$, we can conclude that $x \leq w$ if u is above or on $\mathrm{line}(w, \mathrm{succ}(w))$. Additionally, we can conclude that $x \geq w$ if u is below $\mathrm{line}(\mathrm{pred}(w), w)$. Thus we can do a binary search for x by traversing the path from the root to x in H_v. Since we have the pointers to find $\mathrm{succ}(\cdot)$ and $\mathrm{pred}(\cdot)$ in constant time, it follows that we can find x in $O(\mathrm{depth}(x))$ time, where $\mathrm{depth}(x)$ is the depth of x in H_v. By Theorem 6 on page 59, $\mathbf{E}[\mathrm{depth}(x)] \leq 2 \ln(|H_v|) + 1$, so adding a point takes expected $O(\log |H_v|)$ time.

The total time spent in adding points is $O(\log^2 n)$ in expectation. To see this, note that each addition takes $O(\log n)$ time in expectation, and there are at most the depth of l in T of them. Theorem 6 states that $\mathrm{depth}(l)$ is $O(\log n)$ in expectation. Finally, $\mathrm{depth}(l)$ is independent of the time taken for any point additions, since the former depends on h and the latter on g, so the expectation of their product is the product of their expectations.

Merging Two Upper Hulls. When rotating up the newly inserted point p in T, we must recompute H_v for each v involved in a rotation. We do this by *merging* the hulls of the children of v, say u and w, and then add v as described above. We can do this so that the expected time for all merges when adding a point to the top-level treap is $O(\log n)$. Our approach mimics that of [OvL81].

Suppose we want to merge the hulls of the children of v, say u and w, and then add v as described above. We initially begin with H_u and H_w such that all of the points in the former are smaller than all the points in the latter. We must find the *bridge* between them, that is, the pair of points (x, y) such that the upper hull of $V[T_u] \cup V[T_w]$ is $\{q \in H_u : q \leq x\} \cup \{q \in H_w : q \geq y\}$. Once we find x and y, two splits and a join immediately gives us the treap representation of the desired upper hull in $O(\log |V[T_v]|)$ expected time.

To find the bridge (x, y), we start with a guess $(x_0, y_0) = (\mathrm{root}(H_u), \mathrm{root}(H_w))$. We iteratively develop improved guesses (x_t, y_t), maintaining the invariant that x_t is an ancestor of x and y_t is an ancestor of y. In each step, we replace at least one of $\{x_t, y_t\}$ with one of its children to get (x_{t+1}, y_{t+1}), being careful to maintain the invariant. Clearly, after $\mathrm{depth}(x) + \mathrm{depth}(y)$ steps, we find the bridge.

To compute (x_{t+1}, y_{t+1}) from (x_t, y_t), we first find $\mathrm{pred}(x_t)$, $\mathrm{pred}(y_t)$, $\mathrm{succ}(x_t)$, and $\mathrm{succ}(x_t)$. Then, line-side tests can be used to find an improving move. Fig-

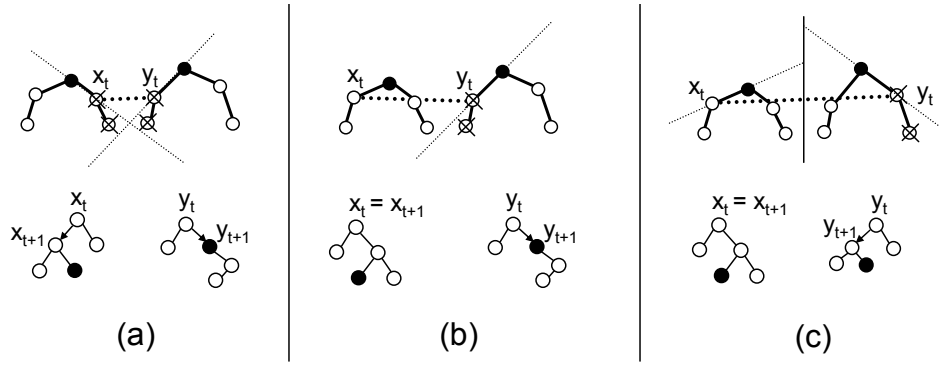


Figure 5.13: Finding the bridge. Both the convex hull and treaps storing them are displayed, with the bridge nodes in black. To test the current guess for the bridge (thick dotted line), we employ line-side tests. The thin dotted lines denote lines of interest in each of the three cases. Nodes marked "X" are discovered not to be incident on the bridge.

ure 5.13 shows three of four cases, where the forth case is identical to case (b) if we swap the role of x_t and y_t. More specifically, we apply the following tests in order.

1. Test if $\text{line}(\text{pred}(x_t), x_t)$ is below y_t (or equivalently, if $\text{line}(\text{pred}(x_t), y_t)$ is above x_t). If so, we may infer that $x < x_t$ in the binary search tree order, and thus we set x_{t+1} to be the left child of x_t. This is the case in Figure 5.13(a).

2. Test if $\text{line}(y_t, \text{succ}(y_t))$ is below x_t (or equivalently, if $\text{line}(x_t, \text{succ}(y_t))$ is above y_t). If so, we may infer that $y > y_t$ in the binary search tree order, and thus we set y_{t+1} to be the right child of y_t. This is the case in Figures 5.13(a) and 5.13(b).

3. If neither of the above tests allowed us to make progress (i.e., $\text{line}(\text{pred}(x_t), x_t)$ is above y_t and $\text{line}(y_t, \text{succ}(y_t))$ is above x_t), then test if $\text{line}(x_t, y_t)$ is below either $\text{succ}(x_t)$ or $\text{pred}(y_t)$. If so, arbitrarily select a vertical line l such that all points in H_u are to the left of l and all points in H_w are to the right of it. Let z_0 be the intersection of $\text{line}(x_t, \text{succ}(x_t))$ and l, and let z_1 be the intersection of $\text{line}(\text{pred}(y_t), y_t)$ and l. If z_0 is above z_1, then we may infer $x > x_t$ and set x_{t+1} to be the right child of x_t. Otherwise, we may infer $y < y_t$ and set y_{t+1} to be the left child of y_t, as is the case in Figure 5.13(c).

If no three points are collinear and none of these tests make any progress, then

we may infer that we have found the bridge, i.e., $x = x_t$ and $y = y_t$. Though we omit the proof of this fact, Figure 5.14 illustrates some of the relevant cases, and is suggestive of how the proof goes. See [OvL81] for further details.

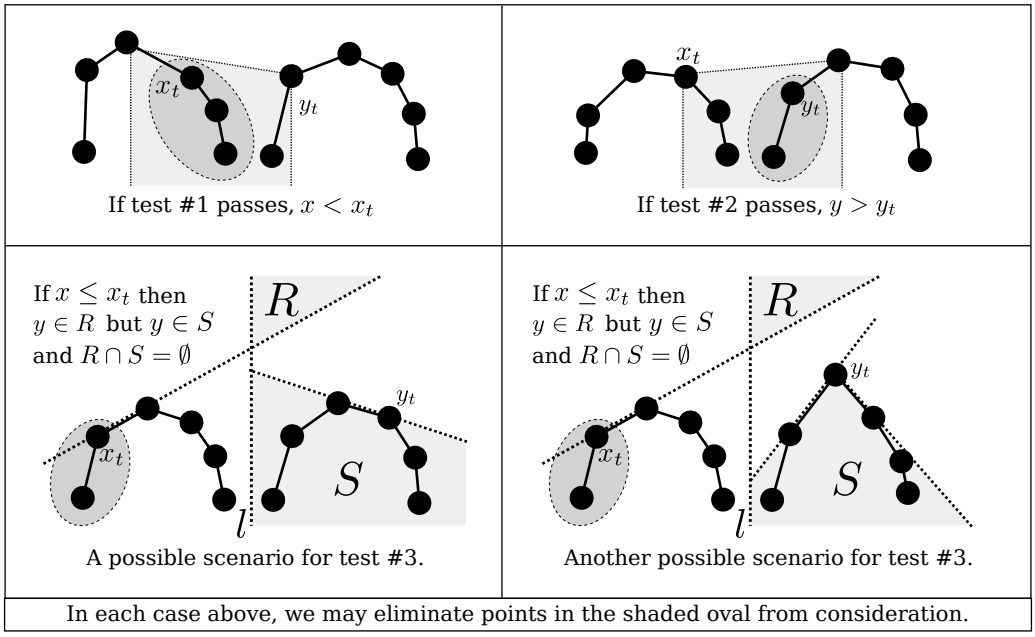


Figure 5.14: Illustrations of some of the cases encountered when merging two convex hulls, with proof sketches.

In expectation, there are $\mathbf{E}[\text{depth}(x) + \text{depth}(y)] = O(\log |V[T_v]|)$ steps to find the bridge, and each step takes $O(1)$ time using the pointers for $\text{pred}(\cdot)$ and $\text{succ}(\cdot)$. Since treaps provide expected logarithmic time insertions even if tree rotations take linear time, and the cost to merge is independent of the top level treap's structure, the expected time for all merges when adding a point to the top-level treap is $O(\log n)$.

Deletions. To delete a point p, we rotate p down to its correct leaf position l (by treating it as though its priority were $-\infty$), then cut the leaf. We must then update H_v and H'_v for each v on the path from l to the root. Working our way up, and recomputing H_v and H'_v by merging their children's corresponding treaps, and then adding v itself, we can update all the necessary entries in $O(\log^2 n)$ expected time. The running time argument is similar to that for insertions, and we omit it here.

The Convex Hull Queries. To obtain the convex hull we simply do an in-order traversal of H_r, where r is the root of T. This clearly takes time linear in the size of the hull.

To determine if a query point p is in the convex hull, we simply search H_r for it, where r is the root of T. To determine if p is in the interior of the convex hull, find the predecessor u_{up} and successor v_{up} of p in the upper hull. Then find the predecessor u_{low} and successor v_{low} of p in the lower hull. Then p is in the interior of the convex hull if and only if $\text{line}(u_{\text{up}}, v_{\text{up}})$ is above p and $\text{line}(u_{\text{low}}, v_{\text{low}})$ is below p. Given random treaps containing the upper and lower convex hulls, both of these queries clearly take expected $O(\log k)$ time, where k is the number of points in the convex hull.

Finding the tangents to the convex hull from an exterior query point p involves searching the treap storing the convex hull, where lines side tests involving a candidate node u and its predecessor and successors allow one to determine whether a tangent point is in the left or right subtree of u in constant time. Determining the intersection of the convex hull with a query line is similar. Both queries can thus be done in expected $O(\log k)$ time. We omit the details on how to guide the searches, and instead refer the reader to [OvL81].

Space Usage. The top level treap T clearly requires only $O(n)$ space. Most of the space required is in storing $\{H_v : v \in T\}$. We store these treaps functionally, so storing H_v requires only $O(\log |V[T_v]|)$ pointers in expectation. Specifically, suppose v has children u and w. Given H_u and H_w, storing H_v requires storing a split path in each of H_u and H_w, and the insertion path for v. Each of these requires $O(\log |V[T_v]|)$ pointers in expectation [SA96]. Thus the total space is $O(\sum_v \log |V[T_v]|)$ in expectation, and we need to bound $\mathbf{E}[\log |V[T_v]|]$ for each v. Lemma 13 states that $\mathbf{E}[\log |V[T_v]|] = O(1)$ for each v, so the total space usage is $O(n)$ in expectation.

Lemma 13. *A treap T on n nodes with priorities drawn from an 11-wise independent hash function $h : U \to \{0, 1, \ldots, r\}$ with $r \geq n^3$ satisfies $\mathbf{E}[\log |T_v|] = O(1)$.*

Proof: It is easy to prove that probability of a collision on priorities is at most $\binom{n}{2}/r < 1/2n$. In this case, we use the trivial bound $|T_v| \leq n$ to show that

$$\mathbf{E}[\log |T_v|] \leq \log(n)/2n + \mathbf{E}[\log(|T_v|) \mid \text{no collisions}].$$

So assume there are no collisions on priorities. Lemma 4 on page 91 states that

$\mathbf{Pr}[|T_v| = k] = O(1/k^2)$ for any $1 \leq k < n$ and $\mathbf{Pr}[|T_v| = n] = O(1/n)$. Thus $\mathbf{E}[\log |T_v|] = \sum_{k=1}^{n} \mathbf{Pr}[|T_v| = k] \cdot \log(k) = \sum_{k=1}^{n-1} O(\frac{\log k}{k^2}) + O(\frac{\log n}{n}) = O(1)$. ∎

5.11 Unique Representation via Obliviousness

In this section we will present a new general technique for converting certain *oblivious* data structures implemented on a pure pointer machine into a RAM data structure that is uniquely represented with high probability. Oblivious data structures were introduced by Daniele Micciancio [Mic97]. The general definition of obliviousness is somewhat technical, however for our purposes obliviousness is merely weak history independence (see Definition 1 on page 21) in a suitable machine model. In the specific case of oblivious trees, obliviousness is merely weak history independence in a pointer machine model. In other words, a tree is oblivious if all sequences of operations (starting from the initial state) that induce the same logical state induce the same distribution over the *pointer structure* of the tree. By analogy with strong history independence, we say that a data structure is *strongly oblivious* if all sequences of operations (starting from the initial state) that induce the same logical state induce the same pointer structure. Our derived RAM data structure will have nearly the same space and time efficiency as the original, and will be uniquely represented if the original data structure is strongly oblivious. This reduces the problem of developing a uniquely represented data structure to developing a strongly oblivious data structure on a pure pointer machine. This applies, for example, to the treaps of Seidel and Aragon [SA96], and many of the dynamized algorithms of Acar *et al.* [ABH+04].

