Amortized Complexity of Data Structures

Rajamani Sundar

October 1991

A dissertation in the Department of Computer Science submitted to the faculty of the Graduate School of Arts and Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy at New York University.

Approved

(Signed) _____

Ravi Boppana
Research Advisor

Amortized Complexity of Data Structures

Rajamani Sundar
Advisor: Ravi Boppana

Abstract

This thesis investigates the amortized complexity of some fundamental data structure problems and introduces interesting ideas for proving lower bounds on amortized complexity and for performing amortized analysis. The problems are as follows:

- *Dictionary Problem:* A *dictionary* is a dynamic set that supports searches of elements and changes under insertions and deletions of elements. It is open whether there exists a dictionary data structure that takes constant amortized time per operation and uses space polynomial in the dictionary size. We prove that dictionary operations require log-logarithmic amortized time under a multilevel hashing model that is based on Yao's cell probe model.

- *Splay Algorithm's Analysis: Splay* is a simple, efficient algorithm for searching binary search trees, devised by Sleator and Tarjan, that uses rotations to reorganize the tree. Tarjan conjectured that Splay takes linear time to process deque operation sequences on a binary tree and proved a special case of this conjecture called the *Scanning Theorem.* We prove tight bounds on the maximum numbers of various types of right rotations in a sequence of right rotations performed on a binary tree. One of the lower bounds refutes a conjecture of Sleator. We apply the upper bounds to obtain a nearly linear upper bound for Tarjan's conjecture. We give two new proofs of the Scanning Theorem, one of which is a potential-based proof that solves a problem of Tarjan.

- *Set Equality Problem:* The task of maintaining a dynamic collection of sets under various operations arises in many applications. We devise a fast data structure for maintaining sets under equality-tests and under creations of new sets through insertions and deletions of elements. Equality-tests require constant time and set-creations require logarithmic amortized time. This improves previous solutions.

Acknowledgements

It is a pleasure to acknowledge the help of many people in performing the work of this thesis.

I would like to thank my advisor Ravi Boppana for teaching me many useful research skills and teaching/presentation skills. I learnt from him the importance of simple and clear formulation of ideas, and about several useful ideas in Theoretical Computer Science and Discrete Mathematics. The work on the Dictionary Problem owes a great deal to the various things I learnt from him and to his help. He showed me the initial directions to proceed along, he came up with fresh approaches to pursue whenever I failed, and he kept encouraging me to succeed.

I would like to thank my co-advisor Richard Cole for initiating me into the area of amortized analysis, for always being a source of inspiration, and for always giving me kind attention and advice when I needed them. The work on the Deque Conjecture owes greatly to the several fruitful discussions I had with him.

I would like to thank Mike Fredman, Bud Mishra, and Bob Tarjan for serving on my thesis defense committee and for their role in this work. Mike Fredman and Bob Tarjan supported and guided me during the summer of 1989 when I was a student under them at Bellcore and at Bell labs, respectively. They were always sources of nice problems to work on, and of inspiration and encouragement. The work on the Set Equality Problem is part of a larger joint-work done with Bob Tarjan. Bud Mishra supported me during the academic year 1988-89, and was always a source of encouragement and help.

I would like to thank the Courant Institute community for providing a great environment for this work.

Finally, I would like to thank IBM Corporation for graciously supporting me during the academic years 1989-91.

Contents

Chapter 1

Introduction

1.1 Amortized Complexity

In many applications of data structures, the data structure is embedded within some algorithm that performs successive operations on it. In these applications, we are interested only in the time taken by the data structure to process operation sequences as a whole and not in the time spent on isolated operations. Amortized data structures are data structures tailored to such applications: these data structures may perform poorly on a few individual operations but perform very well on all operation sequences. The natural performance measure for an amortized data structure is its *amortized complexity,* defined to be the maximum cost of operation sequences performed on the data structure as a function of the lengths of the sequences. Amortized data structures are appealing because they dispense with complicated constraints and associated information present in data structures that achieve a fast performance on all operations and they use simple reorganizing heuristics, instead, to achieve a fast amortized performance. Some examples of these data structures are the compressed tree data structures for the Union-find Problem [1,23,32], the Splay Tree [23,28,32], and the Pairing Heap [16].

Amortized data structures are simple to describe but their performance analysis is often quite involved. Since operation sequences on these data structures are mixtures of operations of varying costs that very finely interact with one another it is tricky to accurately estimate their amortized complexity. Of the three amortized data structures mentioned above only the first one has been analyzed thoroughly; even its analysis was accomplished only several years after the data structure was originally conceived. The

complete analysis of the other two is still open.

A useful framework for performing amortized analysis involves defining an appropriate potential function for the data structure [34]. In this framework, each state of the data structure is assigned a real number called its *potential.* The *amortized cost* of a data structure operation is defined to be the actual cost incurred by that operation plus the increase in potential it causes through change of state. The total cost of an operation sequence performed on the data structure equals the total amortized cost of the operations in the sequence plus the decrease in potential from the initial state to the state at the end of the operation sequence. Choosing a suitable potential function that yields a sharp estimate on the amortized complexity is a task that demands ingenuity.

We illustrate the potential framework through a simple example. A *priority queue with attrition (PQA)* is a dynamic set of real numbers supporting two operations: DELETEMIN deletes and returns the smallest element of the set; INSERT(x) deletes all elements of the set greater than x and adds x to the set. A PQA can be represented by a sorted list of its elements. In this representation, DELETEMIN takes constant time, and INSERT takes time proportional to the number of deleted elements. If we define the potential of the data structure equal to the number of elements it contains then PQA operations take constant amortized time; PQA operation sequences take linear time.

The notions of amortized complexity and amortized cost are usually used in a much wider sense than defined above. Amortized complexity is usually a maximizing function of several parameters of the operation sequences, instead of their length alone. The amortized costs of operations are usually a set of functions of the operation sequence parameters that, when added together, yield a good estimate of the amortized complexity. For example, the compressed tree data structures for the Union-find Problem have amortized complexity equal to $O(n + m\alpha(m + n, n))$, and take constant time on UNION operations and $O(\alpha(m + n, n))$ amortized time on FIND operations, where $\alpha(m, n)$ is an inverse function of the Ackerman function, and m and n respectively denote the total number of FINDs and the total number of UNIONs in the operation sequence [23,32].

Let us relate amortized complexity to other measures of data structure performance. There exist data structure problems whose amortized complexities are lower than their worst-case complexities; for instance, the Union-find Problem is solvable in nearly constant amortized time per operation but requires nearly logarithmic worst-case time per

operation [7,15,32]. There probably exist problems whose randomized (or average-case) complexities are lower than their amortized complexities, but this is yet to be proven; for instance, the Dictionary Problem and certain other problems that involve testing equality of objects appear to be good candidates for this class. The amortized complexity of a dynamic data structure problem is often intimately related to the complexity of its static version via transformations of solutions/adversaries of one problem into another [6,39].

An appropriate model for proving lower bounds on the amortized complexity of a data structure problem is Yao's *cell probe model* [38]. This model abstracts out the arithmetic and indexing capabilities of random access machines without ignoring their word-size limitations. The model has a memory consisting of an array of b-bit locations. Data structure operations are performed in a series of memory probes in which the next probe location is always computed as a general function of the values so far read. In this model, Fredman and Saks [15] proved tight lower bounds on the amortized complexity of many problems, including the Union-find Problem. The only other interesting lower bound known in this model is for a static data structure problem, due to Ajtai [3]. The complexity of many other problems, notably, the Dictionary Problem and the Priority Queue Problem, remains unexplored.

This completes our introduction to amortized complexity. Further information on this subject can be found in [23,32,34].

1.2 Our Work

This thesis investigates the amortized complexity of some fundamental data structure problems. We introduce interesting ideas for proving lower bounds on amortized complexity and for performing amortized analysis that enable progress towards resolving some open questions. The problems studied are as follows.

In Chapter 2, we study the amortized complexity of the Dictionary Problem. A *dictionary* is a dynamic set that supports searches, insertions, and deletions of elements. It is an open problem whether a dictionary can be maintained in constant amortized time per operation using space polynomial in the dictionary size; we denote the dictionary size by n. While hashing schemes solve the problem in constant amortized time per oper-

ation on the average or using randomness, the best deterministic solution (uses hashing and) takes $\Theta(\log n / \log\log n)$ amortized time per operation. We study the deterministic complexity of the dictionary problem under a multilevel hashing model that is based on Yao's cell probe model, and prove that dictionary operations require $\Omega(\log\log n)$ amortized time in this model. Our model encompasses many known solutions to the dictionary problem, and our result is the first nontrivial lower bound for the problem in a reasonably general model that takes into account the limited wordsize of memory locations and realistically measures the cost of update operations. This lower bound separates the deterministic and randomized complexities of the problem under this model.

In Chapter 3, we study a problem arising in the analysis of Splay, a rotation-based algorithm for searching binary search trees that was devised by Sleator and Tarjan. Tarjan proved that Splay takes linear time to scan the nodes of a binary tree in symmetric order; this result is called the *Scanning Theorem.* More generally, he conjectured that Splay takes linear time to process deque operation sequences on a binary tree; this conjecture is called the *Deque Conjecture.* We prove that Splay takes linear-times-inverse-Ackerman time to process deque operation sequences. In the process, we obtain tight bounds on some interesting combinatorial problems involving rotation sequences on binary trees. These problems arose from studying a conjecture of Sleator that we refute. We give two new proofs of the Scanning Theorem. One proof is a potential-based proof; Tarjan had originally posed the problem of finding such a proof. The other proof uses induction.

In Chapter 4, we study the problem of maintaining a dynamic collection of sets under equality-tests of two sets and under creations of new sets through insertions and deletions of elements. We devise a data structure that takes constant time on equality-tests and logarithmic amortized time on set-creations. The data structure derandomizes a previous randomized data structure, due to Pugh and Teitelbaum, that took logarithmic expected amortized time on set-creations.

Some of our work has been published before. The work on the Deque Conjecture was reported in [30]. The work on the Set Equality Problem was reported in a joint-paper with Robert E. Tarjan [31] that also dealt with other related problems.

Chapter 2

The Dictionary Problem

The *Dictionary Problem* is a classical data structure problem arising in many applications such as symbol table management in compilers and language processors. The problem is to maintain a dynamic set under the operations of search, insertion, and deletion of elements. The problem can be solved in constant time per operation using a bit vector that has a separate bit for each element of the universe indicating its prescence in the set. This simple solution is unsatisfactory since it uses space proportional to the universe size and the dictionary is usually very small compared to the universe. Does there exist a constant-amortized-time solution that uses only space polynomial in the dictionary size?

In this chapter, we study the amortized complexity of the dictionary problem under a multilevel hashing model that is based on Yao's cell probe model, and prove that dictionary operations require log-logarithmic amortized time in this model.

2.1 Introduction

The *Dictionary Problem* is to maintain a dynamic set, called a *dictionary,* over a finite, ordered universe U under the following operations:

- SEARCH(x): Determine whether element x is in the dictionary; if so, find a memory location storing x.
- INSERT(x): Add element x to the dictionary.
- DELETE(x): Remove element x from the dictionary.

The dictionary is initially the empty set. We would like to know how fast the problem can be solved using only space polynomial in the dictionary size (denoted by n) on a RAM with $(\log |U|)$-bit memory words. Here we are restricting the space used in terms of the dictionary size in order to exclude the trivial bit vector solution and a family of its generalizations, due to Willard [37], that process dictionary operations in constant time and use space depending on the size of the universe.

The Dictionary Problem has been extensively studied and has a rich variety of solutions.

Balanced search trees solve the problem in $O(\log n)$ time per operation and use $O(n)$ space [1,21,23,32]. Self-adjusting search trees such as Sleator and Tarjan's Splay Tree [28] and Galperin and Rivest's Scapegoat Tree [18] take $O(\log n)$ amortized time per operation. There is no hope of improving the $\log n$ behaviour of search trees since they use only comparisons to perform searches.

Hashing schemes solve the problem in constant expected time per operation and use $O(n)$ space. Uniform hashing performs dictionary operations in constant expected time when the operations are randomly chosen from a uniform distribution [21,1]; the space used is $O(n)$. Universal hashing is an improved randomized hashing scheme that performs searches in constant expected time and performs updates in constant expected amortized time [8]; the expectation, here, is over the random hash functions chosen by the algorithm and the bounds apply to all possible operation sequences. Dynamic perfect hashing [13] (see also [2,14]) is a further improvement that performs searches in constant worst-case time and performs updates in constant expected amortized time. All of the above hashing schemes fall under the general category of multilevel hashing schemes. Roughly speaking, a multilevel hashing scheme consists of a hierarchically organized system of hash functions that successively partition the dictionary into finer pieces until all elements in the dictionary have been separated plus an algorithm to update the configuration after each dictionary operation.

The fastest deterministic solution to the problem, at present, is a multilevel hashing scheme, due to Fredman and Willard [17], that takes $O(\log n / \log\log n)$ amortized time per operation and uses $O(n)$ space.

It has been possible to prove nonconstant lower bounds for the problem on certain deterministic hashing models. Dietzfelbinger et al. [13] showed a tight bound of $\Theta(\log n)$ on

the amortized complexity of dictionary operations under a wordsize-independent multilevel hashing model that does not include search trees. Mehlhorn et al. [24] strengthened this model so that it includes search trees and showed that dictionary operations have $\Theta(\log\log n)$ amortized complexity under their model. These models assume that memory locations have an unbounded wordsize and overestimate the costs of operations; this simplifies the task of proving lower bounds but the models are not realistic. If memory locations have a sufficiently large wordsize, the problem is solvable in constant time per operation (Paul and Simon [25]; Ajtai et al. [4]). When memory locations can only represent a single element of the universe, however, the best available solution is Fredman and Willard's $O(\log n/\log\log n)$-time solution [17].

We prove a nonconstant lower bound for the Dictionary Problem under a multilevel hashing model, based on Yao's cell probe model, that takes into account the limited wordsize of memory locations and realistically measures the costs of operations.

We define a generic multilevel hashing model for solving the Dictionary Problem from which various lower bound models for the problem can be derived by specifying suitable cost measures for the operations. The model has a vertically organized memory that consists of a *root* location at level 1 and an array of at most m locations at level i, for each $i \geq 2$. Memory locations have b bits each, for some $b \geq \log|U|$. Each memory location stores some c different elements of the universe plus a $(b - c\log|U|)$-bit value that guides search operations; here the number of elements stored at a location varies over time, and is not constant. A search operation locates an element in memory by performing a sequence of memory probes: the first probe of the sequence always reads the root location and the i-th probe, for $i \geq 2$, reads a location at level i that is computed as a general function of the sequence of $i-1$ values so far read and the element being searched. The operation either locates the element in some location after a series of probes or concludes that the element is not in the dictionary after a series of unsuccessful probes. A search operation finally replaces the current memory configuration by a new configuration, representing the same dictionary, by computing a general function of the current configuration. An update operation simply replaces the current memory configuration by a new configuration that represents the new dictionary by computing a general function of the current configuration.

We define a lower bound model for the problem by imposing a measure of operation costs on the generic model. The *Hamming cost* of an operation is defined as the maximum number of locations at any level whose contents change due to the operation. The cost of a search operation is defined as the number of read probes performed by the operation (called the *search cost* of the operation) plus the Hamming cost of the operation. The cost of an update operation is defined as its Hamming cost. When this cost measure is imposed on the generic model we get a lower bound model called the *Hamming cost model.*

We prove our lower bound in the following model that has a different cost measure from the Hamming cost model. The *search path* of an element x of U in a memory configuration C is defined as the sequence of locations probed by a search operation on x performed in configuration C. We say that an operation *refreshes* a memory location l if there is some element x in the final dictionary such that l lies on the search path of x after the operation but l did not lie on the search path of x before the operation. The *refresh cost* of an operation is defined as the maximum number of locations at any level that get refreshed by the operation. Define the cost of an operation as the sum of the search cost and the refresh cost of the operation. The lower bound model with this cost measure is called the *refresh cost model.* The difference between this model and the Hamming cost model is that the refresh cost measure is sensitive to locations that participate in rerouting the search paths of dictionary elements during an operation, on the other hand, the Hamming cost measure is sensitive to locations whose contents change.

A nonconstant lower bound for the Dictionary Problem in the Hamming cost model would translate into a similar lower bound for the problem in the cell probe model. We believe that such a lower bound exists, but we can only prove a lower bound under the refresh cost model. The refresh cost model seems to be incomparable in power to the cell probe model and to the Hamming cost model. The justification for the refresh cost model is that, in realistic hashing schemes, an operation might have to read, and, if necessary, modify, the locations it refreshes in order to correctly reroute the search paths of dictionary elements. The refresh cost model encompasses many of the known solutions to the dictionary problem, but not all of them; for instance, the model includes B-trees, red-black trees, and various hashing schemes, but the class of rotations-based

search trees is not included.

Our lower bound is given by:

Theorem 2.1 *Consider the refresh cost multilevel hashing model with at most m memory locations per level, a universe U, and a wordsize b. Let n be a positive integer satisfying $|U| \geq m^{\log\log n}$. Consider dictionary operation sequences during which the maximum dictionary size is n. Under this model, the amortized complexity of dictionary operations equals $\Omega(\log(\log n / \log b))$.*

In a typical situation where $|U| = m^{\log\log n}$, $m = \text{poly}(n)$, and $b = \log|U|$, the theorem yields a lower bound of $\Omega(\log\log n)$ on the amortized complexity of dictionary operations. This lower bound separates the deterministic and randomized complexities of hashing schemes in the refresh cost model, since this model encompasses randomized hashing schemes like universal hashing [8] and dynamic perfect hashing [13] that process dictionary operations in constant randomized amortized time.

The proof technique of the theorem can be used to show that single-level hashing schemes in the Hamming cost model require $\Omega(n^{\alpha})$ amortized time to process dictionary operations, for some constant α. We hope that the proof technique will be helpful in showing a general nonconstant lower bound for the dictionary problem in the Hamming cost model.

The chapter is organized as follows. In Section 2.2, we introduce the basic ideas needed to prove Theorem 2.1 by proving a nonconstant lower bound under the simpler model of single-level hashing. In Section 2.3, we prove the theorem.

2.2 Single-level Hashing Model

In this section, we prove a nearly linear lower bound for the Dictionary Problem under the refresh cost single-level hashing model.

We define the lower bound model. The model consists of an array of m locations each capable of storing an element of U, a family of at most 2^b hash functions from U to the array, and a b-bit root location storing a hash function from the family. The root hash function is always chosen so that it sends the elements of the dictionary into different locations, and collisions of elements are forbidden; in general, when a hash function sends

the elements of a set into distinct locations it is called a *perfect hash function* [14] for the set and it is said to *shatter* the set. Search operations are performed in two probes: the first probe reads the hash function from the root location; the second probe checks if the element is present in the array location where it is sent by the hash function. An update operation can change the root hash function and can write into some array locations in order to appropriately relocate the dictionary elements. The cost of a search operation is 2 units. The cost of an update operation is its refresh cost, which equals the number of dictionary elements that are relocated during the operation.