We introduce some terminology before formally stating our result. For our purposes, a *pure pointer machine* is a machine with a finite set of registers, which can create nodes called *cons cells*. Each cons cell has two fields, each of which may store either a pointer to a cons cell or a word of data. Similarly, each register may store either a pointer to a cons cell or a word of data. The machine may do arithmetic on data in registers, but not on pointers. All access to cons cells is through following pointers. The machine is called *pure* because the fields of each cons cell may be set only on creation. Thus all data stored in the graph is immutable as in a purely functional language, and the pointer machine can only create directed acyclic graphs. See [BA95] for an explanation of various pointer machines models.

For a state σ of the pointer machine, let $r_i(\sigma)$ be the contents of register i in

state σ, let $G(\sigma)$ be the directed graph of cons cells reachable from the registers in state σ, and let $f_i(v, \sigma)$ be the value in the i^{th} field of cons cell $v \in V[G(\sigma)]$ in state σ. Let $F_D(\sigma)$ and $F_C(\sigma)$ denote all fields of vertices in $G(\sigma)$ storing data and pointers to cons cells, respectively. For a pointer p, let $[p]$ denote the object it points to. Two states of the pointer machine, σ_1 and σ_2, are said to be *structurally equal* if for all registers i storing data in σ_1, $r_i(\sigma_1) = r_i(\sigma_2)$, and there exists an isomorphism $g : V[G(\sigma_1)] \to V[G(\sigma_2)]$ such that for all registers i storing pointers to cons cells in σ_1, $g([r_i(\sigma_1)]) = [r_i(\sigma_2)]$ and for all fields $f_j(v, \sigma_1) \in F_D(\sigma_1)$, $f_j(v, \sigma_1) = f_j(g(v), \sigma_2)$, and for all fields $f_j(v, \sigma_1) \in F_C(\sigma_1)$, $f_j(v, \sigma_1) = v' \iff f_j(g(v), \sigma_2) = g(v')$.

We are now ready to state the theorem. We defer the proof to the end of this section.

Theorem 24. *Any algorithm on a pure pointer machine can be mapped on to a RAM such that with high probability the (multiplicative) space overhead is constant, $f(n)$ time operations in the pointer machine take amortized expected $O(f(n))$ time in the RAM, and structural equality of two states in the pointer machine implies equality of the memory representation of the two states in the RAM.*

We use the technique of *hash-consing* [Got74] to obtain this result. Hash-consing is the technique of hashing a cons cell into memory using both of its fields together as the key (see Section 5.2). Thus if two cons cells have identical contents, they hash to the same location. One limitation of this approach is that all accesses to the data structure in a pure pointer machine must proceed down through the data structure starting at a register. Thus, whereas some operations in some data structures may be significantly faster if one is given the right initial pointer, uniquely represented data structures constructed with this technique will not in general be able to exploit such speedups. It remains an open problem to obtain a result akin to Theorem 24 using a construction that allows one to exploit such speedups. On the positive side, we conclude that any algorithm implemented on a pure pointer machine that is strongly oblivious (i.e., where each state has a unique pointer structure) can be implemented as an computation in a RAM which with high probability is uniquely represented, has only constant space overhead, and has constant expected slowdown.

Proof of Theorem 24: We first show how to simulate the pointer machine on a RAM. Simulating the pointer machine registers via an array is trivial. Let C be the set of cons cells, and let D be the set of data values. To reduce the probability of collisions, we assign labels drawn from a large set $\mathcal{L}$ as follows. Select fast pairwise independent hash functions $\varphi_D : D \to \mathcal{L}$ and $\varphi_{\mathcal{L}} : \mathcal{L}^2 \to \mathcal{L}$. When a cons cell is

created, compute its label using $\varphi_{\mathcal{L}}$ applied to the labels associated with its fields. Proposition 3 on page 76 then guarantees that the probability that there exists two cons cells with the same label is $O(n^2/|\mathcal{L}|)$, where there are n cons cells. Thus using a set of labels of size n^c, where $c > 2$ is user specified, with high probability no two cons cells receive the same label. Should any two cons cells receive the same label, we abort.

When simulating the pointer machine algorithm, we simply store labels (rather than pointers) in fields. We can now use these labels as keys when hashing the cons cells into RAM using, e.g., the uniquely represented hash table of Section 3.1. To maintain unique representation, we will also have to implement garbage collection, however if we use a reference counting garbage collector (see e.g., [Jon96]) we can easily amortize the cost of garbage collection against the cost of creating the cons cells.

Because the labels are assigned using φ_D, $\varphi_{\mathcal{L}}$, and the labels of a cell's children's labels, it is straightforward to prove via structural induction that two structurally equal states σ_1 and σ_2 with isomorphism g satisfy the condition that $v \in V[G(\sigma_1)]$ and $g(v) \in V[G(\sigma_2)]$ receive the same label. Thus they define the same set of keys (with identical auxiliary data) to be stored in the uniquely represented memory allocator, and have identical memory representations in the RAM.

It remains to prove the claimed performance guarantees. Assume no two cons cells receive the same label. The n cons cells can be stored in a uniquely represented hash table with $O(n)$ slots, where each slot is large enough to store a cons cell. The hash functions φ_D and $\varphi_{\mathcal{L}}$ require only $O(1)$ words of RAM to store. Thus the whole simulation requires only $O(n)$ cons cells worth of space. As for time, consider a pointer machine operation that takes $f(n)$ time, for some function f. The operation may perform the following native pointer machine operations: dereference a pointer, read data from cons cells and registers, perform basic computations on that data, create a cons cell, and alter the contents of a register. We consider each of these actions in turn.

Dereference a Pointer. Dereferencing pointers is simulated by a hash table lookup on the appropriate label, and thus each pointer dereference is simulated in expected constant time.

Read Data and Perform Basic Computations. Reading data from cons cells and registers takes constant time per read in a RAM, and the RAM can perform all the basic computations supported by the pointer machine in constant time.

Thus the time to perform these actions in simulation can be charged against the time to perform them in the pointer machine.

Create a Cons Cell. To simulate the creation of a cons cell, the RAM must create the cell, generate a label for it, and hash it into memory. This takes expected constant time. Note that because we are dealing with a pure pointer machine, when a cons cell is created it has no cons cells pointing to it. Thus, the creation of a cons cell cannot affect the labels of any other cons cells.

Alter the Contents of a Register. Altering the contents of a register takes constant time on the RAM. However, if the register previously stored a pointer and updating it caused some subgraph $G' = (V', E')$ stored by the pointer machine to become unreachable from all of the registers, then the unique representation constraint requires eager garbage collection of G'. It is straightforward to see that this can be accomplished in expected $O(|V'|)$ time by traversing G', using, for example, depth-first search, and deleting all the cons cells that compose it. We amortize the cost of this garbage collection against the cost to create the cons cells. That is, we may use $\Phi = |C|$, the number of cons cells, as the potential function for the amortization scheme.

Each native pointer machine operation is thus simulated in amortized expected constant time, so that a pointer machine operation that takes $f(n)$ time will take amortized expected $O(f(n))$ time when simulated on a RAM. Finally, note that amortization is used only for garbage collection; any pointer machine operation that does not require garbage collection and takes $f(n)$ time will take expected $O(f(n))$ time when simulated on a RAM. ■

5.12 Dynamic Trees and Unique Representation via Dynamization

Acar *et al.* [ABH+04] considered the problem of running an off-line algorithm in an online environment while maintaining enough state so that as the input changes the algorithm can exploit its earlier work as much as possible rather than restarting the computation from scratch each time. Their machine model is a non-standard RAM in which each memory location can be written to only once, and reads must occur during a function call. The first restriction simplifies the problem and is satisfied by all purely functional programs, while the second restriction simplifies

the tracking of memory reads. In this model, Acar *et al.* define a notion of the *stability* of a program with respect to changes in the input, which is an upper bound on the amount of recalculation that must be done when the input changes. More formally, given an algorithm A and an input I define the *trace* of A on I, denoted $A_T(I)$, to be the function-call tree of the execution of A on I. Let Δ be the class of valid input changes, with $(I, I') \in \Delta$ if there is a valid way to change I to I'. Define a distance function on traces, δ, which is the sum of the costs of function calls that are different in the two input traces. Then A is $O(f(n))$-stable if

$$\max_{(I,I')\in\Delta,\ |I|=n} \{\delta(A_T(I), A_T(I'))\} \ = \ O(f(n)).$$

Finally, an algorithm is said to be r-*round parallel* if it operates in r rounds where reads occurring in round i only read locations written to in rounds $j < i$, for all i. Acar *et al.* then prove the following theorems.

Theorem 25 (Theorem 4.2 of [ABH+04])**.** *Let A be an r-round parallel algorithm that is $O(f(n))$-stable for some class of input changes Δ, then the output of A can be updated in $O(f(n) + r)$ time for any change in Δ. Furthermore, if each location is read only a constant number of times, then there is a strongly history independent implementation of the resulting dynamic algorithm.*

Theorem 26 (Theorem 4.1 of [ABH+04])**.** *Let A_T be a the trace-generator for an algorithm A. If A_T is $O(f(n))$-stable for a class of input changes Δ, then the output of A can be updated in $O(f(n) \log f(n))$ time for any change from Δ.*

We discuss applications of these theorems in the following subsections.

5.12.1 Dynamic Trees

The *dynamic trees* abstract data type stores a collection of vertex-disjoint trees and supports the following operations.

- $\text{make-tree}(x)$: create a new singleton tree on a new vertex labeled x.
- $\text{link}(u, v, x)$: add edge $\{u, v\}$ of cost x. This operation assumes u and v initially reside in different trees.
- $\text{cut}(\{u, v\})$: delete the edge $\{u, v\}$.

- $\text{update}(u, v, x)$: add real number x to the costs of all edges on the path from u to v, assuming such a path exists.
- $\text{cost}(\{u, v\})$: return the cost of edge $\{u, v\}$.
- $\text{min-cost}(u, v)$: return the minimum cost edge e on the the path from u to v, assuming such a path exists.

The dynamic trees abstract data type was originally introduced in a different form by Sleator and Tarjan [ST83, ST85]. The original specification stores a collection of vertex-disjoint *rooted* trees and supports the following operations.

- $\text{make-tree}(x)$: create a new singleton tree on a new vertex labeled x.
- $\text{cut}(v)$: delete the edge $(v, \text{parent}(v))$.
- $\text{link}(u, v, x)$: add edge $\{u, v\}$ of cost x and make v the parent of u. This operation assumes u and v initially reside in different trees, and u is the root of its tree.
- $\text{evert}(v)$: Make v the root of the tree that contains it.
- $\text{root}(v)$: return the root of the tree containing vertex v.
- $\text{update}(v, x)$: add real number x to the costs of all edges on the path from v to $\text{root}(v)$.
- $\text{parent}(v)$: return the parent of v.
- $\text{cost}(v)$: return the cost of edge $\{v, \text{parent}(v)\}$.
- $\text{min-cost}(v)$: return the vertex w closest to $\text{root}(v)$ such that edge $\{w, \text{parent}(w)\}$ has minimum cost among edges on the tree path from v to $\text{root}(v)$.

Sleator and Tarjan gave two data structures supporting all of the above operations (and a few additional ones) in amortized $O(\log n)$ time and worst-case $O(\log n)$ time, respectively [ST83, ST85]. Their solution represents each tree as a collection of vertex-disjoint paths. Their implementation presumably influenced their choice of operations and we favor the abstract data type definition involving unrooted trees, which is arguably more natural. Note that any implementation of that abstract data type can easily be extended to an implementation of Sleator

and Tarjan's abstract data type. From now on when we refer to the dynamic trees abstract data type, we will be referring to the one on unrooted trees.

There has been extensive additional work on the dynamic trees problem. For a detailed discussion, we refer the reader to Werneck's thesis [Wer06]. Among the various approaches to the problem is *tree contraction* via the *rake and compress trees* of Miller and Reif [MR89, MR91]. This is the approach used by Acar *et al.*, who develop a strongly history independent solution that supports all the dynamic tree operations expected $O(\log n)$ time using $O(n \log n)$ space [ABH$^+$04, ABV05]. Using the uniquely represented hash tables of Section 3.1 to store the elements in their data structure, we can immediately reduce the space to $O(n)$ and obtain the following result.

Theorem 27. *There exists a uniquely represented implementation of dynamic trees that supports* $\mathrm{make\text{-}tree}$*,* cut*,* link*,* update*,* cost*, and* $\min\text{-}\mathrm{cost}$ *operations in expected* $O(\log n)$ *time, and uses* $O(n)$ *space.*

Acar *et al.* [ABV05] also mention several other queries that their data structure can support in expected $O(\log n)$, all of which carry over to the uniquely represented version. Let $c(e)$ denote the cost of edge e. The edge costs induce a distance function d on pairs of nodes, where $d(u, v)$ is equal to the sum of the edge costs among edges in the path from u to v. The supported queries include the following.

Path Queries: Fix an arbitrary associative operator $\oplus : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ that can be computed in constant time. Given u and v in the same tree, let $P = \langle w_1, \ldots, w_t \rangle$ be the path from u to v. Return $c(\{w_1, w_2\}) \oplus c(\{w_2, w_3\}) \oplus \cdots \oplus c(\{w_{t-1}, w_t\})$.

Subtree Queries: Fix an associative and commutative operator $\oplus : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ that can be computed in constant time. Given v and r in the same tree T, let T_v denote the subtree of v in the tree obtained from rooting T at r. Let E be the edges of T_v. Return $\bigoplus_{e \in E} c(e)$.

Diameter: Given a node v in tree T, return the *diameter* of T, defined as the cost of the maximum cost simple path in T.

Centers: Given a node v in tree T, return a *center* of T, defined as a node u minimizing the maximum distance to any other node in T.

Set Distance: Given a node u and set of nodes S, return $d(u, S) := \min_{v \in S}\{d(u, v)\}$.

LCA: Given u, v, and r in the same tree T, return the *least common ancestor* (LCA) of u and v in the tree obtained from rooting T at r, defined as the common ancestor of u and v at maximum distance from the root.

5.12.2 Unique Representation via Dynamization

The automatic dynamization technique of Acar *et al.*, in their own words,

> "[...] builds a dynamic dependence graph [that encodes dependencies between code and data] to represent an execution and uses a change propagation algorithm to update the output and the dependencies whenever the input changes [ABH06]. A dynamic dependence graph maintains the relationship between code and data in a way that makes it easy for the change-propagation algorithm to identify code, called a reader, that depends on a changed value. Readers are stored as closures, i.e., functions with environments. This enables re-execution of a reader in the same state as before — modulo of course the changed values. The change-propagation algorithm maintains a queue of readers affected by changes and re-executes them in sequential execution order. When a reader is re-executed, the dependencies that it created in the previous execution are deleted and dependencies created during re-execution are added to the dependence graph. It is in this way that the dependence graphs are dynamic. Re-executing a reader may change other data values, whose readers are then inserted into the queue. Change propagation terminates when the reader queue becomes empty." [ABH+04]

Acar *et al.* note that the dependence graph itself is strongly history independent. To make the dynamized data structure strongly history independent, it is thus necessary and sufficient to ensure that the supporting data structures used for the dynamization are also strongly history independent (including the memory allocator). The required data structures are a hash table, a priority queue, and an order maintenance data structure. Efficient strongly history independent versions of these data structures can be found in Chapter 3, Section 4.5, and Section 5.6, respectively. Applying Theorem 26 (Theorem 4.1 of [ABH+04]), we immediately obtain the following result.

Theorem 28. *Fix a computational problem $\mathcal{P}$ and an abstract data type $\mathcal{A}$ that is a dynamic version of problem $\mathcal{P}$. Let Δ be the class in input changes allowed by $\mathcal{A}$. If there is an algorithm A for $\mathcal{P}$ in the restricted RAM model of [ABH+04] with a trace-generator A_T that is $O(f(n))$-stable for Δ then there is a strongly history independent implementation of $\mathcal{A}$ on a RAM model enhanced with direct support for function calls that supports changes in Δ in expected $O(f(n) \log f(n))$ time.*

Moreover, if $\mathcal{A}$ is reversible then by same argument used in the proof of Theorem 1 on page 23 the implementation is uniquely represented as well.

Chapter 6

Adaptive Variants of Uniquely Represented Data Structures

In previous chapters we have considered uniquely represented data structures that are strongly history independent, and thus store absolutely no information about their historical use beyond that required by their abstract data types. We have demonstrated uniquely represented implementations of several important abstract data types that match, or very nearly match, the time and space complexity of the best corresponding conventional implementations, up to taking expectations. However, our running time guarantees are typically worst-case (with respect to the instance) expected time bounds. Many data structures are specifically designed to store information about the historical sequence of operations, in some form or another, in order to allow the data structure to *adapt* to its workload and thus provide superior amortized performance. Splay trees [ST85] and the union-find data structure [Tar75] (see Section 7.2) are two examples. Unfortunately, there is no hope of creating uniquely represented data structures that adapt to the historical sequence of operations in this manner. Indeed, the phrase "adaptive uniquely represented data structures" is something of a contradiction in terms. In this chapter we consider a novel relaxation of unique representation and strong history independence, in which the representation is allowed to depend on certain information about the historical use of the data structure that is specified up front. The hope is that limited history dependence will allow the data structure to adapt to the input and achieve improved performance without storing *unauthorized* information about the historical use of the data structures.

6.1 Relaxing Unique Representation

Given an abstract data type $\mathcal{A} = (V, v_0, \tau, \lambda)$ with operations $\mathcal{O}$, we model the problem of developing an "adaptive uniquely represented data structure" (relative to some notion of what history we are allowed to reveal) as the problem of developing a uniquely represented implementation for a related abstract data type $\mathcal{A}'$. Before specifying how we obtain $\mathcal{A}'$ from $\mathcal{A}$, we describe formally how to model what historical information we are allowed to reveal. Let H be an arbitrary set, called the set of *histories*, and fix a *historical transition function*

$$\psi : V \times \mathcal{O} \times H \to H.$$

Also define a distinguished history $h_0 \in H$ called the *initial history*. The historical information that the implementation is allowed to store given that it has historically performed the sequence of operations $\langle O_0, O_1, O_2 \ldots, O_{k-1} \rangle$ is then h_k, where $h_{i+1} := \psi(v_i, O_i, h_i)$ and v_i is the logical state of the data structure after operations $\langle O_0, O_1, O_2 \ldots, O_{i-1} \rangle$ (i.e., $v_i := \tau(v_i, O_i)$ for all $i \geq 0$). We call h_k the *current history* in this case.

Once we have H and ψ, we can define $\mathcal{A}' = (V', v_0', \tau', \lambda')$ as follows. The logical states V' of $\mathcal{A}'$ consist of all pairs $(v, h) \in V \times H$ that are reachable from (v_0, h_0), by which we mean there is a sequence of operations that puts $\mathcal{A}$ in logical state v with current history h. The start state is simply $v_0' := (v_0, h_0)$. The operations for $\mathcal{A}'$ are the same as for $\mathcal{A}$. For all operations O the transition function is

$$\tau'((v, h), O) := (\tau(v, O), \psi(v, O, h))$$

and the output function is

$$\lambda'((v, h), O) := \lambda(v, O).$$

This formalism provides a versatile means to specify just what historical information is allowed to be stored and revealed. Here are some examples.

1. If $H = \{h_0\}$, then we recover the original notions of unique representation and strong history independence.

2. If H equals all (possibly complex) paths of length k or less in the state transition graph $G = (V, E)$ of $\mathcal{A}$, $h_0 = \langle v_0 \rangle$, and ψ is defined as

$$\psi(v, O, \langle u_0, u_1, \ldots, u_j \rangle) = \begin{cases} \langle u_0, u_1, \ldots, u_j, \tau(v, O) \rangle & \text{if } j < k - 1 \\ \langle u_1, \ldots, u_j, \tau(v, O) \rangle & \text{if } j = k - 1 \end{cases}$$

then the current history is precisely the previous k logical states that the abstract data type was in.

3. For a data structure storing a set of elements from a universe U, if $H = \{X : X \subseteq U, |X| \leq k\}$ and $h_0 = \emptyset$, then by defining ψ appropriately we can ensure the current history is the set of the previous k elements inserted/deleted/accessed.

4. For an abstract data type with k operations, if $H = \{0, 1, \ldots, p-1\}^k$ we can store counts of how many times each operation was performed modulo some large number p. Similarly, we can keep track of how many times each element was accessed or modified. For example, if the abstract data type is a binary search tree of keys from universe U, then we can set H to be all partial functions from U to $\mathbb{N}$, and set ψ to ensure that the current history h has as its domain the current set of keys being stored and that $h(e)$ is the number of times e has been accessed.

In general, in our setting we can model any notion of what history we are allowed to reveal as an equivalence relation $\sim$ over the set of all finite sequences of operations $\mathcal{H}$. The revealed history is then an equivalence class. Without loss of generality, $\sim$ refines the equivalence relation $\approx$, where $\omega \approx \omega'$ if and only if ω and $\omega' \in \mathcal{H}$ lead to the same logical state. Any such notion can be expressed by some pair (H, ψ) using our formalism. More complex notions of what history we are allowed to reveal are possible if, for example, there is some probabilistic model of user behavior or if the machine can hide random bits from the observer. We leave such matters for future work. Finally, note that we can also use this formalism to characterize what history is leaked by existing conventional data structures.

6.2 Adaptive Uniquely Represented Search Trees

We initiate the study of adaptive uniquely represented data structures starting with the binary search tree abstract data type (see Section 4.5). Among binary search trees that adapt to their sequence of operations, the splay trees of Sleator and Tarjan [ST85] are distinguished for their many performance guarantees when servicing various access patterns. We present two adaptive uniquely represented data structures that each have at least one of these desirable guarantees, namely *static optimality* and the *working set bound*, yet do not store very much history. The first

construction is actually due to Kalai and Vempala [KV05], whereas the second is a slight variant of a data structure due to Bădoiu *et al.* [BCDI07]. Here are the corresponding results for splay trees.

Theorem 29 (Static Optimality Theorem for Splay Trees [ST85]). *Fix a splay tree on n elements. For any item i, let q_i be the access frequency of item i, that is, the total number of times i is accessed. Let m be the total number of accesses. If every item is accessed at least once, then the total access time is $O(m + \sum_{i=1}^{m} q_i \log(m/q_i))$.*

Theorem 30 (Working Set Theorem for Splay Trees [ST85]). *Fix a splay tree on n elements. Fix a sequence $\{a_i\}_{i=1}^{m}$ of m accesses. For $i \in [1 : m]$, let t_i be the number of different elements accessed before the i^{th} access since the last access of element a_i, or since the beginning of the sequence if i is the first index at which element a_i appears. Then the total access time is $O(n \log n + m + \sum_{i=1}^{m} \log(t_i + 1))$.*

The static optimality theorem is so named because the best fixed binary search tree for a sequence of m accesses in which each element e is accessed $q_e > 0$ times requires $\Omega(m + \sum_{i=1}^{m} q_i \log(m/q_i))$ time in total to perform the accesses. This is an easy corollary of Shannon's source coding theorem (see Theorem 9 of [Sha48]); simply treat the fixed search tree as a coding scheme in which each symbol is encoded as the path from the root to it, as in a trie.

6.2.1 Achieving Static Optimality

As Theorem 29 suggests, static optimality is really only helpful when accesses greatly outnumber insertions and deletions. Thus in this section we focus on the case that the set of keys is fixed; the tree in this section will support only lookup operations.

Fix any infinite sequence of accesses $\rho = \langle a_1, a_2, a_3, \ldots \rangle$ on n keys. Let $\rho_i = \langle a_1, a_2, \ldots, a_i \rangle$. Kalai and Vempala [KV05] designed a binary search tree data structure that has the following property.

> For $i \in \mathbb{Z}^+$, let C_i^* be the total cost of the best fixed search tree for ρ_i to perform ρ_i, and let C_i be the total cost of our data structure to perform ρ_i. Then
>
> $$\lim_{k \to \infty} \frac{\mathbf{E}[C_k]}{C_k^*} = 1.$$

Kalai and Vempala obtain this result using techniques for online algorithms from the field of machine learning (see [Blu98] for a survey). We present a slight modification that stores relatively little historical information. (See Section 1.2 of [KV05] for the original construction.) Since Kalai and Vempala do not appear to be concerned with limiting the storage of historical information, it seems that this is one of those auspicious occasions in research where a result has previously unsuspected implications.

Theorem 31. *Fix any $T \in \mathbb{N}$ and any sequence of T accesses $\rho = \langle a_1, a_2, \ldots, a_T \rangle$ on n keys. Let C_T^* be the total cost of the statically optimal tree for ρ. Then there exists a tree such that*

1. *The only historical information the tree maintains are two functions*

$$e \mapsto |\{i : i \leq t_1, a_i = \text{lookup}(e)\}| \text{ and } e \mapsto |\{i : i \leq t_2, a_i = \text{lookup}(e)\}|$$

   with domain equal to the elements being stored at time t_1 and t_2, respectively.

2. *if C_T is the cost of the tree on ρ, then $\mathbf{E}[C_T] \leq C_T^* + 2n\sqrt{nT}$ and thus $\frac{\mathbf{E}[C_T]}{C_T^*} \leq 1 + \frac{2n\sqrt{n}}{\sqrt{T}}$.*

Above, t_1 is the time at which the tree was last modified and t_2 is the current time.

Kalai and Vempala's tree employs the *follow the perturbed leader* algorithm, and periodically computes the statically optimal tree for the current sequence of accesses with some added noise and then uses it for the next several accesses. Given a vector of frequencies of accesses, the optimal static tree for it can be found in $O(n^2)$ time using a clever dynamic programming algorithm due to Knuth [Knu71]. We provide pseudocode for our modification of Kalai and Vempala's algorithm for the case that T is fixed in advance in Figure 6.2.1 on the next page. Our modifications consist of setting v_i according to a hash function, and storing both f_i and r_i terms. In the pseudocode, f_i denotes the observed frequency of access of element i at the time of the previous tree modification, and r_i denotes the number of accesses to element i since then. The history stored then consists of $e \mapsto f_e$ and $e \mapsto (f_e + r_e)$.

The performance guarantee of Theorem 31 follows from that of Kalai and Vempala's original tree. The fact that the leaked history is exactly $e \mapsto f_e$ and $e \mapsto (f_e + r_e)$ follows from the fact that these functions are stored, and the tree structure is a deterministic function of them and the random bits of the machine. In fact, we have ensured that our implementation stores these functions to aid in precisely

```
lazy-leading-tree(integer N)
    For (i = 1, 2, ..., n)
        Set f_i = 0 and r_i = 0;
        Set v_i = h(i) where h is a hash function from elements
        to {1, 2, ..., N} drawn from a suitable hash family.
    Start with the best static tree as if there were v_i accesses to element i.
    Let ρ = ⟨a_1, a_2, ..., a_T⟩ be the access sequence.
    For each access a_t = lookup(i)
        Set r_i = r_i + 1;
        If (f_i + r_i ≥ v_i)
            Set v_i = v_i + N;
            For (i = 1, 2, ..., n)
                Set f_i = f_i + r_i and r_i = 0;
            Change trees to the best static tree if there were v_i accesses
            to each element i.
```

Figure 6.1: Pseudocode for Kalai and Vempala's "lazy leading tree" [KV05], slightly modified from the original. For Theorem 31, N is set to $\sqrt{T/n}$.

specifying what historical information is stored. A variant that stores only $(f_i + r_i)$ instead of both f_i and r_i would store strictly less historical information, though characterizing exactly what history this variant stores is slightly more complex; rather than the function $e \mapsto f_e$ indicating the number of accesses of each element at the time of the last tree modification, this variant would store an equivalence class of possible candidates for the function, of the form $\{f : \forall i \; v_i - N \leq f(i) < v_i\}$.