The lower bound for single-level hashing is given by:

Theorem 2.2 *Consider the refresh cost single-level hashing model with an array of m memory locations, a universe U, and a b-bit root location. Assume that $|U| \geq 2m$. Consider dictionary operation sequences during which the maximum dictionary size is at most n, where $n \leq m$. Under this model, the amortized complexity of dictionary operations equals $\Omega(n/b)$.*

A typical situation to apply this theorem is when $b = \log |U|$ and m and $|U|$ are polynomial in n. Under these assumptions, the theorem yields a lower bound of $\Omega(n/\log n)$ on the amortized complexity of dictionary operations under the single-level hashing model.

The main idea behind the proof of the theorem is an adversary who keeps creating random collisions in U. Any hashing scheme with a small hash functions family can not succeed against this adversary. The proof uses the following sampling lemma, due to Hoeffding [20], that a random sample closely behaves like the whole population.

Lemma 2.1 (Binary Sampling Lemma) *Let $k \geq 1$ and let $0 < \beta < \alpha < 1$. Consider a population of at least k elements of which some α-fraction of the elements are colored red. A random k-subset of the population has more than βk red elements with probability at least $(1 - e^{-c_{\alpha,\beta} k})$, where*

$$c_{\alpha,\beta} = \begin{cases} (\alpha - \beta)^2/(2\alpha(1-\alpha)) & \text{if } \alpha - \beta < 1 - \alpha \\ 2(\alpha - \beta)^2 & \text{otherwise.} \end{cases}$$

Let us prove the theorem. Denote the amortized cost per update operation incurred by a hashing scheme by w. We prove that $e^{\Omega(n/w)}$ hash functions are needed by the

hashing scheme in order to successfully process all sequences of $O(n)$ insertions. This would imply the theorem. The proof of this result is organized in a series of steps. A hash function is called *uniform* if it sends the same number of elements into each location. In Section 2.2.1, we prove the result for the case when the hash functions used by the hashing scheme are all uniform and the update cost is bounded in the worst-case. In Section 2.2.2, we handle nonuniform hash functions. In Section 2.2.3, we handle amortized update costs.

2.2.1 Uniform Hash Functions and Worst-case Complexity

In this section, we prove that any single-level hashing scheme in the refresh cost model that uses only uniform hash functions and has worst-case update cost w needs at least $e^{\Omega(n/w)}$ hash functions in order to handle all sequences of $O(n)$ insertions.

The adversary performs two batches of insertions: a large batch followed by a small batch. The large batch is a random n-subset of U. Let h denote the hash function used after the large batch; h is a random variable. The small batch is constructed as follows: Randomly select $k = n/cw$ locations, where c is a constant. For each selected location pick a random pair of elements of U that h sends into that location. The small batch is the union of these pairs. Since we assumed that $|U| \geq 2m$ and that h is uniform, h sends at least two elements of U into each location.

The following lemma gives the lower bound on the size of the hash functions family.

Lemma 2.2 *Any hashing scheme requires $e^{\Omega(k)}$ hash functions in order to succeed against the adversary.*

The idea behind the proof of this lemma is that any fixed pair of hash functions (h, g) that are respectively used after the two batches has a low probability of success against the adversary. For if h and g are sufficiently similar then g can not shatter the small batch. On the other hand if h and g are sufficiently dissimilar then too many elements in the large batch will change locations during the small batch.

Proof of Lemma 2.2. Consider a pair of hash functions (h, g) that are respectively used after the two batches. We compute the probability of success of the hashing scheme against the adversary using this pair of hash functions. Define $\delta(h, g) =$ the number of elements of U in which h and g differ. Two cases have to be considered:

Case 1. $\delta(h,g) \geq |U|/8$: By the Binary Sampling Lemma, the large batch has more than $n/16$ elements that differ in h and g with probability at least $(1 - e^{-c_{1/8,1/16}n})$. Thus the update cost incurred by (h,g) during the small batch exceeds $n/16$ with this probability. Setting $k = n/32w$, the maximum update cost allowed during the small batch equals $n/16$. Hence $\Pr[(h,g) \text{ succeeds}] \leq e^{-c_{1/8,1/16}n}$.

Case 2. $\delta(h,g) \leq |U|/8$: A location is called *similar* if g sends into that location greater than $(1/2)$-fraction of the elements that h sends into it. At least $(3/4)$-fraction of the locations are similar locations, since h and g differ in at most $(1/8)$-fraction of U. By the Binary Sampling Lemma, the random set of k locations selected during the small batch contains more than $k/2$ similar locations with probability at least $(1 - e^{-c_{3/4,1/2}k})$.

Consider a set of k locations, comprising at least $k/2$ similar locations, that are used to construct the small batch. g shatters a random pair of elements sent by h into a similar location with probability at most $3/4$. Hence g shatters a random small batch constructed using the above locations with probability at most $(3/4)^{k/2}$.

Combining the above calculations, $\Pr[(h,g) \text{ succeeds}] \leq e^{-c_{3/4,1/2}k} + (3/4)^{k/2}$.

We are ready to compute the constants.

$$c_{1/8,1/16} = 1/56; \;\; c_{3/4,1/2} = 1/6; \;\; (1/2)\ln(4/3) > 1/7.$$

For sufficiently large k, the success probability of (h,g) is at most $e^{-k/8}$. It follows that at least $e^{k/16}$ hash functions are needed by the hashing scheme in order to succeed against the adversary. □

2.2.2 Nonuniform Hash Functions

In this section, we extend the lower bound of the previous section to families of nonuniform hash functions.

The adversary once again inserts a large batch which is a random n-subset of U and then inserts a small batch. The small batch is constructed based on the hash function h that is used by the hashing scheme after the large batch. A *multiple location* of h is a location into which h sends two or more elements of U. Focus on the subuniverse $\bar{U}$ of elements of U that h sends into its multiple locations. Select a random k-subset of $\bar{U}$, for $k = n/cw$. For each element of the subset select a random element of $\bar{U}$ that collides with it under h. The small batch consists of the subset and the elements selected.

Let us see why Lemma 2.2 still holds for this adversary. Once again the idea is to show that any fixed pair of hash functions (h, g) has a low probability of success. The case when $\delta(h, g) \geq |U|/8$ is handled as before. Consider the case $\delta(h, g) \leq |U|/8$. Since $|\bar{U}| \geq |U|/2$, h and g differ in at most $(1/4)$-fraction of $\bar{U}$. An element of $\bar{U}$ that is sent by both h and g into a similar location is called a *similar element.* At least $(1/4)$-fraction of $\bar{U}$ are similar elements. By the Binary Sampling Lemma, we can expect at least $k/8$ elements of the k-subset from which the small batch is constructed to be similar elements. Now if two of these similar elements collide under h then g fails to shatter the small batch. So suppose that h and g send all the similar elements of the small batch into different locations. In this case, as in the previous section, it is easy to see that g fails to shatter the small batch with probability greater than $(1/2)^{k/8}$. This completes the second case and the proof that $e^{\Omega(k)}$ hash functions are needed to succeed against the adversary.

2.2.3 Amortization

In this section, we incorporate amortization into the previous section's argument. Let w denote the amortized cost per update operation incurred by the hashing scheme. The adversary of the previous section is modified by performing a greedy sequence of insertions between the two batches. Immediately after the large batch, the adversary performs a maximal sequence of insertions σ such that σ incurs an update cost of more than $2w|\sigma|$. Then the small batch is performed as before based on the hash function h used after σ. Observe that $|\sigma|$ is at most n, so only $O(n)$ insertions are totally performed. The maximality of σ ensures that the total update cost incurred during the small batch is at most $4wk$. Hence we can apply the argument of the previous section to obtain a lower bound on the hash functions family.

2.3 Multilevel Hashing Model

In this section, we prove a log-logarithmic lower bound for the Dictionary Problem under the refresh cost multilevel hashing model.

The main idea behind the proof is a randomized adversary who alternately performs greedy searches of the dictionary elements and creates random collisions at the various

levels of the multilevel hashing scheme, descending the levels of the scheme. The collisions force the scheme to either incur a large cost on searches or incur a large cost in compressing the dictionary to few levels. The collisions are created in batches of insertions. Each batch is constructed by selecting a set of locations from the first i levels, by focusing on the subuniverse of elements of U whose search paths involve just these locations during the first i probes, and by picking a random subset from the subuniverse of appropriate size; the subset is always chosen to be several times larger than the number of selected locations in order to create collisions.

We sketch the lower bound proof. Let us confine ourselves to worst-case complexity since amortized complexity can be easily handled, as in the single-level hashing lower bound, by appropriately performing greedy searches and greedy insertions during the adversary sequence. The adversary is defined recursively on levels, and the proof has an inductive structure that is reminiscent of a previous lower bound for a static data structure problem in the cell probe model, due to Ajtai [3]. In order to carry out the induction, we need to prove a stronger result that applies to a more general hashing model, called the *partial hashing model.* In a partial hashing scheme, each value of the root location determines a working subuniverse of U on which the scheme supports dictionary operations. The lower bound applies to partial hashing schemes that work, occasionally, on a dense subuniverse of U; we show that such schemes fail, almost surely, against the adversary.

The main feature of the proof is the handling of nonuniform root hash functions. The value stored at the root location of the hashing scheme defines a partial hash function which is essentially the second probe function used by the scheme to perform searches. We say that this hash function is *narrow* if it sends a constant fraction of the universe into a small set of level-2 locations; otherwise the hash function is said to be *wide.* The adversary first recursively performs a *narrow phase* of insertion batches in U that force the scheme to use narrow hash functions and then recursively performs a *wide phase* of much smaller insertion batches within a random subuniverse $\bar{U}$ of U. The wide phase is not always performed; it is performed only if the narrow phase gets prematurely truncated because the hashing scheme has stopped using narrow hash functions. The adversary consists of two different phases because the hashing scheme behaves differently depending on whether it mostly uses narrow hash functions or it mostly uses wide hash

functions and two phases are needed to fool all hashing schemes. We show that the probability of success of the hashing scheme through the completion of either of the phases is small. We analyze the narrow phase using induction; we analyze the wide phase by showing that fixed sequences of root hash functions used during this phase have a low success probability, using some random sampling lemmas and induction.