6.2.2 Achieving the Working Set Bound

The main feature of the working set bound is that as the number of accesses m increases beyond $n \log n$, the average amortized cost of a $\text{lookup}(e)$ is $O(\log k)$ where there were k distinct elements accessed since the last $\text{lookup}(e)$ operation. Standard binary search trees would typically guarantee a time bound of $O(\log n)$ in this scenario. Given a binary search tree abstract data type storing n elements $\{1, 2, \ldots, n\}$, we will allow the implementation to reveal the order in which the elements were last accessed. For example, if $n = 4$ and the sequence of accesses was $\langle 4, 1, 3, 2, 4, 2, 3, 2 \rangle$ the revealed history would be $\langle 2, 3, 4, 1 \rangle$ because 2 is the most recently accessed element (last accessed during operation $\#8$) followed by 3

(operation #7), 4 (operation #5), and finally 1 (operation #2). We describe the revealed history formally in the next paragraph. The constructions of this section will support lookup, insert, and delete operations, but not join and split. We prove the following theorem.

Theorem 32. *There is a data structure for the binary search tree abstract data type that reveals only the history (H, ψ) described below, supports* insert *and* delete *operations in expected $O(\log n)$ time, and* lookup(e) *in expected $O(\log k)$ time, where there were k distinct elements in the tree accessed since the last access of e. Moreover, the data structure requires only linear space and $O(\log n)$ random bits.*

The Revealed History

Let U be the universe of elements. Let H, the set of histories, consist of all permutations of all subsets of U. We use the convention that a permutation of S is a sequence in which each element of S appears exactly once. We let h_0 be the empty permutation on the empty set. We define the historical transition function ψ by describing how to evaluate it case by case. Let $h = \langle u_1, u_2, \ldots, u_j \rangle$ be the current history.

- $\psi(v, \text{insert}(e), h)$: If $e \notin \{u_1, u_2, \ldots, u_j\}$, append e to the front of h to obtain the new current history $\langle e, u_1, \ldots, u_j \rangle$. If $e \in \{u_1, u_2, \ldots, u_j\}$ so that $e = u_i$, delete u_i from h and then append e to the front of h to obtain the new current history $\langle e, u_1, \ldots, u_{i-1}, u_{i+1}, \ldots, u_j \rangle$.

- $\psi(v, \text{delete}(e), h)$: Remove e from h to obtain the new current history. Thus if $e = u_i$, the new current history is $\langle u_1, \ldots, u_{i-1}, u_{i+1}, \ldots, u_j \rangle$.

- $\psi(v, \text{lookup}(e), h)$: If e is among the stored elements, that is, $e \in \{u_1, \ldots, u_j\}$, then $\psi(v, \text{lookup}(e), h) = \psi(v, \text{insert}(e), h)$. Otherwise $\psi(v, \text{lookup}(e), h) = h$.

The Construction

Our construction is a slight variant of the *working-set structure* of Bădoiu *et al.* [BCDI07]. The main difference is that we use uniquely represented binary search trees and linked lists instead of conventional versions of those data structures. We include the construction and its analysis for completeness.

Let $h = \langle u_1, u_2, \ldots, u_n \rangle$ be the current history. Define an infinite sequence $\{s_i\}_{i \geq 0}$ via $s_0 = 0$ and $s_i = 2^{2^i}$ for all $i \geq 1$. We will maintain a collection of τ uniquely represented binary search trees $\{T_1, \ldots, T_\tau\}$ and τ uniquely represented linked lists $\{L_1, \ldots, L_\tau\}$ where $\tau := \min\{i : s_i \geq n\} = \lceil \log \log n \rceil$. We maintain the invariant that T_i and L_i store $\{u_j : s_{i-1} < j \leq s_i\}$ for all $1 \leq i \leq \tau$. In each tree T_i the elements will be stored in binary search tree order according to their natural ordering, whereas in each L_i the elements will be ordered according to the current history. We cross-link the nodes (with labels rather than actual pointers), so that given $e \in T_i$ we can find $e \in L_i$ and vice-versa in expected constant time. We also store the total number of elements being stored, and the number of trees τ being stored in each collection.

Before discussing how to implement the operations, we note some properties of this construction. Observe that $s_{i+1} = s_i^2$ for $i \geq 1$ so that the expected depth of any element in T_{i+1} should be about twice the expected depth of any element in T_i, assuming $1 \leq i < \tau - 1$. Note also that the space usage is linear, since the each element is stored in one uniquely represented binary search tree and one uniquely represented linked list. We implement the operations as follows.

Lookup: To lookup element u, run $\mathrm{lookup}(u)$ in each tree $T_i \in \{T_1, \ldots, T_\tau\}$ in increasing order of i. If u is not found in any T_i, report that u is not present. Otherwise u is present in the data structure. In this case, let $i(u)$ be the index of the tree in $\{T_1, \ldots, T_\tau\}$ containing u. Thus the $\mathrm{lookup}(u)$ operation is successful in $T_{i(u)}$ and the correct output is the result of this lookup. However, before returning this result we must do some work to maintain the invariants for $\{T_1, \ldots, T_\tau\}$ and $\{L_1, \ldots, L_\tau\}$. Let $h = \langle u_1, u_2, \ldots, u_n \rangle$ be the current history *before* the performance of the current lookup of u. Insert u into T_1 and at the front of L_1. Delete u from $T_{i(u)}$ and $L_{i(u)}$, using u's location in $T_{i(u)}$ to find its location in $L_{i(u)}$. For $i = 1, 2, \ldots, i(u) - 1$, let v_i be the last element of L_i (i.e., the least recently accessed element in L_i), delete v_i from T_i and L_i and insert v_i into T_{i+1} and at the front of L_{i+1}. Finally, return u and its auxiliary data, if any.

Insertion: Consider operation $\mathrm{insert}(u, d)$. If u already exists in the data structure, then this operation merely updates its auxiliary data to d. This can be achieved exactly as a lookup operation, with the minor modification that the auxiliary data of u should be set to d at the end of the operation instead of returning u and its auxiliary data.

So suppose u does not already exist in the data structure. Insert u into T_1 and

at the front of L_1. T_1 can also store d with u. For $i = 1, 2, \ldots, \tau - 1$, let v_i be the last element of L_i (i.e., the least recently accessed element in L_i), delete v_i from T_i and L_i and insert v_i into T_{i+1} and at the front of L_{i+1}. If the number of elements, n, now equals $s_\tau + 1$ then it is time to add new (initially empty) structures $T_{\tau+1}$ and $L_{\tau+1}$ to the collections. In this case, define v_τ analogously to v_1 through $v_{\tau-1}$, delete v_τ from T_τ and L_τ, and insert v_τ into $T_{\tau+1}$ and $L_{\tau+1}$.

Deletion: Consider operation $\mathrm{delete}(u)$, where we assume u is present in the data structure. For each $i \in \{1, 2, \ldots, \tau\}$ search for u in T_i. Suppose $u \in T_{i(u)}$. Delete u from $T_{i(u)}$. For $i = i(u) + 1, \ldots, \tau$, let v_i be the first element of L_i (i.e., the most recently accessed element in L_i), delete v_i from T_i and L_i and insert v_i into T_{i-1} and at the end of L_{i-1}. If the number of elements, n, now equals $s_{\tau-1}$ then we must also delete (the now empty) structures T_τ and L_τ.

We now proceed to prove Theorem 32 on page 161.

Proof of Theorem 32: We have already argued above that the space usage is linear, since each element is stored once in a uniquely represented linked list and once in a uniquely represented binary search tree and by Proposition 2 on page 52 and Theorem 7 on page 59 the space usage of these structures is linear. The fact that only $O(\log n)$ random bits are required likewise follows from Proposition 2 and Theorem 7. So consider the running times.

We begin with the expected $O(\log k)$ bound for a $\mathrm{lookup}(u)$ operation, where where there were k distinct elements in the tree accessed since the last access of u. Note that for our purposes accesses include insertions and updates to auxiliary data. If u is not present, then the cost of the operation is the cost to perform $\mathrm{lookup}(u)$ operations in each tree in $\{T_1, \ldots, T_\tau\}$. This takes expected $O(\sum_{i=1}^{\tau} \log |T_i|)$ time, however for $1 < i \leq \tau$ we have

$$|T_i| \leq 2^{2^i} - 2^{2^{i-1}}.$$

This in turn implies $\log |T_i| \leq 2^i$, so that

$$\sum_{i=1}^{\tau} \log |T_i| \;\leq\; 2^{\tau+1} \;\leq\; 4 \log(n).$$

Thus $O(\sum_{i=1}^{\tau} \log |T_i|)$ is $O(\log n)$. However, in this case $k = n$, so the operation takes expected $O(\log k)$ time as promised.

So suppose u is present in tree $T_{i(u)}$. In this case we must perform one search, insertion, and deletion within T_i as well as one insertion and deletion within L_i

for each $i \in \{1, 2, \ldots, i(u)\}$. Since the operations on lists take expected $O(1)$ time, the running time is dominated by the tree operations. Each such operation takes expected $O(\log |T_i|)$ time, so the total expected time is $O(\sum_{i=1}^{i(u)} \log |T_i|)$. From our comments above,

$$\sum_{i=1}^{i(u)} \log |T_i| \leq 2^{i(u)+1}$$

and we claim this is $O(\log k)$. From the definition of k and $i(u)$, we may infer $s_{i(u)-1} < k \leq s_{i(u)}$. If $i(u) = 1$ the claim is trivial, so suppose $i(u) > 1$. In this case, $2^{2^{i(u)-1}} < k$ so that $2^{i(u)-1} \leq \log(k)$. Therefore $2^{i(u)+1} \leq 4 \log(k)$ and $\sum_{i=1}^{i(u)} \log |T_i|$ is bounded at $O(\log k)$.

The running time for $\mathrm{insert}(u, d)$ operations that update the auxiliary data for u rather than inserting a new element u likewise take $O(\log k)$ time, by the same analysis as for lookup operations. The running times for $\mathrm{insert}(u, d)$ operations that add a new element u to the set, and likewise for $\mathrm{delete}(u)$ operations, can be bounded via an analysis analogous to the one above for $\mathrm{lookup}(u)$ operations where u is not in the set of elements being stored. We omit the precise details.

Finally, we prove that the only history revealed is the order of last accesses to the elements, that is, the current history h defined by (H, ψ) above. Note that once the random bits of the uniquely represented binary search trees and the memory allocator have been fixed, the machine representation of the data structure is a deterministic function of h. Also note that the h is in fact stored by the data structure, as it can be computed by appending the lists in $\{L_1, \ldots, L_\tau\}$. Together these observations imply that h is *precisely* the historical information that is revealed by the machine state. ∎

Chapter 7

Lower Bounds

In previous chapters we have shown that many fundamental abstract data types have efficient uniquely represented implementations. Indeed, after seeing those results one is tempted to ask if there is a general theorem stating that the overhead for unique representation is extremely low for *all* abstract data types. Unfortunately that is not the case. In this chapter we present the first complexity gap between uniquely represented and conventional implementations of a natural abstract data type in the RAM model, and shed some light on a particular barrier that uniquely represented implementations face and conventional implementations do not.

Our lower bounds hold in the *cell probe* model [Yao81]. The cell probe model assumes a RAM is used for computation, however it assumes a more lenient cost model, in which the cost (time complexity) of a computation is the number of word reads and writes. All other computations are free. The word size w is a parameter of the model. This more lenient cost model implies that lower bounds in the cell probe model hold for the RAM as well; lower bounds for the cell probe model are thus stronger than corresponding results for the RAM.

7.1 The Information-Destruction Bound

As mentioned, uniquely represented data structures are strongly history independent. This means that no evidence of the historical sequence of operations that led to the current logical state may be stored in any way. However, some abstract data type operations might destroy significant amounts of historical information, in ways we will make precise shortly. Destroying information is expensive for all

implementations, however uniquely represented implementations must do so eagerly whereas conventional implementations need not do so at all. This is the basis of what we will call the *information-destruction bound*.

To formalize what we mean by the destruction of information, consider the state transition graph of a RAM $\mathcal{M}$. Recall from Section 2.2 that the state transition graph has a vertex for each configuration of the RAM, and an edge (u, v) for each u and v such that a single RAM operation can take the machine from configuration u to configuration v. Fix an abstract data type $\mathcal{A}$ and a uniquely represented implementation of $\mathcal{A}$ on $\mathcal{M}$. For a logical state v, let $\sigma(v)$ denote the unique machine state in $\mathcal{M}$ encoding v. Consider some fixed operation of $\mathcal{A}$, which we identify with a function f of type $V_{\mathcal{A}} \to V_{\mathcal{A}}$, where $V_{\mathcal{A}}$ is the set of logical states of $\mathcal{A}$. Informally we say applying f *destroys information* if we are in logical state v, and there exists logical state $u \neq v$ such that $f(u) = f(v)$. A measure of the information destroyed is $|f^{-1}(f(v))|$, where $f^{-1}(w) := \{u : f(u) = w\}$. Given a logical state u, we call $|f^{-1}(u)|$ the *in-degree* of f at u. We also call this quantity the in-degree of the operation corresponding to f. The basic idea of the lower bound is then to show that if the in-degree of an operation is large enough, then the RAM implementation will have to perform many operations to transform the machine from a configuration appropriately chosen from $\{\sigma(u) : u \in f^{-1}(v)\}$ to the configuration $\sigma(v)$. The proof of this fact then relies on a volumetric argument; essentially, the set of states that are near some machine state σ in the state transition graph must be small, so any large set of states must contain many states that are far from σ.