The proof of the lower bound is organized as follows. In Section 2.3.1, we describe the partial hashing model. In Section 2.3.2, we describe the adversary used for proving a lower bound on worst-case complexity. In Section 2.3.3, we state and prove two technical random sampling lemmas needed to analyze the wide phase. In Section 2.3.4, we prove the worst-case lower bound. In Section 2.3.5, we handle amortized complexity.

2.3.1 Partial Hashing Model

In this section, we describe the partial hashing model.

A *partial hashing scheme* is a hashing scheme that processes dictionary operations in a tower of universes $U_1 \supseteq U_2 \supseteq \cdots \supseteq U_k$. U_k is called the *working universe* of the scheme. The scheme has a vertically organized memory consisting of a (Wb)-bit *root* location and an infinite word-size *advice* location at level-1, and an array of at most m b-bit locations at each level $i \geq 2$; we require that $b \geq \log |U_1|$ so that a location can store any element of the universes. Each possible value v of the root location defines a tower of subuniverses $S_1(v) \supseteq S_2(v) \supseteq \cdots \supseteq S_k(v)$ such that $S_i(v) \subseteq U_i$ for all i. $S_k(v)$ is called the *working subuniverse* corresponding to value v. A search operation starts by probing the root location; if the element being searched is in the current working subuniverse, then the search proceeds downwards as in the standard multilevel hashing model; if the element is not in the current working subuniverse, then the search operation probes the advice location and continues downwards in the standard fashion. An update operation changes the memory configuration; a search operation can also change the memory configuration. The search cost of an operation is defined as the number of read probes performed by the operation, not counting probes on the advice location. The refresh cost of an operation is a suitably defined number that is at least the maximum number of locations at any level that get refreshed by the operation. The cost of an operation is defined to be the sum of its search cost and its refresh cost. This completes the description of the model.

In order to prove a lower bound for partial hashing schemes, we have to require these schemes to use dense subuniverses, at least occasionally. The *density* of a set $S \subseteq T$ in T equals $|S|/|T|$; S is said to be α-*dense* in T if S has a density of at least α in T. Let $\bar{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_k)$ be a vector of values in the interval $[0, 1]$, let e be a positive integer, and let H be a partial hashing scheme. A configuration C of H is said to be $\bar{\alpha}$-*dense* if the subuniverses $S_1(v) \supseteq S_2(v) \supseteq \cdots \supseteq S_k(v)$ corresponding to configuration C have respective densities $\geq \alpha_1, \alpha_2, \ldots, \alpha_k$ in universes $U_1, U_2, \ldots, U_k$. A configuration C of H is said to be $(\bar{\alpha}, e)$-*good* if there is an extension sequence of at most e insertions in U_1, starting from C, after which H is in a $\bar{\alpha}$-dense configuration; otherwise C is said to be $(\bar{\alpha}, e)$-*bad.* H is said to be $(\bar{\alpha}, e)$-*good* if H always uses $(\bar{\alpha}, e)$-good configurations. Our lower bound applies to partial hashing schemes that are $(\bar{\alpha}, e)$-good.

2.3.2 Adversary

In this section, we describe a randomized adversary for proving a lower bound on the worst-case complexity of good partial hashing schemes.

We define an adversary $A^r_{\bar{\alpha},n,w,b}$ against a partial hashing scheme H with a tower of universes $U_1 \supseteq U_2 \supseteq \cdots \supseteq U_k$ and a working universe $U = U_k$ that has been partitioned into n equal-sized blocks; here $\bar{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_k)$. The adversary is tailored against $(\bar{\alpha}, e)$-good partial hashing schemes with worst-case search cost r and worst-case update cost w, but it is defined against any partial hashing scheme; we will define e later. The adversary either performs a complete sequence of $O(n)$ insertions on the initial dictionary leaving H in a $\bar{\alpha}$-dense configuration or performs a truncated sequence of operations which could not be completed because the hashing scheme has entered an $(\bar{\alpha}, e)$-bad configuration.

We need some definitions. Let p be a positive integer, let $\epsilon \in [0, 1]$, and let h be a partial hash function from a universe $\tilde{U}$ to an array of locations. The domain of h is denoted by $\mathrm{dom}(h)$. A *random* (ϵ, p)-*sample* from h is a random subset R of $\mathrm{dom}(h)$ constructed as follows. First delete from $\mathrm{dom}(h)$ all the elements that go into locations where h sends less than an (ϵp)-fraction of $\tilde{U}$; then pick a random p-subset S from $\mathrm{dom}(h)$; for each location into which h sends k elements of S, pick a random subset with density ϵk in $\tilde{U}$ that h sends into that location; R is the union of the subsets picked from the various locations. A partial hash function is said to be (α, W)-*narrow,* for some

$\alpha \in [0,1]$ and some positive integer W, if it sends at least an α-fraction of $\tilde{U}$ into some set of W locations; otherwise the hash function is said to be (α, W)-*wide.* A partial hash function is said to be β-*biased,* for some $\beta \in [0,1]$, if it sends at most a β-fraction of $\tilde{U}$ into any single location. We can *prune* a partial hash function and make it β-biased, for any given $\beta \in [0,1]$, by deleting sufficiently many elements from its domain.

We are ready to define the adversary. The adversary first inserts a random n-subset of U that is formed by picking a random element from each block of U (called the *first batch*) and then performs a phase of insertions, called the *tail phase.* The tail phase is defined recursively on r:

Basis. $r = 1$: If there exists an extension sequence $\bar{\sigma}$ consisting of at most $e = n$ insertions in U_1 taking H to a $\bar{\alpha}$-dense configuration, perform $\bar{\sigma}$ and announce completion; otherwise H is in a $(\bar{\alpha}, e)$-bad configuration, so announce truncation.

Induction Step. $r \geq 2$: We first perform a recursive subphase, called the *narrow phase;* if this phase gets truncated we perform another recursive subphase, called the *wide phase.* The adversary announces completion if one of the two phases completes; otherwise the adversary announces truncation. The two phases are performed as follows:

Narrow Phase: Let W_1 be a suitably defined positive integer. We construct a new hashing scheme H_1 from H, having a tower of universes $U_1 \supseteq U_2 \supseteq \cdots \supseteq U_k = U_{k+1}$, by collapsing the top two levels of H into a single level. Each possible value stored at the root location of H defines a partial hash function from U_k to the set of level-2 locations which equals the second probe function used by search operations when that value is stored at the root. We rank the level-2 locations of H according to each partial hash function h used by H at its root location as follows. The i-th location of a partial hash function h from a domain $D \subseteq U$ to the set of level-2 locations is defined to be the location into which h sends the i-th largest subset of D; we resolve ties in favor of the location with the smallest index. The root location of H_1, at any time, consists of the root location of H juxtaposed with the first $W_1/2$ level-2 locations of the current root partial hash function (this set of locations is denoted by L_1); the advice location of H_1 consists of the advice location of H juxtaposed with the remaining level-2 locations of H; the i-th level of H_1, for $i \geq 2$, coincides with the $(i+1)$-st level of H. The working subuniverse of H_1, at any time, is the subset of the working subuniverse of H that the root hash function of H maps into the set of locations L_1; the tower of subuniverses of

H_1, at any time, consists of the tower of subuniverses of H followed by H_1's working subuniverse. Operations are performed by H_1 by simulating the behaviour of H. The *search cost* of an operation on H_1 is defined in the usual way. The *refresh cost* of an operation on H_1 is defined equal to the refresh cost of that operation on H. Notice that this definition of refresh cost is more liberal than the usual definition of the refresh cost as the maximum number of locations at any level that get refreshed.

The narrow phase of the adversary $A^r_{\bar{\alpha},n,w,b}$ against H is recursively defined to be the tail phase of the adversary $A^{r-1}_{\bar{\beta}_1,n,w,b}$ against H_1, starting from the same configuration as H's configuration prior to the narrow phase; here $\bar{\beta}_1 = (\alpha_1, \alpha_2, \ldots, \alpha_k, \alpha_k/2)$. The narrow phase completes if the recursive adversary announces completion; otherwise the narrow phase is truncated. The narrow phase always forces H to use $(\alpha_k/2, W_1/2)$-narrow root hash functions; if the phase gets truncated then H will always use $(\alpha_k/2, W_1/2)$-wide root hash functions in any $\bar{\alpha}$-dense configuration during the next e_1 insertions in U_1; we will specify e_1 later.