Fix a directed graph $G = (V, E)$ with positive edge costs. For $u, v \in V$ let $d_G(u, v)$ be the minimum cost of any u to v path in G (where the cost of a path P is the sum of costs for all edges in P). Let $\mathrm{Ball}_G(v, r)$ denote the closed in-ball of radius r around v, i.e., $\{u : d_G(u, v) \leq r\}$. In the state transition graph of a RAM, $d_G(u, v)$ is the minimum number of words one needs to change to convert u to v, where u and v are configurations of the word array of the RAM. The above line of reasoning leads to the following lemmata.

Lemma 14. *Let $G = (V, E)$ be the state transition graph of a RAM with w bit words and with a word array of length m. Then for any $v \in V$ and $r \in [0, m]$,*

$$|\,\mathrm{Ball}_G(v, r)| \;=\; \sum_{k=0}^{r} \binom{m}{k} (2^w - 1)^k \;\leq\; m^r \cdot 2^{w \cdot r}$$

and if $r \leq m/3$ then $\binom{m}{r}(2^w - 1)^r \leq |\,\mathrm{Ball}_G(v, r)| \leq 2\binom{m}{r}(2^w - 1)^r$.

Proof: In G, a path from u to v with k edges can be uniquely specified by selecting k words from the word array and selecting new values for the k words. There are $\binom{m}{k}$ choices for words, and for each word, $2^w - 1$ possible word values that differ from the current one. Thus there are $\binom{m}{k}(2^w-1)^k$ paths of length k starting from any machine state v. Summing over k yields the formula for $|\operatorname{Ball}_G(v,r)|$.

We next prove that $|\operatorname{Ball}_G(v,r)| \leq m^r \cdot 2^{wr}$. This inequality holds because we can specify any element of $u \in \operatorname{Ball}_G(v,r)$ given v by specifying a list L_0 of r array indices including all those modified in the path from u to v (and possibly some others), and a list L_1 of the r word values that those indices store in state u. There are $m^r \cdot 2^{wr}$ such pairs of paths (L_0, L_1), so this is an upper bound on $|\operatorname{Ball}_G(v,r)|$.

Finally, consider the case that $r \leq m/3$. Given the exact formula for $|\operatorname{Ball}_G(v,r)|$, $\binom{m}{r}(2^w-1)^r \leq |\operatorname{Ball}_G(v,r)|$ trivially. To prove $|\operatorname{Ball}_G(v,r)| \leq 2\binom{m}{r}(2^w-1)^r$, it suffices to prove that $\binom{m}{k-1}(2^w-1)^{k-1} \leq \frac{1}{2}\binom{m}{k}(2^w-1)^k$ for all $k \leq m/3$, in which case $|\operatorname{Ball}_G(v,r)| \leq \sum_{k=0}^{r} 2^{k-r}\binom{m}{r}(2^w-1)^r \leq 2\binom{m}{r}(2^w-1)^r$. However,

$$\frac{\binom{m}{k-1}(2^w-1)^{k-1}}{\binom{m}{k}(2^w-1)^k} \;\leq\; \frac{k}{(m-k+1)\cdot(2^w-1)} \;\leq\; \frac{m/3}{(2m/3)\cdot(2^w-1)} \;\leq\; \frac{1}{2}.$$

■

Lemma 15. *Let $G = (V, E)$ be the state transition graph of a RAM with w bit words and with a word array of length m, so that V is the set of configurations the word array can take. For $u, v \in V$ let $d_G(u,v)$ be the minimum number of words one needs to change to convert u to v. Fix any $S \subseteq V$ and $v \in V$. Then*

$$\max_{u \in S}\left(d(u,v)\right) \geq \frac{\log|S|}{w + \log(m)} - 1$$

and in expectation when sampling u uniformly at random from S,

$$\mathbf{E}_{u \sim S}\left[d(u,v)\right] \geq \left(1 - \frac{1}{m \cdot 2^w}\right)\left(\frac{\log|S|}{w+\log(m)} - 1\right).$$

Proof: We first address the worst-case bound concerning $\max_{u\in S}(d(u,v))$. Clearly, if $|\operatorname{Ball}_G(v,r)| < |S|$ then $\max_{u\in S}(d(u,v)) > r$. So it suffices to prove $|\operatorname{Ball}_G(v,r)| < |S|$ when $r = \frac{\log|S|}{w+\log(m)} - 1$. From Lemma 14 we know that $|\operatorname{Ball}_G(v,r)| \leq m^r 2^{wr} = (m \cdot 2^w)^r$, and we have set $r = \log_b|S| - 1$ where the base $b = m \cdot 2^w$. This implies $|\operatorname{Ball}_G(v,r)| \leq |S|/(m \cdot 2^w) < |S|$.

Now consider $\mathbf{E}_{u\sim S}[d(u,v)]$. Note that

$$\begin{aligned}\mathbf{E}_{u\sim S}[d(u,v)] &= \frac{1}{|S|}\sum_{r=0}^{m}(|S| - |S\cap \mathrm{Ball}_G(v,r)|) &(7.1)\\ &\geq \frac{1}{|S|}\max_{r\in\mathbb{N}}((r+1)\cdot(|S| - |S\cap \mathrm{Ball}_G(v,r)|)) &(7.2)\end{aligned}$$

Using $|\mathrm{Ball}_G(v,r)| \leq m^r 2^{wr}$, we set $r = \left(\frac{\log \epsilon|S|}{w+\log(m)}\right) - 1$ and infer $|\mathrm{Ball}_G(v,r)| \leq \epsilon|S|$, $|S| - |S\cap \mathrm{Ball}_G(v,r)| \geq (1-\epsilon)|S|$ and

$$\mathbf{E}_{u\sim S}[d(u,v)] \geq (1-\epsilon)\left(\frac{\log \epsilon|S|}{w+\log(m)}\right) \tag{7.3}$$

Setting $\epsilon = 1/(m\cdot 2^w)$ then yields

$$\mathbf{E}_{u\sim S}[d(u,v)] \geq \left(1 - \frac{1}{m\cdot 2^w}\right)\left(\frac{\log|S|}{w+\log(m)} - 1\right) \tag{7.4}$$

as claimed. ∎

We put Lemma 15 to good use in the lower bound proofs of the following sections. We will also make use of *Yao's Minimax Principle* (see [MR95] for a treatment).

Theorem 33 (Yao's Minimax Principle [Yao77])**.** *Let $\mathcal{A}$ be a finite set of algorithms for some problem Π, and let $\mathcal{I}$ be a finite set of instances of Π. Let $\mathcal{D}_\mathcal{A}$ be the set of all distributions over $\mathcal{A}$ and let $\mathcal{D}_\mathcal{I}$ be the set of all distributions over $\mathcal{I}$. For $\pi \in \mathcal{D}_\mathcal{A}$, let A_π denote a random algorithm drawn from $\mathcal{A}$ according to distribution π, and similarly for $\pi \in \mathcal{D}_\mathcal{I}$ let I_π denote a random instance drawn from $\mathcal{I}$ according to distribution π. Fix any function $c : \mathcal{A}\times\mathcal{I} \to \mathbb{N}$, where $c(A,I)$ denotes the cost of running algorithm A on instance I. Then*

$$\max_{\pi\in\mathcal{D}_\mathcal{I}}\min_{A\in\mathcal{A}}\mathbf{E}[c(A,I_\pi)] \leq \min_{\pi\in\mathcal{D}_\mathcal{A}}\max_{I\in\mathcal{I}}\mathbf{E}[c(A_\pi,I)]$$

Observe that $\max_{\pi\in\mathcal{D}_\mathcal{I}}\min_{A\in\mathcal{A}}\mathbf{E}[c(A,I_\pi)]$ is the worst-case performance of the best deterministic algorithm A in $\mathcal{A}$ (where A may be tailored to the distribution π) on a distribution from $\mathcal{D}_\mathcal{I}$. Also observe that $\min_{\pi\in\mathcal{D}_\mathcal{A}}\max_{I\in\mathcal{I}}\mathbf{E}[c(A_\pi,I)]$ is the worst case performance of the best randomized algorithm (in the set $\{A_\pi : \pi\in\mathcal{D}_\mathcal{A}\}$) on an input selected from $\mathcal{I}$. With these observations in mind, Yao's minimax principle yields Corollary 2.

Corollary 2. *Fix any function $s : \mathbb{N} \to \mathbb{N}$, an abstract data type, and some (possibly infinite) set $\hat{\mathcal{A}}$ of deterministic data structures implementing that abstract data type. If for some input distribution every deterministic data structure in $\hat{\mathcal{A}}$ that uses at most $s(n)$ space takes at least $f(n)$ time in expectation to perform some operation, then every randomized algorithm that uses at most $s(n)$ space and always terminates requires at least $f(n)$ time in expectation to perform that operation in the worst case.*

The first expectation mentioned above is computed over the input distribution, whereas the second is computed over the random bits of the randomized algorithm. A meticulous reader might be concerned with the finiteness conditions for $\mathcal{A}$ and $\mathcal{I}$, and how they are absent from the statement of Corollary 2. We offer the following proof sketch.

Proof: We provide a proof by contradiction. Consider a counterexample to Corollary 2. Fix an operation and let $c(A, I)$ be the time to run deterministic algorithm A that performs that operation on input I. The counterexample must include a randomized algorithm A_π for the operation such that

$$\max_{I \in \mathcal{I}} \mathbf{E}[c(A_\pi, I)] < f(n_0)$$

where $\mathcal{I}$ is the set of instances of size $n_0 \in \mathbb{N}$ and A_π uses at most $s(n_0)$ words of space. In other words, A_π must have an expected performance guarantee strictly better than $f(n_0)$ for all inputs in $\mathcal{I}$. Let $\mathcal{A} \subseteq \hat{\mathcal{A}}$ be the set of deterministic algorithms that make up the *support* of A_π (i.e., the algorithms that A_π has positive probability of executing).

We claim that the support of A_π must be finite. This is because the RAM has finitely many configurations, namely 2^{ws} where w is the word size and s is the length of the word array, and this is the entire state of the system. Let $G = (V, E)$ be state transition graph of the RAM. Each program p can be modeled as a set of paths $\{P_v : v \in V_p\}$ where P_v is the execution trace of p when starting from state $v \in V_p$ and $V_p \subseteq V$ is the set of valid starting states for program p. (The execution trace is the sequence of machine configurations p will take the machine through as it executes. We assume that for each abstract data type operation the data structure ensures the RAM is in a valid starting state for the program implementing that operation whenever it is called.) Note that if there is a cycle in P_v then p does not terminate when starting from v, since it will repeatedly transition through the cycle forever. Since the randomized algorithm always terminates when started from a valid starting state by assumption, these paths must all be simple. This implies that the number of programs is finite. Note that this argument is independent of

whatever read-only objects are present in the RAM, including the finite program itself and the infinite sequence of random bits, which allows us to conclude that the support of A_π is finite as claimed.

Finally, the counterexample to Corollary 2 must also ensure that there exists some input distribution $\pi \in \mathcal{D}_\mathcal{I}$ such that every deterministic data structure in $\hat{\mathcal{A}}$ using at most $s(n)$ space takes $f(n)$ time in expectation to perform the operation on inputs sampled from π. This implies

$$\max_{\pi \in \mathcal{D}_\mathcal{I}} \min_{A \in \mathcal{A}} \mathbf{E}[c(A, I_\pi)] \geq f(n_0).$$

However, this contradicts Yao's minimax principle, using $\mathcal{I}$, $\mathcal{A}$, and c as defined above. Note that with $s(n)$ set sufficiently large (e.g., $s(n) = 2^{2^n}$) the space usage constraint is not particularly restrictive. ■

Extensions to Data Structures that are Nearly Uniquely Represented. Note that the proof of Lemma 15 easily extends to give the following lower bounds:

$$\max_{u \in S} \min_{v \in T} d(u, v) \geq \frac{\log(|S|/|T|)}{w + \log(m)} - 1$$

and

$$\mathbf{E}_{u \sim S}\left[\min_{v \in T} d(u, v)\right] \geq \left(1 - \frac{1}{m \cdot 2^w}\right)\left(\frac{\log(|S|/|T|)}{w + \log(m)} - 1\right).$$

These bounds are useful to prove lower bounds for data structures that have a small number of machine representations for any particular abstract data type state, if we interpret T as the set of machine states representing some abstract data type state. This can be used to lower bound the number of machine representations (and thus the number of bits of historical information) that an implementation must have to circumvent the lower bound arguments in the following sections. The calculations are quite straightforward and we omit the details.

7.2 Union-Find

The *union-find* abstract data type, also known as the *disjoint-set* abstract data type, maintains a labeled partition on a set of elements U under the following operations:

- $\text{union}(A, B, C)$: Given labels A and B of sets in the partition and a label C, merge the two sets with labels A and B into a single set with label C. (To maintain the invariant that different sets of the partition have different labels, C may not be equal to the labels of any other sets in the partition, with the exception of A and B.)

- $\text{find}(u)$: Given $u \in U$, return the label of the set of the partition containing u.