Wide Phase: The wide phase consists of two subphases: the *first phase* incrementally constructs a random subuniverse $U_{k+1} \subseteq U$ and alternately performs random insertion batches in U_{k+1} and extension batches in U_1; the *second phase* recursively performs further random insertion batches in U_{k+1} and extension batches in U_1. The wide phase completes either if the second phase completes or if the second phase gets truncated but H is in an $(\bar{\alpha}, e)$-good configuration following the phase (in the latter case, the wide phase is completed by performing a suitable extension sequence in U_1 that takes H to a $\bar{\alpha}$-dense configuration); otherwise the wide phase is truncated. We can prune every root hash function used by H in a $\bar{\alpha}$-dense configuration during the wide phase by deleting from its domain a subset of density at most $(\alpha_k/2)$ in U and make it (α_k/W_1)-biased since these hash functions are $(\alpha_k/2, W_1/2)$-wide. Let $\alpha = \alpha_k$, let W_2 and e_2 be suitably chosen positive integers, and let $\epsilon = (\alpha^2/64W_2m)$. The two subphases are performed as follows:

First Phase: We maintain a pair of sets U^1 and $U^2 \subseteq U^1$ that are initially the empty sets and repeat the following step as many times as possible:

Step: If there is an extension sequence $\bar{\sigma}$ of at most e_2 insertions in U_1 following which H is in a $\bar{\alpha}$-dense configuration and H's pruned root hash function has a domain D such that $D \backslash U^1$ is $(\alpha/4)$-dense in U, then perform $\bar{\sigma}$ and continue the step; otherwise

terminate the step. This ensures that, in any $\bar{\alpha}$-dense configuration of H during the next e_2 insertions in U_1 following the first phase, the domain of H's pruned root hash function will always intersect U^1 in a $(\alpha/4)$-dense subset of U. Let h be the pruned root hash function used by H following $\bar{\sigma}$, and let D' be a domain of density $(\alpha/4)$ in U that is disjoint to U^1 and on which h is defined. We insert D' into U^1, pick a random $(\epsilon, W_2/2)$-sample U^3 from $h|_{D'}$ and insert it into U_2, and, finally, insert a random $(\alpha e_2/8)$-subset S of U^3 into the dictionary. The set S is constructed by uniformly partitioning U^3 into $(\alpha e_2/8)$ equal blocks in any way and picking a random element from each block. This completes the description of the step.

Second Phase: Let U_{k+1} = the value of set U^2 at the end of the first phase, let L_2 = the set of level-2 locations from which the samples U^3 were constructed during the steps of the first phase, and let n' = the total number of random elements inserted into the dictionary during the first phase. The second phase is performed only if $U_{k+1} \neq \phi$; otherwise the second phase is truncated. We construct a new hashing scheme H_2 having a tower of universes $U_1 \supseteq U_2 \supseteq \cdots \supseteq U_k \supseteq U_{k+1}$ by collapsing the top two levels of H, by appending the set of locations L_2 to the root location of H, and by appending the remaining level-2 locations to the advice location of H. The working subuniverse of H_2, at any time, is the subset of U_{k+1} that H's pruned root hash function sends into the locations of L_2. Operations on H_2 are performed by simulating H's behaviour; the costs of operations on H_2 are defined analogous to H_1.

The second phase is recursively defined as the tail phase of adversary $A^{r-1}_{\bar{\beta}_2, n', w, b}$ against H_2; here $\bar{\beta}_2 = (\alpha_1, \alpha_2, \ldots, \alpha_k, \alpha_k/32)$.

This essentially completes the definition of the tail phase and of the adversary.

We now mention some details that had been omitted in the definition of the wide phase. During the wide phase, often, the root hash function has to be restricted to a subdomain such as when pruning a root hash function or when choosing an appropriate subdomain D' of a pruned root hash function h during a step of the first phase. These restrictions are performed in a *unique* way. We prune every root hash function in a unique way depending only on the hash function and on the bias β. We choose domain D' during a step of the first phase in a unique way depending only on the pruned hash function h and U^1. In the construction of the random subsets $S \subseteq U^3$ during the first phase, we uniformly partition U^3 in a unique way depending only on its value.

The size of the universe reduces by a factor of $128m/\alpha^2$ when carrying out the recursion during the wide phase. We need $|U| \geq (32)^{r^2}(128m)^r n/\alpha^{2r}$ to ensure that the universe has at least n elements by the time the adversary reaches the r-th level.

The parameters e, e_1, e_2, W_1, and W_2 are defined as functions of the adversary parameters r, α, and n using the following recurrence relations:

$$e^r(\alpha,n) = \begin{cases} e^{r-1}(\alpha/32, \alpha e_2^r(\alpha,n)/8) & \text{if } r \geq 2 \\ n & \text{otherwise} \end{cases} \tag{2.1}$$

$$e_1^r(\alpha,n) = e^{r-1}(\alpha/2,n) \tag{2.2}$$

$$e_2^r(\alpha,n) = \alpha W_1^r(\alpha,n)/(c_0 w) \tag{2.3}$$

$$W_1^r(\alpha,n) = W^{r-1}(\alpha/2,n) \tag{2.4}$$

$$W^r(\alpha,n) = \begin{cases} W_2^r(\alpha,n)\alpha/(c_1^r b) & \text{if } r \geq 2 \\ \alpha n/c_1 b & \text{otherwise} \end{cases} \tag{2.5}$$

$$W_2^r(\alpha,n) = W^{r-1}(\alpha/32, \alpha e_2^r(\alpha,n)/8) \tag{2.6}$$

The constants c_0 and c_1 used in these recurrences will be specified later. We obtain a recurrence involving W alone by eliminating the other parameters:

$$W^r(\alpha,n) = \begin{cases} \frac{W^{r-1}(\alpha/32, \frac{\alpha^2 W^{r-1}(\alpha/2,n)}{8c_0 w})\alpha}{c_1^r b} & \text{if } r \geq 2 \\ \alpha n/c_1 b & \text{otherwise} \end{cases}$$

This recurrence has the following solution:

$$W^r(\alpha,n) = n\frac{\alpha^{2^{r+1}-3}}{(64)^{(r-1)(5r-8)/2} c_1^{2^{r+1}-r-2} b^{2^r-1} (8c_0 w)^{2^{r-1}-1}}.$$

Back-substituting this solution into the above recurrences, we find the values of parameters e, e_1, e_2, W_1, and W_2 to be approximately $n(\alpha/wb)^{2^r}$.

The adversary definition requires all the parameters to be at least 1. We need n to be sufficiently large ($n \approx (wb/\alpha)^{2^r}$) to guarantee this.

We state some useful facts about the adversary. These facts can be proved using the definition of the adversary and induction.

Lemma 2.3 *i.* $W^r(\alpha,n) \leq W_2^r(\alpha,n)/2 \leq W_1^r(\alpha,n)/2$, *for all* r, α, *and* n.

ii. $(6/\alpha)e_2^r(\alpha,n) \leq e_1^r(\alpha,n)$, *for all* r, α, *and* n.

Lemma 2.4 *i. The maximum number of insertions performed by $A^r_{\bar{\alpha},n,w,b}$ is at most $3n$.*

ii. The maximum number of random insertion batches performed by $A^r_{\bar{\alpha},n,w,b}$ is at most $(34)^{r-1}/\alpha$.

iii. The maximum number of insertions performed by $A^r_{\bar{\alpha},n,w,b}$ during the second phase is at most e_2.

iv. The maximum number of insertions performed by $A^r_{\bar{\alpha},n,w,b}$ during the wide phase is at most $(6/\alpha)e_2 \leq e_1$.

Lemma 2.5 *i. A complete adversary sequence leaves the hashing scheme in an $\bar{\alpha}$-dense configuration; a truncated adversary sequence leaves the hashing scheme in an $(\bar{\alpha}, e)$-bad configuration.*

ii. Whenever the adversary performs a random insertion batch, the hashing scheme is in a $\bar{\alpha}$-dense configuration.

iii. The hashing scheme always uses $(\alpha/2, W_1/2)$-wide root hash functions in a $\bar{\alpha}$-dense configuration during the wide phase of the adversary.

2.3.3 Two Random Sampling Lemmas

The analysis of the wide phase uses two technical lemmas for analyzing the behaviour of random samples under a sequence of partial hash functions, so, first, in this section, we state and prove these lemmas.

We need some definitions before stating the lemmas. Consider partial hash functions from a universe U to an array of m locations. An *α-hash function,* for any $\alpha \in [0,1]$, is a partial hash function that is defined on a domain of density α in U. A sequence of partial hash functions is called a *hashtopy;* we think of a hashtopy as a deformation of a partial hash function over time, where time signifies the integers from 1 to the length of the hashtopy. For a hashtopy $\mathcal{H}$, $\mathcal{H}_t$ denotes the t-th partial hash function in the hashtopy. A hashtopy consisting of only α-hash functions is called an *α-hashtopy;* a *β-biased hashtopy* is defined analogously. Consider a hashtopy $\mathcal{H}$. A *location snap* of $\mathcal{H}$ is a pair (l,t), where $1 \leq l \leq m$ and $1 \leq t \leq |\mathcal{H}|$. A set $S \subseteq U$ *refreshes* a location snap (l,t) of $\mathcal{H}$ if, for some element $x \in S$, $\mathcal{H}$ sends x to location l at time t and $\mathcal{H}$ had

sent x to a different location the last time before t when x appeared in $\mathcal{H}$. The *refresh cost* incurred by a set $S \subseteq U$ in $\mathcal{H}$ is defined as the total number of locations snaps of $\mathcal{H}$ refreshed by S. The *oscillation* of an element $x \in U$ in $\mathcal{H}$ is defined as the refresh cost incurred by x in $\mathcal{H}$. The *oscillation* of $\mathcal{H}$ is defined as the mean oscillation of an element of U in $\mathcal{H}$. A *fixed element* of $\mathcal{H}$ is defined as an element whose oscillation in $\mathcal{H}$ equals 0; a *fixed subset* of $\mathcal{H}$ is defined analogously.

Our first lemma estimates the refresh cost incurred by a random sample in a β-biased hashtopy when the sample is constructed by uniformly partitioning the universe and sampling each partition independently.

Lemma 2.6 (Refresh Cost Lemma) *Consider a β-biased hashtopy $\mathcal{H}$ from a universe U to an array of m locations. Let $T = |\mathcal{H}|$, let $n \geq 1/\beta$ be a positive integer, and let $\bar{\omega}$ = the oscillation of $\mathcal{H}$. Partition U into n equal-sized blocks, and select a random n-subset R from U by picking a random element from each block. R incurs a refresh cost of at least $\bar{\omega}/4\beta$ in $\mathcal{H}$ with probability at least $(1 - e^{-(n\bar{\omega})/(16(T-1))})$.*

Our second lemma says that a random sample from a low oscillation α-hashtopy is likely to intersect the domains of the hash functions in the hashtopy in dense fixed subsets.