The union-find abstract data type was introduced by Galler and Fisher [GF64], though they did not require the user to specify labels for the sets. Their approach is to store each set as a rooted tree, and store the label of the set at the root. Let T_L be the rooted tree with label L. To perform $\text{union}(A, B, C)$, simply make the root of T_A a child of the root of T_B and change the label of the root of T_B to C. This approach can be improved by applying two heuristics, *path compression* and *union by rank*. Path compression is a postprocessing step after a $\text{find}(u)$ operation, in which each vertex v on the path from u to the root r of the tree containing u has its parent changed to r. Union by rank simply involves making the smaller of T_A and T_B the subtree of the larger when performing a $\text{union}(A, B, C)$ operation. Tarjan [Tar75] proved that the resulting data structure performs $m \geq n$ finds and $n - 1$ unions in $\Theta(m\alpha(m, n))$, where $\alpha(x, y)$ is an inverse of *Ackermann's function* defined below.

Definition 13 (Ackermann's Function and Inverse)**.** *Define Ackermann's function*[1] $A : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ *via*

$$A(0, x) = 2x \ \ \forall x \in \mathbb{N}, \quad A(x, 0) = 0 \ \ \forall x \in \mathbb{N}$$
$$A(x, 1) = 2 \ \ \forall x \in \mathbb{N}, \quad A(x, y) = A(x - 1, A(x, y - 1)) \ \ \forall x \geq 1, y \geq 2$$

Define Ackermann's inverse by $\alpha(x, y) := \min\{k : A(x, k) \geq \log(y)\}$.

Fredman and Saks [FS89] proved that the $\Theta(m\alpha(m, n))$ bound is optimal in the cell probe model with $w = O(\log n)$. This result subsumed several earlier lower bounds in weaker computational models [Pou96, Tar79]. The above implementation of the union-find abstract data type is therefore in some sense optimal. Unions take constant time, and m finds, if m is sufficiently large, take $O(m\alpha(m, n))$ time. The function $n \mapsto \alpha(m, n)$ grows exceedingly slowly for even modest values of m. For example, Tarjan points out that $\alpha(2, n) = O(\log^*(n))$. Theorem 34 shows that the picture for uniquely represented implementations is quite different.

[1]Unfortunately, there are various inequivalent definitions of Ackermann's function. We use the definition in [Tar75].

Theorem 34. *In the cell probe model with word size w, any uniquely represented implementation of union-find on n elements and using at most s space must take expected $\Omega(\frac{n}{w+\log(s)})$ time per* union *operation in the worst case. In the typical case of $w = O(\log n)$ and $s = \text{poly}(n)$, each* union *operation takes expected $\Omega(\frac{n}{\log n})$ time in the worst case.*

Proof: Let $U = \{1, 2, \ldots, n\}$ and assume n is even. Let $G = (V, E)$ be the state transition graph of a RAM with word size w and s words of memory. Let P be the set of partitions of U into two nonempty sets, and let $V_P \subset V$ be the set of machine states corresponding to partitions in P.

Let $\hat{u} \in V$ be the unique state obtained after all of U is merged into one set (with some arbitrarily fixed label). Ignoring the labels, there are $|P|$ different states v from which we could reach $\hat{u}$ via a single union operation, and which contain two non-empty subsets of U. Lemma 15 then guarantees that if we sample v uniformly at random from V_P, then

$$\mathbf{E}[d_G(v, \hat{u})] \geq \left(1 - \frac{1}{s \cdot 2^w}\right)\left(\frac{\log |V_P|}{w + \log(s)} - 1\right)$$

Define a distribution π on union-find instances that uniformly samples an initial state from P and then asks the data structure to perform a union to merge the two partitions into one set containing all elements. Clearly, any deterministic uniquely represented implementation must take the machine to state $\hat{u}$, which will cost at least $d_G(v, \hat{u})$ starting from state v in the cell probe model. Let $\hat{\mathcal{A}}$ be the set of all deterministic uniquely represented implementations of the union-find abstract data type. Corollary 2 then implies that any randomized uniquely represented implementation of union-find using only s space in a RAM with word size w makes at least $\left(1 - \frac{1}{s \cdot 2^w}\right)\left(\frac{\log |V_P|}{w+\log(s)} - 1\right)$ word writes in expectation. Since $|V_P| = 2^{|U|} - 2$, this implies the claimed lower bound of $\Omega(\frac{|U|}{w+\log(s)})$. ∎

Amortized Bounds. A natural question to ask is if there is an amortized lower bound for a sequence of union operations, perhaps something like $\Omega(\frac{|U|^{1+\epsilon}}{w+\log(s)})$ for some $\epsilon > 0$. However, it is not too hard to see that this is not possible. We can develop a uniquely represented implementation of the union-find abstract data type by storing each set as a treap in which each node stores the label of the root of the tree that contains it, each root stores the set-label of its tree, and node labels are generated via the hash-consing technique of Section 5.2. If we implement unions

by inserting the elements of the smaller set, one at a time, into the treap storing the larger set then each node is inserted into a treap at most $\lfloor \log |U| \rfloor$ times, and each insertion takes expected $O(\log |U|)$. By this reasoning the total time to perform a sequence of unions generating a set $S \subseteq U$ is at most $O(|S| \log^2 |S|)$ in expectation. This implementation also supports expected $O(1)$ time find operations. We have proved the following theorem.

Theorem 35. *There exists a uniquely represented implementation of a bounded union-find structure of capacity N whose memory is statically allocated and stores elements of a fixed size, such that*

- *the* union *operation takes expected $O(|A| \log |B|)$ time to merge two sets A and B with $|A| \leq |B|$, and the total time to perform any sequence of* union *operations that creates a set S starting from $\{\{e\} : e \in S\}$ is $O(|S| \log^2 |S|)$ in expectation,*
- *the* find *operation takes expected $O(1)$ time,*
- *the implementation requires $O(\log N)$ random bits, and*
- *the implementation uses only $O(N)$ space.*

We show in Chapter 8 how to remove the assumptions that the size of the data structure is bounded and the memory is is statically allocated.

7.3 Meldable Heaps

In Section 4.6 we presented uniquely represented heaps, and in particular described a good uniquely represented analogue of a binary heap. However, our heap did not support fast merges, as binomial heaps and Fibonacci heaps can (see table 4.5 on page 63). The following theorem provides justification for this failure; building a uniquely represented heap that supports fast merges is impossible in a RAM.

Theorem 36. *In the cell probe model with word size w, any uniquely represented implementation of meldable heaps on n elements and using at most s space must take expected $\Omega(\frac{n}{w+\log(s)})$ time per* merge *operation in the worst case. In the typical case of $w = O(\log n)$ and $s = \mathrm{poly}(n)$, each* merge *operation takes expected $\Omega(\frac{n}{\log n})$ time in the worst case.*

Proof: The proof of Theorem 36 is essentially the same as the proof of Theorem 34 in Section 7.2. The only difference is that V_P should now be the set of machine states corresponding to two nonempty heaps that together contain all n elements. ■

Chapter 8

Putting It All Together: Uniquely Represented Systems

In this, the concluding chapter, we discuss how to convert the statically memory-allocated uniquely represented data structures from previous chapters into dynamically memory-allocated versions, and how to compose these data structures into uniquely represented computational systems. We also discuss various additional considerations when designing such systems, and directions for future work.

8.1 Dynamic Memory Allocation

We have made certain assumptions about the uniquely represented data structures of the previous chapters, namely that

- they have bounded size, and
- they have a block of memory statically allocated to them on initialization.

In this section we will discuss how to remove these assumptions.

As discussed in Section 3.3, we use a uniquely represented hash table to serve as a memory allocator, assuming the data structure *components* have hashable labels. Components refer to constant sized parts of the data structure, for example graph nodes, integer variables, and array slots. These labels can be generated using a variety of methods, including *hash-consing* (see Section 5.2), and are essential to map pointer structures into the one dimensional memory array provided by both the RAM model of computation and real machines.

If desired, the labeling schemes that we have provided for each construction in this dissertation can be augmented to further differentiate components from different data structures as follows. We assume that each data structure has a unique *name* that is strongly history independent; that is, the name of a data structure does not depend on the sequence of operations. Then a component labeled l belonging to a data structure with name η can be labeled $f(\eta, l)$ for some function f such as concatenation. Note that this is strictly optional, and in some cases it might be advantageous not to do so. For example, if we have a collection of binary search trees that have many subtrees in common, then hash-consing provides a way to store a single copy of each distinct subtree and thus save space. In any event, we will have to be wary of two inequivalent components[1] receiving the same label, and take the steps described in Section 5.2.2 to ensure that inequivalent components receive different labels.

Given these component labels, we present two ways to dynamically allocate memory for data structures in a uniquely represented manner. We describe each in turn. In both cases, a uniquely represented array A of *blocks* is implemented on top of the RAM memory, where a block is any fixed number of words. This array is statically allocated the entire RAM memory. A uniquely represented hash table (see Chapter 3) is implemented on top of A and serves as the memory allocator.

One-Level Allocation. In its most basic form, the memory allocator stores in each block a single component and its label. When a data structure with name η requires more space, it issues a request for k more components worth of space, for some $k \in \mathbb{N}$. When requesting the space, the data structure also provides k labels $\{l_1, \ldots, l_k\}$. The memory allocator then assigns these labels to the data structure. Reads and updates take place as if the data structure had control of the entire RAM memory though a uniquely represented hash table interface, however the hash table ensures that no two inequivalent components receive the same label, as discussed above and in Section 5.2.2.

In this scheme, unlike the one described in the next paragraph, there is only one level of uniquely represented hashing. The price we pay for this is that locality of reference is completely destroyed.

[1]For pointer machines, inequivalent with respect to the $\approx$ equivalence relation of Section 5.2.2. More generally, inequivalent in the sense that it is essential to store both components separately.

Two-Level Allocation. Assume each component belongs to a named data structure. When a data structure with name η requires more space, it issues a request for k more words, for some $k \in \mathbb{N}$. Suppose each block has β words, so that k words amounts to $k' := \lceil k/\beta \rceil$ blocks. The memory allocator keeps track of the number of blocks b_η that η is using, and *assigns* keys $\{\eta \oplus (b_\eta + i) : i \in [1 : k']\}$ to be under the control of η, where $\oplus$ is the concatenation operator. The data structure η may then assume it has control of k' additional blocks, and when reading or writing to the i^{th} block under its control, it (transparently) is actually reading or writing to the block that $\eta \oplus i$ hashes to. Of course, η is stored in a uniquely represented manner within the blocks B_η assigned to it. The blocks under η's control can then be treated as a word array of length $\beta \cdot |B_\eta|$, where the block with key $\eta \oplus i$ for $i \in [1 : |B_\eta|]$ contains slots with indices from $(i-1)\beta$ to $i \cdot \beta - 1$. This word array then serves as the foundation for the uniquely represented constructions of previous sections.

We call this style of allocation *two-level* because there is a system-wide memory allocator assigning blocks to data structures, and then within each data structure there is a memory allocator assigning space to each component. This style of allocation maintains more locality among the components of any particular data structure at the cost of adding another level of hashing. This is likely to improve performance in external memory models of computation and in real systems with memory hierarchies.

8.1.1 Dynamic Resizing

Some data structures, for example arrays and hash tables, have built-in capacities. A standard way to remove the capacities is to dynamically resize the data structure as needed. Thus, an array A with storing n elements might have $2^{\lceil \log n \rceil}$ words of space allocated to it, and as $\lceil \log n \rceil$ changes value the space allocated to A may be cut in half or doubled as required. This approach will not yield good expected performance against any oblivious adversary however. Simply consider the sequence of operations that adds 2^k elements and then repeatedly adds and deletes an element so that the array must resize on every operation. To fix this problem, we choose a random threshold for resizing. In the case of arrays, select real number $r \in [1, 2]$ uniformly at random, define a set of thresholds $T := \{\lceil r2^k \rceil : k \in \mathbb{N}\}$, and ensure that an array storing n elements has exactly $\min\{t : t \in T, t \geq n\}$ words of space. (A nearly identical solution is to ensure that an array storing n elements has exactly $\lceil r \cdot 2^{\lceil \log(n/r) \rceil} \rceil$ words of space allocated to it.) In either case, a fixed insertion or deletion causes the number of elements to cross a threshold with

probability $\Theta(1/n)$; if in this case we spend expected $O(nf(n))$ time to resize the data structure the unconditional expected time only increases by $O(f(n))$.

More precisely, fix any $n \in \mathbb{N}$. We consider an insertion, so that the number of elements increases from n to $n+1$. This causes a resizing event if there exists a k such that $n \leq \lceil r2^k \rceil$ and $n+1 > \lceil r2^k \rceil$. This can only occur for $k = \lfloor \log n \rfloor$. We claim that

$$\mathbf{Pr}\left[n \leq \lceil r2^k \rceil \text{ and } n+1 > \lceil r2^k \rceil\right] \leq \frac{2}{n}.$$

To see this, note that $n \leq \lceil r2^k \rceil$ implies $r > (n-1)/2^k$ and $n+1 > \lceil r2^k \rceil$ implies $r \leq n/2^k$. Thus this event occurs only if $r \in [\frac{n-1}{2^k}, \frac{n}{2^k}]$ which occurs with probability $1/2^k$. However, $k = \lfloor \log n \rfloor \geq \log(n) - 1$ so $1/2^k \leq 2/n$. The case for deletions is proved analogously.

8.1.2 Eager Garbage Collection

For obvious reasons, uniquely represented data structures require *eager garbage collection*. In other words, whenever a part of the data structure is no longer needed, it must be removed (i.e., garbage collected) immediately to put the data structure in the same state as if it had never existed. For a comprehensive treatment of garbage collection, we refer the reader to the textbook of Jones and Lins [JL99]. A suitable way to do this is with a *reference counting* garbage collector. Typically, this type of garbage collector maintains a counter for each object indicating how many *references* to that object exist. References to x include pointers to x, instances of x in the call stack of a program, or any other handles on x. When a reference to an object x is deleted, if there are no references into x then x is deleted and all of the objects x references (e.g., points to) have their counts decremented. This description is something of a simplification; for example we have not considered directed cycles of pointers with no references into the cycle (e.g., x points to y and y points to x, but nothing else references either x or y), but these too must be garbage collected immediately.