Lemma 2.7 (Density Lemma) *Let $\mathcal{H} = (h_1, h_2, \ldots, h_{p+q})$ be a $(1/p)$-hashtopy from a universe U to an array of m locations in which the domains of $h_1, h_2, \ldots, h_p$ are all disjoint. Suppose that the oscillation of $\mathcal{H}$ is at most $(1/2p)$. Let k be a positive integer, let $\epsilon \leq (1/4kmp^2)$, and suppose that $|U| \geq 1/\epsilon$. Construct a random sample R of U by picking a random (ϵ, k)-sample from each of the hash functions $h_1, h_2, \ldots, h_p$ and taking the union of these samples. Let F = the set of elements of U that are left fixed by $\mathcal{H}$. With probability $\geq (1 - 2qe^{-k/192})$, $F \cap R \cap dom(h_i)$ is a $(1/8p)$-dense subset of R, for all $i \geq p + 1$.*

We need some further lemmas for proving the above lemmas.

Lemma 2.8 (Martingale Lemma) *Let n be a positive integer and let $0 < \beta < \alpha < 1$. Let $X_1, X_2, \ldots, X_n$ be a sequence of random variables in the range $[0, 1]$ that are exposed*

one by one and satisfy the following relation:

$$E[X_1] + E[X_2|X_1] + E[X_3|X_1, X_2] + \cdots + E[X_n|X_1, X_2, \ldots, X_{n-1}] \geq \alpha n.$$

$Pr[X_1 + X_2 + \cdots + X_n \geq \beta n] \geq (1 - e^{-c_{\alpha,\beta} n})$, *where* $c_{\alpha,\beta}$ *was defined in the Binary Sampling Lemma.*

Proof. We generalize the proof of Hoeffding's inequality [20] that gives the lemma when the random variables are mutually independent and have fixed means. We show that

$$E[e^{h(X_1+X_2+\cdots+X_n)}] \leq (1 - \alpha + \alpha e^h)^n, \quad \text{for all } h \leq 0,$$

and then complete the proof of the lemma as in Hoeffding's inequality. We prove this statement by induction on n using Hoeffding's ideas, namely the convexity of e^x and the inequality between arithmetic and geometric means.

Basis. $n = 1$: For any x in the range $[0, 1]$, the convexity of e^{hx} gives

$$\begin{aligned} e^{hx} &\leq 1 - x + xe^h, \quad \text{so taking expectations, we get} \\ E[e^{hX_1}] &\leq 1 - E[X_1] + E[X_1]e^h \\ &\leq 1 - \alpha + \alpha e^h, \quad \text{since } h \leq 0. \end{aligned}$$

Induction Step. $n \geq 2$: The idea is to expose X_1 first and apply induction to the sequence $X_2, \ldots, X_n$:

$$\begin{aligned} E[e^{h(X_1+\cdots+X_n)}] &= E_{X_1}[e^{hX_1} E_{X_2,\ldots,X_n}[e^{h(X_2+\cdots+X_n)}|X_1]] \\ &\leq E_{X_1}[e^{hX_1}(1 - \frac{\alpha n - E[X_1]}{n-1}(1 - e^h))^{n-1}] \\ &\quad \text{(by induction)} \\ &\leq (1 - E[X_1](1 - e^h))(1 - \frac{\alpha n - E[X_1]}{n-1}(1 - e^h))^{n-1} \\ &\quad \text{(by convexity of } e^{hx}) \\ &\leq (1 - \alpha + \alpha e^h)^n \\ &\quad \text{(by the arithmetic and geometric means inequality).} \end{aligned}$$

This completes the proof of the lemma. □

Lemma 2.9 (Fractional Sampling Lemma) *Let $k_1, k_2, \ldots, k_t$ be positive integers with sum k, let $\alpha_1, \alpha_2, \ldots, \alpha_t$ be values in the interval $[0,1]$ with weighted mean $\alpha = (k_1\alpha_1 + \cdots + k_t\alpha_t)/k$, and let $0 < \beta < \alpha$. Consider a family of disjoint populations $U_1, U_2, \ldots, U_t$ of values in the interval $[0,1]$ with means $\alpha_1, \alpha_2, \ldots, \alpha_t$, respectively. Construct a random k-subset R by picking a random k_i-subset from U_i, for each i, and taking the union of these subsets. R has a mean greater than β with probability at least $(1 - e^{-c_{\alpha,\beta}k})$, where $c_{\alpha,\beta}$ was defined in the Binary Sampling Lemma.*

Proof. Suppose R is constructed by selecting a sequence of random values $Y_{11}, Y_{12}, \ldots, Y_{1k_1}$ from U_1, $Y_{21}, Y_{22}, \ldots, Y_{2k_2}$ from U_2, and so on. Define a set of independent random variables $\{X_{ij} | 1 \leq j \leq k_i\}$ as follows: X_{ij} is simply a random value chosen from U_i. A result of Hoeffding [20] (Theorem 4) says that

$$Ef(\sum_j Y_{ij}) \leq Ef(\sum_j X_{ij})$$

for any convex function f and for all i. For all real numbers h, we have:

$$\begin{aligned}
Ee^{h\sum_{ij} Y_{ij}} &= \prod_i Ee^{h\sum_j Y_{ij}} \\
&\quad \text{(as } \{\textstyle\sum_j Y_{ij} | i = 1, \ldots, t\} \text{ are mutually independent)} \\
&\leq \prod_i Ee^{h\sum_j X_{ij}} \\
&\quad \text{(by convexity of } e^{hx} \text{ and by Hoeffding's result)} \\
&= \prod_{ij} Ee^{hX_{ij}} \\
&\quad \text{(since } X_{ij} \text{ are mutually independent).}
\end{aligned}$$

We use this inequality and complete the proof of the lemma as in Hoeffding's inequality [20]. □

We are ready to prove the main lemmas of this section.

Proof of the Refresh Cost Lemma. The basic idea behind the proof is to construct the random sample R incrementally and use the Martingale Lemma. Let $U_1, U_2, \ldots, U_n$ denote the blocks of the partition of U. Construct R incrementally by randomly selecting elements from the blocks, one by one. Denote the set of first i elements added to R by R_i. For each $i \in \{1, 2, \ldots, n\}$, define a random variable $X_i =$

the refresh cost incurred by R_i in $\mathcal{H}$ minus the refresh cost incurred by R_{i-1} in $\mathcal{H}$. The refresh cost incurred by R in $\mathcal{H}$ equals $X_1 + X_2 + \cdots + X_n$.

In order to apply the Martingale Lemma, we construct a sequence of random variables $\{Y_i | i = 1, \ldots, n\}$ whose sum has the same distribution as the sum of the X_is for small values of the sums:

$$Y_i = \begin{cases} X_i & \text{if } X_1 + \cdots + X_i \leq \bar{\omega}/2\beta \\ T - 1 & \text{otherwise.} \end{cases}$$

Observe that $\sum_i Y_i = \sum_i X_i$ whenever $\sum_i X_i \leq \bar{\omega}/2\beta$ and that $\sum_i Y_i \geq \sum_i X_i$ always. It follows that the probability that $\sum_i Y_i$ exceeds $\bar{\omega}/4\beta$ equals the probability that $\sum_i X_i$ exceeds $\bar{\omega}/4\beta$. We normalize the Y_is to the interval $[0, 1]$ and form new random variables $Z_i = Y_i/(T-1)$, for all i. The following lemma says that the Z_is satisfy the condition of the Martingale Lemma.

Lemma 2.10

$$E[Z_1] + E[Z_2|R_1] + \cdots + E[Z_n|R_1, R_2, \ldots, R_{n-1}] \geq \frac{\bar{\omega}n}{2(T-1)}.$$

By the Martingale Lemma and using $n \geq 1/\beta$, we conclude that the probability that $\sum_i Z_i$ exceeds $\bar{\omega}/(4\beta(T-1))$ is at least $(1 - e^{-(n\bar{\omega})/(16(T-1))})$. The Refresh Cost Lemma follows immediately.

It remains to prove Lemma 2.10.

Proof of Lemma 2.10. We need some definitions. For any set $S \subseteq U$ and element $x \in U$, the *oscillation of x modulo S* is defined as the refresh cost incurred by $S \cup \{x\}$ minus the refresh cost incurred by S. For any pair of sets $S, T \subseteq U$, the *oscillation of T modulo S*, denoted $\bar{\omega}_S(T)$, is defined as the mean oscillation of an element of T modulo S.