It is relatively straightforward to modify a standard reference counting garbage collector to work in a uniquely represented computing environment. Pointers are replaced with abstract pointers (i.e., labels) and deletions must occur via calls to the uniquely represented memory allocator instead of some other way. The whole process must be done eagerly. None of these considerations presents any difficulties. Note that the space overhead for a reference counting garbage collector is a multiplicative constant. Also note that because the time to update counters

can be amortized against the time to create or delete references and the time to delete objects during garbage collection can be amortized against the time to create them in the first place, the time overhead is amortized expected $O(f(n))$ for any operation taking expected $f(n)$ time.

8.1.3 Dynamically Memory Allocated, Unbounded, Uniquely Represented Data Structures

Using the techniques of this section we can obtain the following result in a straightforward manner.

Theorem 37. *Every statically memory allocated uniquely represented data structure of bounded size in this dissertation can be implemented as a dynamically memory allocated uniquely represented data structure of unbounded size with the same amortized expected running time and space guarantees, up to multiplicative constant factors.*

8.2 Composability & Uniquely Represented Systems

The uniquely represented data structures in this dissertation are *composable*, by which we mean that they can be combined together to yield more complex uniquely represented constructions. This can be shown as follows. All of the uniquely represented data structures in this dissertation can be broken up into components, as discussed in Section 8.1. These components are constant sized pieces of the data structure that are given unique labels. There are two tasks implicit in all of our constructions of uniquely represented data structures.

1. Create a set of components that is strongly history independent (and thus uniquely represented), yet supports the requested operations efficiently.

2. Store this set of components in RAM in a strongly history independent (and thus uniquely represented) and efficient manner. That is, provide an efficient strongly history independent dynamic map of components into the one dimensional memory provided by the RAM.

It is not hard to prove that performing these two tasks is sufficient to obtain an efficient uniquely represented implementation for some abstract data type. Now,

fix any collection of uniquely represented data structures $\mathcal{C}$. We can think of $\mathcal{C}$ as implementing some abstract data type $\mathcal{A}$ obtained by combining the abstract data types implemented by the data structures in $\mathcal{C}$. In Section 8.1 we have discussed how to ensure that no two inequivalent components from the set of components obtained from $\mathcal{C}$ receive the same label. Since the labels are generated in a uniquely represented manner, the set L of such labels is a deterministic function of the RAM's random bits and the contents of the data structures in $\mathcal{C}$. Thus if we imagine $\mathcal{C}$ as a single data structure implementing $\mathcal{A}$, we have satisfied the first condition for it. In Section 8.1 we also discussed how to use L to map the components into the one dimensional memory provided by the RAM, essentially by simply using uniquely represented hash tables. Thus we have also satisfied the second condition for obtaining a uniquely represented implementation of $\mathcal{A}$.

The composability of uniquely represented data structures enables the creation of efficient uniquely represented systems, to the extent that the state of a system can be modeled as a collection of data structures in some computational model. Of course, real systems possess various complicating sources of state including

- Hidden CPU registers, e.g., those used for diagnostic purposes
- Cache lines and multilevel memory hierarchies
- Multiple sources of storage, e.g., modern video cards, network cards, printers, and optical drives typically have buffers or other forms of storage.

It is beyond the scope of this dissertation to address how current systems might most easily be modified to make them entirely uniquely represented. The complexity of modern systems means that such a task would require significant engineering effort. However, the constructions we have presented provide the theoretical foundation for the design of efficient uniquely represented systems and applications that store *precisely* the information their designs specify, including

- Filesystems,
- Databases,
- Voting Machines,
- Information kiosks,
- Web Browsers, and
- Advanced file formats.

The main contribution of this dissertation is to show that efficient uniquely represented data structures do indeed exist in the RAM model of computation, and thus there are no significant fundamental barriers to the development of uniquely represented systems. We anticipate other uses for efficient uniquely represented data

structures as well, as articulated in Section 1.1.1.

8.3 Future Work & Open Problems

The results in this dissertation represent a significant increase in our understanding of what is possible with respect to uniquely represented data structures in the RAM model of computation. However, there are several lines of inquiry and future work that remain.

Exploiting Existing Results in Practice

A natural line of future work is to exploit the theoretical results described in this dissertation and implement the applications discussed in Section 1.1.1, including filesystems, databases, voting machines, information kiosks, advanced file formats, web browsers, certification of computations, and faster equality testing for theorem provers and verifiers. There are also questions about how uniquely represented data structures and the ideas behind them can be exploited with minimal modifications to the computational infrastructure that exists today. This perspective provides us with an interesting variant of the secure redaction problem, namely that of purging historical information from files.

Purging Historical Information from Files. This problem involves generating a history independent version of an arbitrary file in some known format. For example, can we create a program that when given an arbitrary document in Adobe PDF format generates a "clean" version that stores *exactly* the information available via the legitimate interface for reading and editing PDFs? Such a program would allow us to solve the file redaction problem for PDFs without having to alter the PDF specification. This would also allow us to use conventional data structures rather than uniquely represented ones when editing the newly cleaned file. One difficulty with this approach is that it may be unclear exactly what the document is intended to store. For example, there may be multiple legitimate interfaces for reading and editing PDFs that reveal different answers to the same queries on some document. However, suppose we fix a particular interface. There is still the problem that the document format may use data structures such as AVL trees [AVL62] or red-black trees [Bay72, GS78] that are not uniquely represented. Simply replacing such data

structures with uniquely represented analogues of them would require changing the document format specification, which we are attempting to avoid in deference to existing software infrastructure. Instead, we need to augment each data structure with a $\mathrm{canonicalize}$ operator that converts it to a canonical form. For example, a canonical form for AVL trees can be obtained by inserting the keys in sorted order into a fresh AVL tree. That is, we ensure that the result is history independent by enforcing a canonical historical sequence of operations to generate the tree. This technique applies to many other search trees and many other data structures as well. Once put in canonical form, we must then map the data structure into the one dimensional byte array provided by a RAM in a canonical manner. This latter task can often be accomplished by storing the components of the data structure (such as nodes in a tree) in sorted order. Thus it appears likely that a principled solution to this version of the file redaction problem exists which does not require modifying existing document formats.

Mapping the Theoretical Terrain in More Detail

For which abstract data types do there exist uniquely represented implementations matching the performance of the best conventional implementations? While we have provided efficient uniquely represented implementations for many of the most widely used abstract data types, there remain several without uniquely represented implementations that match the performance of the best conventional implementations. For example, our constructions for orthogonal range queries (range trees), horizontal point location, orthogonal segment intersection, and dynamic 2D convex hull have gaps of $O(\log \log n)$ or $O(\log n)$ in time and space relative to the best known conventional implementations. We have not addressed certain abstract data types at all, for example counters supporting worst case $O(1)$ time increment, dynamic connectivity, and dynamic minimum spanning tree. Conversely, the information-destruction bound of Section 7.1 provides useful lower bounds for uniquely represented implementations of a few abstract data types, but tells us nothing useful about most of them. Are there other barriers to the development of efficient uniquely represented data structures?

More generally, is there a general result that characterizes which abstract data types have uniquely represented implementations matching the performance of the best conventional implementations?

Another set of questions pertains to the number of random bits needed to get good expected performance. The uniquely represented hash table of Section 3.1 im-

plements the dynamic (unordered) dictionary abstract data type in expected $O(1)$ for all operations using only $O(\log n)$ random bits by exploiting a breakthrough result of Pagh, Pagh, and Ružić [PPR07]. It is an interesting question whether the same running times can be obtained with even fewer random bits. However, it remains an open problem to determine, even for conventional implementations, the minimum number of random bits necessary to achieve these running times for dynamic (unordered) dictionaries. The complexity of conventional deterministic dictionaries is better understood – for example, there is a deterministic dictionary supporting all operations in $O(\sqrt{\log n / \log \log n})$ time [BF02] – however open problems remain concerning tradeoffs of query time versus update time (see [Ruž08] and the references therein). Interestingly, many of the deterministic dictionary constructions make use of dynamization, the technique discussed in Section 5.12. It is thus natural to ask if there are uniquely represented deterministic dictionaries matching the performance of their conventional counterparts. To the best of my knowledge, it is still open if there is even a uniquely represented deterministic dictionary that is space efficient and supports insertions, deletions, and lookups in $o(n)$ time, let alone $O(\log n)$ or even $O(\sqrt{\log n / \log \log n})$ time.

Finally, another line of inquiry concerns the adaptive uniquely represented data structures of Chapter 6. Just what is the tradeoff between history dependence and adaptivity? This seems related to the question of learning in severely space limited settings.

Bibliography

[ABH+04] Umut A. Acar, Guy E. Blelloch, Robert Harper, Jorge L. Vittes, and Shan Leung Maverick Woo. Dynamizing static algorithms, with applications to dynamic trees and history independence. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM Symposium on Discrete Algorithms*, pages 531–540, 2004. 8, 9, 15, 27, 145, 148, 149, 151, 152, 153

[ABH06] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(6):990–1034, 2006. 152

[ABV05] Umut A. Acar, Guy E. Blelloch, and Jorge L. Vittes. An experimental analysis of change propagation in dynamic trees. In *ALENEX/ANALCO '05: Proceedings of the Seventh Workshop on Algorithm Engineering and Experiments and the Second Workshop on Analytic Algorithmics and Combinatorics*, pages 41–54, 2005. 151

[Aca05] Umut A. Acar. *Self-adjusting computation*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2005. Co-Chair-Guy Blelloch and Co-Chair-Robert Harper. 8

[AG93] Andrew W. Appel and Marcelo J. R. Gonçalves. Hash-consing garbage collection. Technical Report CS-TR-412-93, Princeton University, 1993. 73

[AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey Ullman. *Data Structures and Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983. 49, 103

[AK74] Ole Amble and Donald E. Knuth. Ordered hash tables. *The Computer Journal*, 17(2):135–142, May 1974. 14, 31

[All78] John Allen. *Anatomy of LISP*. McGraw-Hill, Inc., New York, NY, USA, 1978. 69, 71

[AO95] Arne Andersson and Thomas Ottmann. New tight bounds on uniquely represented dictionaries. *SIAM Journal of Computing*, 24(5):1091–1103, 1995. 14

[AS00] Noga Alon and Joel H. Spencer. *The Probabilistic Method*. Wiley-Interscience, August 2000. 70

[AVL62] Georgii M. Adelson-Velskii and Evgenii M. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk SSSR*, 146:263–266, 1962. (Russian). English translation by Myron J. Ricci in Soviet Math. Doklady, 3:1259-1263, 1962. 55, 98, 181

[BA95] Amir M. Ben-Amram. What is a "pointer machine" ? *SIGACT News*, 26(2):88–95, 1995. 145

[Bay72] Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972. 55, 181

[BCD+02] Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *ESA '02: Proceedings of the tenth annual European Symposium on Algorithms*, pages 152–164, London, UK, 2002. Springer-Verlag. 119

[BCDI07] 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, 2007. 158, 161

[BF02] Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65(1):38–72, 2002. 98, 183

[BG06] Guy E. Blelloch and Daniel Golovin. Strongly history independent hashing with deletion. Technical Report CMU-CS-06-156, School of Computer Science, Carnegie Mellon University, October 2006. 13

[BG07] Guy E. Blelloch and Daniel Golovin. Strongly history-independent hashing with applications. In *48th Annual IEEE Symposium on Foundations of Computer Science*, pages 272–282. IEEE, October 2007. 13, 29, 78, 101, 124, 127, 131

[BGV08a] Guy E. Blelloch, Daniel Golovin, and Virginia Vassilevska. Uniquely represented data structures for computational geometry. In *SWAT '08: Proceedings of the 11th Scandinavian Workshop on Algorithm Theory*, pages 17–28, Gothenburg, Sweden, July 2008. Springer. 13

[BGV08b] Guy E. Blelloch, Daniel Golovin, and Virginia Vassilevska. Uniquely represented data structures for computational geometry. Technical Report CMU-CS-08-115, Carnegie Mellon University, April 2008. 13

[Blu98] Avrim Blum. On-line algorithms in machine learning. In Amos Fiat and Gerhard Woeginger, editors, *Online Algorithms: The State of the Art*, volume 1442 of *Lecture Notes in Computer Science*, pages 306–325. Springer-Verlag, 1998. 159

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972. 102

[BP06] Niv Buchbinder and Erez Petrank. Lower and upper bounds on obtaining history independence. *Information and Computation*, 204(2):291–337, 2006. 15

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986. 7, 8

[CG86] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading. *Algorithmica*, 1:133–196, 1986. 131, 135

[CL06] Fan Chung and Linyuan Lu. Concentration inequalities and martingale inequalities: A survey. *Internet Mathematics*, 3(1):79–127, 2006. 97

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001. xvii, 49, 63, 103

[Com79] Douglas Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979. 103

[CW79] Larry Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979. 25

[dBvKOS97] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational geometry: algorithms and applications*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997. 131, 132

[Die89] Paul F. Dietz. Fully persistent arrays (extended abstract). In *WADS '89: Proceedings of the Workshop on Algorithms and Data Structures*, pages 67–74, London, UK, 1989. Springer-Verlag. 124

[DKM+94] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal of Computing*, 23(4):738–761, 1994. 11, 35

[DS87] Paul F. Dietz and Daniel D. Sleator. Two algorithms for maintaining order in a list. In *STOC '87: Proceedings of the nineteenth annual ACM Symposium on Theory of Computing*, pages 365–372, New York, NY, USA, 1987. ACM Press. 119, 120