Consider the construction of the sample R by adding elements from the blocks, one by one. Define:

$$\begin{aligned} \bar{\omega}_i &= (\bar{\omega}_{R_{i-1}}(U_i) + \bar{\omega}_{R_{i-1}}(U_{i+1}) + \cdots + \bar{\omega}_{R_{i-1}}(U_n))/n, \quad \text{and} \\ \kappa &= \max\{t | X_1 + X_2 + \cdots + X_t \leq \bar{\omega}/2\beta\}. \end{aligned}$$

We have

$$\bar{\omega}_{\kappa+1} \leq (n - \kappa)(T-1)/n = (E[Z_{\kappa+1}|R_\kappa] + E[Z_{\kappa+2}|R_{\kappa+1}] + \cdots + E[Z_n|R_{n-1}])(T-1)/n.$$

For any $i \leq \kappa$, we have

$$\begin{aligned}\bar{\omega}_i - \bar{\omega}_{i+1} &\leq \bar{\omega}_{R_{i-1}}(U_i)/n + \bar{\omega}_{R_{i-1}}(U) - \bar{\omega}_{R_i}(U) \\ &\leq E[Z_i|R_{i-1}](T-1)/n + \beta X_i \\ &\quad \text{(as each newly refreshed location snap at step } i \text{ reduces} \\ &\quad \text{the oscillation of at most a } \beta\text{-fraction of } U).\end{aligned}$$

Summing i from 1 to κ, we get

$$\begin{aligned}\bar{\omega} &= (\bar{\omega}_1 - \bar{\omega}_2) + (\bar{\omega}_2 - \bar{\omega}_3) + \cdots + (\bar{\omega}_\kappa - \bar{\omega}_{\kappa+1}) + \bar{\omega}_{\kappa+1} \\ &\leq (E[Z_1] + E[Z_2|R_1] + \cdots + E[Z_n|R_{n-1}])(T-1)/n + \beta(X_1 + X_2 + \cdots + X_\kappa) \\ &\leq (E[Z_1] + E[Z_2|R_1] + \cdots + E[Z_n|R_{n-1}])(T-1)/n + \bar{\omega}/2.\end{aligned}$$

The lemma follows from this inequality. □

This completes the proof of the Refresh Cost Lemma. □

Proof of the Density Lemma. Since the oscillation of $\mathcal{H}$ is at most $1/2p$, it follows that $|F| \geq (1 - 1/2p)|U|$. Thus $F \cap \text{dom}(h_j)$ has a density of at least $1/2p$ in U, for all j. We complete the proof of the lemma by showing that the intersection of R with any fixed $(1/2p)$-dense subset of U is a $(1/8p)$-dense subset of R with probability $\geq (1 - 2e^{-k/192})$.

Let S be any $(1/2p)$-dense subset of U. We want to estimate the probability that $R \cap S$ is a $(1/8p)$-dense subset of R. Let D denote the set of elements that are deleted from the domains of hash functions $h_1, \ldots, h_p$ when R is constructed by taking (ϵ, k)-samples of these hash functions. Since $\epsilon \leq (1/4kmp^2)$, it follows that $|D| \leq |U|/4p$. Define $S_1 = S \backslash D$; S_1 is a $(1/4p)$-dense subset of U. We show that $S_1 \cap R$ is likely to be $(1/8p)$-dense in R using two successive applications of the Fractional Sampling Lemma.

Let us review the construction of R. R is constructed by picking a random k-subset from each of the sets $U_i = \text{dom}(h_i) \backslash D$, for $i \leq p$, by forming the union R_1 of these subsets, by picking dense subsets of U that go into the same locations as the elements of R_1, and by forming the union of these dense subsets.

For each element $x \in \text{dom}(h_i)$, define

$$\text{value}(x) = \frac{|h_i^{-1}(h_i(x)) \cap S_1|}{|h_i^{-1}(h_i(x))|}.$$

We have

$$E_{x \in U_i}\text{value}(x) \quad \geq \quad E_{x \in \text{dom}(h_i)}\text{value}(x), \quad \text{and}$$

$$\frac{\sum_i E_{x \in U_i} \text{value}(x)}{p} \quad \geq \quad E_{x \in U} \text{value}(x) \geq 1/4p.$$

By the Fractional Sampling Lemma, it follows that R_1 has a mean value of at least $1/6p$ with probability $\geq (1 - e^{-k/72})$.

We estimate the probability that $R \cap S_1$ is $(1/8p)$-dense in R, given that R_1 has a mean value of at least $1/6p$. For all $i \in \{1, 2, \ldots, p\}$ and all $l \in \{1, 2, \ldots, m\}$, define

$$\begin{aligned} U_{i,l} &= h_i^{-1} l \quad \text{and} \\ k_{i,l} &= |R_1 \cap U_{i,l}|. \end{aligned}$$

Define the characteristic function χ_{S_1} of set S_1:

$$\chi_{S_1}(x) = \begin{cases} 1 & \text{if } x \in S_1 \\ 0 & \text{otherwise} \end{cases}$$

Since the mean value of R_1 is at least $1/6p$, it follows that

$$\frac{\sum_{i,l} k_{i,l} E_{x \in U_{i,l}} \chi_{S_1}(x)}{kp} \geq 1/6p.$$

Since R is formed by picking a random $(k_{i,l}\epsilon|U|)$-subset of $U_{i,l}$ and taking the union of these subsets, by the Fractional Sampling Lemma, it follows that $R \cap S_1$ is a $(1/8p)$-dense subset of R with conditional probability $\geq (1 - e^{-k/192})$, given that R_1 has a mean value $\geq 1/6p$.

We conclude that $R \cap S_1$ is a $(1/8p)$-dense subset of R with probability $\geq (1 - 2e^{-k/192})$. This completes the proof of the lemma. □

2.3.4 The Worst-case Lower Bound

In this section, we analyze the adversary defined in Section 2.3.2 and prove a lower bound of $\Omega(\log(\log n / \log b))$ on the worst-case cost of dictionary operations in the refresh cost multilevel hashing model, where n = the maximum dictionary size during the operation sequences.

Let H be any partial hashing scheme. We say that H *succeeds* on a sequence of operations σ performed by adversary $A^r_{\bar{\alpha},n,w,b}$ if σ is a complete sequence of operations, the maximum cost of an update operation in σ is at most w, and the maximum level of a dictionary element during σ is at most r.

The following lemma bounds the success probability of a partial hashing scheme against the adversary.

Lemma 2.11 *Let $\bar{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_k)$ be a sequence of values in the interval $[0,1]$, let $\alpha = \alpha_k$, and let $W = W^r(\alpha, n) \geq 1$. Let H be a partial hashing scheme that has a (Wb)-bit root location and b-bit locations at levels ≥ 2. H succeeds against $A^r_{\bar{\alpha},n,w,b}$ with probability $\leq e^{-Wb}$.*

The lemma gives a trade-off between the worst-case costs of search operations and update operations incurred by a hashing scheme. Let H be a multilevel hashing scheme in the refresh cost model with wordsize b that incurs a worst-case cost of r on searches and a worst-case cost of w on updates in processing sequences of $O(n)$ operations. H has the following trade-off between r, w, and b:

$$w = \Omega(n^{1/2^r}/b).$$

This trade-off gives a lower bound of $\Omega(\log(\log n/\log b))$ for $\max\{r, w\}$.

The rest of this section is devoted to proving Lemma 2.11. The proof is by induction on r. Let $U_1 \supseteq U_2 \supseteq \cdots \supseteq U_k = U$ be the tower of universes of H.

Basis. $r = 1$: Following a successful adversary sequence, the working subuniverse of H has density $\geq \alpha$ in U. Let $S(v)$ be a fixed α-dense working subuniverse used by H following a successful adversary sequence. By the Fractional Sampling Lemma, the random first batch has $\geq \alpha n/4$ elements in $S(v)$ with probability $\geq (1 - e^{-\alpha n/16})$. The root location can store at most $Wb = \alpha n/c_1$ distinct elements of U, so if $c_1 > 4$, some element of the first batch is stored at a level ≥ 2 with this probability. Since the root location can store at most $2^{\alpha n/c_1}$ distinct values v, the success probability of H is $\leq 2^{\alpha n/c_1} e^{-\alpha n/16}$. We choose $c_1 \geq 32$ so that the success probability of H is $\leq e^{-\alpha n/c_1}$.

Induction Step. $r \geq 2$: We estimate the probability that H succeeds against the adversary:

$$\begin{aligned} \Pr[\text{success}] &= \Pr[\text{narrow phase completes successfully}] + \\ &\quad \Pr[\text{wide phase completes successfully}] \end{aligned}$$

If the narrow phase completes successfully then the induced hashing scheme H_1 succeeds against its adversary $A^{r-1}_{\bar{\beta}_1,n,w,b}$. Thus, by induction, the first term is bounded by $e^{-W_1 b}$.

We bound the second term by showing that any fixed sequence of root hash functions $\mathcal{H} = (h_1, h_2, \ldots, h_l)$ used before the random insertion batches during the wide phase has a low probability of success; actually, here h_l denotes the root hash function used at the end of the adversary sequence and h_{l-1} denotes the root hash function used before the last random insertion batch. The hash functions h_i are all $(\alpha/2, W_1/2)$-wide, so they can be pruned to obtain (α/W_1)-biased hash functions g_i by deleting subsets of density $\leq \alpha/2$ in U from their domains. The g_is are $(\alpha/2)$-hash functions. We apply the incremental construction of the random subuniverse U_{k+1} during the first phase to the sequence $(g_1, g_2, \ldots, g_l)$ and determine the prefix $(g_1, g_2, \ldots, g_p)$ of the sequence from which U_{k+1} is constructed through random sampling. When sampling from a pruned hash function g_i, we restrict g_i to a subdomain D' of density $\alpha/4$ in U and sample from the restriction $f_i = g_i|_{D'}$. The domains of the restrictions f_i, for $1 \leq i \leq p$, are all disjoint and the union of these domains equals U^1. The domains of the hash functions g_j, for $j \geq p+1$, intersect U^1 in a $(\alpha/4)$-dense subset of U, since, otherwise, the construction of U_{k+1} would have also involved g_j. We restrict each hash function g_j, for $j \geq p+1$, to the subdomain $\mathrm{dom}(g_j) \cap U^1$ and obtain a $(\alpha/4)$-hash function f_j. Consider the hashtopy $\mathcal{F} = (f_1, f_2, \ldots, f_l)$ over universe U^1. Two cases arise:

Case 1. The oscillation of $\mathcal{F}$ is at least $\alpha/8$: By the Refresh Cost Lemma, since $\mathcal{F}$ is a (α/W_1)-biased hashtopy, the first batch of the adversary incurs a refresh cost of at least $W_1/32$ in $\mathcal{F}$ with probability at least $(1 - e^{-(n\alpha/128l)})$. We choose $c_0 > 192$ so that the total refresh cost available during the wide phase is at most $6e_2 w/\alpha < W_1/32$. The probability of success of $\mathcal{H}$ is $\leq e^{-(n\alpha/128l)}$.