[DW03] Martin Dietzfelbinger and Philipp Woelfel. Almost random graphs with simple hash functions. In *STOC '03: Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 629–638, New York, NY, USA, 2003. ACM. 25, 38, 41, 42, 70

[Ers58] Andrei P. Ershov. On programming of arithmetic operations. *Communications of the ACM*, 1(8):3–6, 1958. 72

[FC06] Jean-Christophe Filliâtre and Sylvain Conchon. Type-safe modular hash-consing. In *ML '06: Proceedings of the 2006 workshop on ML*, pages 12–19, New York, NY, USA, 2006. ACM. 73

[FKS84] 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, 1984. 11, 35

[FS89] M. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *STOC '89: Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 345–354, New York, NY, USA, 1989. ACM. 171

[FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987. 61

[FW94] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48(3):533–551, 1994. 131

[GF64] Benrard A. Galler and Michael J. Fisher. An improved equivalence algorithm. *Communications of the ACM*, 7(5):301–303, 1964. 171

[Got74] Eiichi Goto. Monocopy and associative algorithms in an extended lisp. Technical Report TR 74-03, University of Tokyo, May 1974. 71, 72, 146

[GS62] David Gale and Lloyd Shapley. College admissions and the stability of marriage. *American Mathematical Monthly*, 69:9–15, 1962. 10, 27, 28

[GS78] Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *FOCS '78: IEEE Symposium on Foundations of Computer Science*, pages 8–21, 1978. 55, 181

[HHM+05] Jason D. Hartline, Edwin S. Hong, Alexander E. Mohr, William R. Pentney, and Emily Rocke. Characterizing history independent data structures. *Algorithmica*, 42(1):57–74, 2005. 1, 14, 15, 20, 21, 22, 23, 35

[HK71] Kenneth Hoffman and Ray Kunze. *Linear Algebra*. Prentice Hall, Inc., second edition, 1971. 7

[Hop71] John E. Hopcroft. An n log n algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *The Theory of Machines and Computations*, pages 189–196. Academic Press, 1971. 7

[Huf54] David A. Huffman. The synthesis of sequential switching circuits. *J. Franklin Institute*, 257:161–191,275–303, 1954. 7

[Jan05] Svante Janson. Individual displacements for linear probing hashing with different insertion policies. *ACM Transactions on Algorithms*, 1(2):177–213, 2005. 36

[Jen06] Johan L. W. V. Jensen. Sur les fonctions convexes et les inégalités entre les valeurs moyennes. *Acta Mathematica*, 30(1):175–193, 1906. 94

[JL99] Richard E. Jones and Rafael D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1999. 178

[Jon96] Richard Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, July 1996. 147

[Knu71] Donald E. Knuth. Optimum binary search trees. *Acta Informatica*, 1:14–25, 1971. 159

[Knu97] Donald E. Knuth. *Stable Marriage and its Relation to Other Combinatorial Problems: An Introduction to the Mathematical Analysis of Algorithms*, volume 10 of *CRM Proceedings and Lecture Notes*. American Mathematical Society, 1997. Originally published in French in 1981 as *Mariages stables et leurs relations avec d'autres problèmes combinatoires: Introduction a l'analyse mathematique des algorithmes*; translated by M. Goldstein. 29

[Koz91] Dexter Kozen. *The Design and Analysis of Algorithms*. Springer-Verlag, New York, 1991. xvii, 63

[Koz97] Dexter Kozen. *Automata and Computability*. Springer-Verlag, New York, 1997. 7

[KT06] Jon M. Kleinberg and Éva Tardos. *Algorithm design*. Pearson/Addison-Wesley, first edition, 2006. 28

[KV05] Adam Kalai and Santosh Vempala. Efficient algorithms for online decision problems. *Journal of Computer and System Sciences*, 71(3):291–307, 2005. xvi, 158, 159, 160

[Mau00] Laurent Mauborgne. An incremental unique representation for regular trees. *Nordic Journal of Computing*, 7(4):290–311, 2000. 73

[McC85] Edward M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985. 56

[Mes97] Xavier Messeguer. Skip trees, an alternative data structure to skip lists in a concurrent approach. *Informatique Théorique et Applications*, 31(3):251–269, 1997. 74

[Mic97] Daniele Micciancio. Oblivious data structures: applications to cryptography. In *STOC '97: Proceedings of the twenty-ninth annual ACM Symposium on Theory of Computing*, pages 456–464, New York, NY, USA, 1997. ACM Press. 10, 14, 20, 145

[MKSW06] David Molnar, Tadayoshi Kohno, Naveen Sastry, and David Wagner. Tamper-evident, history-independent, subliminal-free data structures on PROM storage -or- How to store ballots on a voting machine (extended abstract). In *IEEE Symposium on Security and Privacy*, May 2006. 15

[MN90a] 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. 98, 101

[MN90b] Kurt Mehlhorn and Stefan Näher. Dynamic fractional cascading. *Algorithmica*, 5(2):215–241, 1990. 131

[MNS07] Tal Moran, Moni Naor, and Gil Segev. Deterministic history-independent strategies for storing information on write-once memories. In *ICALP 2007*, volume 4596 of *Lecture Notes in Computer Science*. Springer, July 2007. 15

[Moo56] Edward F. Moore. Gedanken-experiments on sequential machines. In Claude Shannon and John McCarthy, editors, *Automata Studies*, pages 129–153. Princeton University Press, Princeton, NJ, 1956. 7

[Mor03a] Christian Worm Mortensen. Fully-dynamic orthogonal range reporting on RAM. In *Technical report TR-2003-22 in IT University Technical Report Series*, 2003. 124

[Mor03b] Christian Worm Mortensen. Fully-dynamic two dimensional orthogonal range and line segment intersection reporting in logarithmic time. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM Symposium on Discrete Algorithms*, pages 618–627, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics. 135

[Mor06] Christian Worm Mortensen. Fully dynamic orthogonal range reporting on RAM. *SIAM Journal on Computing*, 35(6):1494–1525, 2006. 131

[MR89] Gary L. Miller and John H. Reif. Parallel tree contraction part 1: Fundamentals. In Silvio Micali, editor, *Randomness and Computation*, pages 47–72. JAI Press, Greenwich, Connecticut, 1989. Vol. 5. 151

[MR91] Gary L. Miller and John H. Reif. Parallel tree contraction part 2: further applications. *SIAM Journal on Computing*, 20(6):1128–1147, 1991. 151

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Cambridge University Press, New York, NY, USA, 1995. 70, 168

[Mul94] Ketan Mulmuley. *Computational Geometry: An Introduction through Randomized Algorithms*. Prentice-Hall, Englewood Cliffs, 1994. 91

[Myh57] John Myhill. Finite automata and the representation of events. Technical Report WADD TR-57-624, Wright Patterson AFB, Ohio, 1957. 7

[Ner58] Anil Nerode. Linear automaton transformations. In *Proceedings of the American Mathematical Society*, volume 9, pages 541–544, 1958. 7

[NSW08] Moni Naor, Gil Segev, and Udi Wieder. History-independent cuckoo hashing. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Proceedings of 35th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 5126 of *Lecture Notes in Computer Science*, pages 631–642. Springer, 2008. 16, 41, 42

[NT01] Moni Naor and Vanessa Teague. Anti-persistence: history independent data structures. In *STOC '01: Proceedings of the thirty-third annual ACM Symposium on Theory of Computing*, pages 492–501, New York, NY, USA, 2001. ACM Press. 11, 14, 15, 21, 22, 30, 35

[OP03] Anna Östlin and Rasmus Pagh. Uniform hashing in constant time and linear space. In *STOC '03: Proceedings of the thirty-fifth annual ACM Symposium on Theory of Computing*, pages 622–628, New York, NY, USA, 2003. ACM Press. 25, 38, 41, 42, 70

[Ove81] Mark H. Overmars. Dynamization of order decomposable set problems. *Journal of Algorithms*, 2(3):245–260, 1981. 9

[Ove83] Mark H. Overmars. Corrigendum: Dynamization of order decomposable set problems. *Journal of Algorithms*, 4(3):301, 1983. 9

[Ove87] Mark H. Overmars. *Design of Dynamic Data Structures*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1987. 9

[OvL81] Mark H. Overmars and Jan van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Science*, 23:166–204, 1981. 9, 139, 141, 143, 144

[PMP90] Thomas Papadakis, J. Ian Munro, and Patricio V. Poblete. Analysis of the expected search cost in skip lists. In *SWAT '90: Proceedings of the second Scandinavian workshop on Algorithm theory*, pages 160–172, New York, NY, USA, 1990. Springer-Verlag New York, Inc. 69

[Pou96] J. A. La Poutré. Lower bounds for the union-find and the split-find problem on pointer machines. *Journal of Computer and System Sciences*, 52(1):87–99, 1996. 171

[PPR07] Anna Pagh, Rasmus Pagh, and Milan Ružić. Linear probing with constant independence. In *STOC '07: Proceedings of the thirty-ninth annual ACM Symposium on Theory of Computing*, pages 318–327, New York, NY, USA, 2007. ACM Press. 10, 25, 34, 183

[PR04] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004. 11, 16, 35, 42

[PT89] William Pugh and Tim Teitelbaum. Incremental computation via function caching. In *POPL '89: Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 315–328, 1989. 69

[PT06] Mihai Pătraşcu and Mikkel Thorup. Time-space trade-offs for predecessor search. In *Proceedings of the 38th ACM Symposium on Theory of Computing (STOC)*, pages 232–240, 2006. 98

[Pug88] William W. Pugh. *Incremental computation and the incremental evaluation of functional programs*. PhD thesis, Cornell University, Ithaca, NY, USA, 1988. 8, 9

[Pug90] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990. 67, 69, 70, 74

[RS07] Ronald L. Rivest and Warren D. Smith. Three voting protocols: Three-ballot, VAV, and Twin. In *EVT'07: Proceedings of the USENIX/Accurate Electronic Voting Technology on USENIX/Accurate Electronic Voting Technology Workshop*, Berkeley, CA, USA, 2007. USENIX Association. 4

[Ruž08] Milan Ružić. Uniform deterministic dictionaries. *ACM Transactions on Algorithms*, 4(1):1–23, 2008. 183

[SA96] Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996. xvii, 12, 55, 56, 57, 59, 61, 62, 64, 65, 94, 95, 101, 116, 119, 123, 129, 135, 144, 145

[Sen95] Sandeep Sen. Fractional cascading revisited. *Journal of Algorithms*, 19(2):161–172, 1995. 131

[SG76] Masataka Sassa and Eiichi Goto. A hashing method for fast set operations. *Information Processing Letters*, 5(2):31–34, 1976.

[Sha48] C. E. Shannon. A mathematical theory of communication. *Bell system technical journal*, 27:379–423, 623–656, 1948. 158

[Sie89] Alan Siegel. On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications. In *FOCS '89: IEEE Symposium on Foundations of Computer Science*, pages 20–25, 1989. 25

[Sie95a] Alan Siegel. On universal classes of extremely random constant time hash functions and their time-space tradeoff. Technical report, New York University, New York, NY, USA, 1995. 25, 41

[Sie95b] Alan Siegel. Toward a usable theory of chernoff bounds for heterogeneous and partially dependent random variables. Technical Report TR1995-685, New York University, New York, NY, USA, 1995. 97

[Sny77] Lawrence Snyder. On uniquely representable data structures. In *FOCS '77: IEEE Symposium on Foundations of Computer Science*, pages 142–146. IEEE, 1977. 13, 14

[ST83] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983. 150

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985. 55, 150, 155, 157, 158

[ST90] Rajamani Sundar and Robert E. Tarjan. Unique binary search tree representations and equality-testing of sets and sequences. In *STOC '90: Proceedings of the twenty-second annual ACM Symposium on Theory of Computing*, pages 18–25, New York, NY, USA, 1990. ACM Press. 14

[Ste04] J. Michael Steele. *The Cauchy-Schwarz Master Class: An Introduction to the Art of Mathematical Inequalities*. Cambridge University Press, New York, NY, USA, 2004. 94

[Tar75] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975. 155, 171

[Tar79] Robert Endre Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18(2):110–127, 1979. 171

[Tar83] Robert Endre Tarjan. *Data structures and network algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983. 55

[TZ04] Mikkel Thorup and Yin Zhang. Tabulation based 4-universal hashing with applications to second moment estimation. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 615–624, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics. 25, 45

[vEB77] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977. 101

[vEB90] Peter van Emde Boas. Machine models and simulations. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science (vol. A):*

Algorithms and Complexity, pages 1–66. MIT Press, Cambridge, MA, USA, 1990. 20

[vEBKZ77] Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977. 98, 101

[Vit06] Jeffrey Scott Vitter. Algorithms and data structures for external memory. *Foundations and Trends in Theoretical Computer Science*, 2(4):305–474, 2006. 102, 103

[Vui78] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978. 61

[Vui80] Jean Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980. 12, 56

[WC81] Mark N. Wegman and Larry Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22(3):265–279, 1981. 25

[Wer06] Renato F. Werneck. *Design and Analysis of Data Structures for Dynamic Trees*. PhD thesis, Princeton University, June 2006. 151

[Wil64] John W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7:347–348, 1964. 61

[Wil83] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983. 98, 99

[Wil84] Dan E. Willard. New trie data structures which support very fast search operations. *Journal of Computer and System Sciences*, 28(3):379–394, 1984. 98

[Yao77] Andrew Chi-Chih Yao. Probabilistic computations: Toward a unified measure of complexity. In *FOCS '77: IEEE Symposium on Foundations of Computer Science*, pages 222–227, 1977. 168

[Yao81] Andrew Chi-Chih Yao. Should tables be sorted? *Journal of the ACM*, 28(3):615–628, 1981. 165