Case 2. The oscillation of $\mathcal{F}$ is at most $\alpha/8$: U_{k+1} is formed by picking $(4\epsilon/\alpha p, W_2/2)$-samples from the hash functions f_i, for $i \leq p$, and taking the union of these samples; here we have scaled ϵ to convert densities from U to U^1. $\mathcal{F}$ is a $(1/p)$-hashtopy over U^1, the oscillation of $\mathcal{F}$ is at most $1/2p$, and $(4\epsilon/\alpha p) \leq (1/4kmp^2)$. Let F = the set of elements of U^1 that are left fixed by $\mathcal{F}$. By the Density Lemma, with probability $\geq (1 - 2le^{-W_2/384})$, $F \cap U_{k+1} \cap \mathrm{dom}(h_i)$ is $(\alpha/32)$-dense in U_{k+1}, for all $i \geq p+1$. We call this property of U_{k+1} as the *density property.*

Fix a sequence of insertions σ performed by the adversary prior to the wide phase and fix a subuniverse U_{k+1} chosen by the adversary with the density property. Under these conditions, if H succeeds against the adversary, then the induced hashing scheme

H_2 succeeds against its adversary $A^{r-1}_{\bar{\beta}_2,n',w,b}$; the second phase can not get truncated because U_{k+1} has a dense fixed intersection with $\mathrm{dom}(h_l)$. By induction, under the above conditions, H succeeds against the adversary using $\mathcal{H}$ with probability $\leq e^{-W_2 b}$.

In both cases, the probability that H succeeds using root hashtopy $\mathcal{H}$ is at most

$$\max\{e^{-(\alpha n/128l)}, (2l+1)e^{-W_2/384}\} \leq e^{-W_2/500},$$

for sufficiently large c_1. The total number of root hashtopies $\mathcal{H}$ available is at most $2^{Wb(34)^{r-1}/\alpha}$. If we let $W \leq (W_2\alpha/1000(34)^{r-1}b)$ (by making c_1 large enough), then the probability that H successfully completes the wide phase is at most $e^{-W_2/1000}$.

The probability that H succeeds against the adversary is at most $e^{-W_1 b} + e^{-W_2/1000} \leq e^{-Wb}$, since W is small relative to W_1 and W_2. This completes the proof of the lemma.

2.3.5 Amortization

In this section, we prove a lower bound of $\Omega(\log(\log n/\log b))$ on the amortized cost of dictionary operations in the refresh cost multilevel hashing model, where n = the maximum dictionary size during the operation sequences.

Any hashing scheme H with an amortized cost of r on searches and an amortized cost of w on updates can be converted into a hashing scheme H' with a worst-case cost of r on searches and an amortized cost of w on updates. H' simulates the behaviour of H in processing all the operations, but always stores the dictionary elements in the first r levels. A search operation is performed by H' by simulating H, but H' does not change the memory configuration even if H does. An update operation is performed by H' by simulating H and then compressing the dictionary to the first r levels by performing sufficiently many greedy searches that each have a search cost $\geq r+1$. Consider a sequence σ of u update operations performed on H'. σ translates into a sequence $\bar{\sigma}$ of u update operations and g greedy searches on H. Let w_i = the refresh cost of the i-th operation in $\bar{\sigma}$. The cost of $\bar{\sigma}$ on H is at least $\sum_i w_i + g(r+1)$ and at most $gr + uw$, so it follows that $\sum_i w_i \leq uw$. We conclude that H' incurs a worst-case cost of r on search operations and an amortized cost of w on update operations.

We complete the proof of Theorem 2.1 by showing that a hashing scheme incurs either a worst-case cost of r on searches or an amortized cost of $\Omega(n^{1/2^r}/2^{2^r}b)$ on updates in processing operation sequences of length $O(n)$. We modify the worst-case adversary

$A^r_{\bar{\alpha},n,w,b}$ by appropriately performing greedy insertion batches following random insertion batches. A configuration C of a partial hashing scheme H is said to be *w-amortized,* for a positive integer w, if the cost incurred by H in processing any sequence σ of update operations, starting from configuration C, is at most $|\sigma|w$. The new adversary $\bar{A}^r_{\bar{\alpha},n,w,b}$ is tailored against $(\bar{\alpha}, e)$-good partial hashing schemes that start processing the adversary sequence in a w-amortized configuration; here e is a suitably defined positive integer. Either the adversary performs a complete sequence of $O(n)$ insertions, or it gets truncated because the scheme has entered a $(\bar{\alpha}, e)$-bad configuration or because the scheme was not in a w-amortized configuration, initially.

We define adversary $\bar{A}^r_{\bar{\alpha},n,w,b}$ against a partial hashing scheme H; let α denote the last component of $\bar{\alpha}$. A *w-greedy insertion batch* performed against a hashing scheme H in a configuration C is defined to be a maximal sequence σ of insertions, starting from configuration C, during which H incurs an update cost of at least $w|\sigma|$. The adversary performs a random first batch just like the worst-case adversary, then performs a $(2w)$-greedy insertion batch, and, finally, performs the tail phase; the adversary announces truncation even before performing the first batch if the initial configuration of H is not w-amortized. The tail phase is defined recursively, essentially as before, and consists of a narrow phase, possibly, followed by a wide phase; in the case $r = 1$, as before, the tail phase is a suitable extension batch that takes H to an $\bar{\alpha}$-dense configuration. The narrow phase consists of the tail phase of the recursive adversary $\bar{A}^{r-1}_{\bar{\beta}_1,n,w,b}$ performed against the induced hashing scheme H_1 that is constructed as before. If the narrow phase gets truncated, then H_1 has entered an $(\bar{\alpha}, e_1)$-bad, $(2^{2^{r-2}}w)$-amortized configuration. Equivalently, H has entered a $(2^{2^{r-2}}w)$-amortized configuration such that H uses only $(\alpha_k/2, W_1/2)$-wide root hash functions in any $\bar{\alpha}$-dense configuration during the next e_1 insertions. The wide phase consists of a first phase followed by a second phase that are defined slightly differently from the worst-case adversary. During the first phase, following each batch of random insertions, we perform a $(2^{2^{r-2}+1}w)$-greedy insertion batch so that, at the end of the first phase, H is in a $(2^{2^{r-2}+1}w)$-amortized configuration. The second phase is recursively defined to be the tail phase of adversary $\bar{A}^{r-1}_{\bar{\beta}_2,n',2^{2^{r-2}}w,b}$ performed against the induced hashing scheme H_2 that is constructed as before. This completes the definition of $\bar{A}^r_{\bar{\alpha},n,w,b}$.

We define parameters e, e_1, e_2, W, W_1, and W_2 as functions of r, α, n, and w. Only

the definition of $e_2^r(\alpha, n, w)$ has to be modified since it depends on w:

$$e_2^r(\alpha, n, w) = \alpha W_1^r(\alpha, n, w)/(c_0 2^{2^{r-2}} w) \tag{2.7}$$

The recurrence for W becomes:

$$W^r(\alpha, n, w) = \begin{cases} \frac{W^{r-1}(\alpha/32, \frac{\alpha^2 W^{r-1}(\alpha/2, n, w)}{8c_0 2^{2^{r-2}} w}, 2^{2^{r-2}} w)\alpha}{c_1^r b} & \text{if } r \geq 2 \\ \alpha n / c_1 b & \text{otherwise} \end{cases}$$

This recurrence has the following solution:

$$W^r(\alpha, n, w) = n \frac{\alpha^{2^{r+1}-3}}{(64)^{(r-1)(5r-8)/2} c_1^{2^{r+1}-r-2} 2^{2^{2r-3}-2^{r-2}} b^{2^r-1} (8c_0 w)^{2^{r-1}-1}}.$$

Hence the values of the parameters is approximately $n(\alpha/2^{2^r} wb)^{2^r}$.

The Lemmas 2.3, 2.4, and 2.5 still hold for $\bar{A}^r_{\bar{\alpha},n,w,b}$ with the exception of Lemma 2.4, Parts i. and iv. We modify some parts of the lemmas as follows:

Lemma 2.3 *ii'. $(10/\alpha)e_2^r(\alpha, n, w) \leq e_1^r(\alpha, n, w)$, for all r, α, n, and w.*

Lemma 2.4 *i'. The maximum number of insertions performed by $\bar{A}^r_{\bar{\alpha},n,w,b}$ is at most $4n$. iv'. The maximum number of insertions performed by $\bar{A}^r_{\bar{\alpha},n,w,b}$ during the wide phase is at most $(10/\alpha)e_2 \leq e_1$.*

Lemma 2.5 *i'. A complete adversary sequence leaves the hashing scheme in an $\bar{\alpha}$-dense configuration; a truncated adversary sequence leaves the hashing scheme in a $(2^{2^{r-1}} w)$-amortized $(\bar{\alpha}, e)$-bad configuration.*

Lemma 2.11 still holds for $\bar{A}^r_{\bar{\alpha},n,w,b}$:

Lemma 2.12 *Let $\bar{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_k)$ be a sequence of values in the interval $[0, 1]$, let $\alpha = \alpha_k$, and let $W = W^r(\alpha, n, w) \geq 1$. Let H be a partial hashing scheme that has a (Wb)-bit root location and b-bit locations at levels ≥ 2. H succeeds against $\bar{A}^r_{\bar{\alpha},n,w,b}$ with probability $\leq e^{-Wb}$.*

This lemma gives a trade-off between the worst-case cost of searches, r, and the amortized cost of updates, w, incurred by a multilevel hashing scheme in the refresh cost

model in processing sequences of $O(n)$ operations:

$$w = \Omega(n^{1/2^r}/2^{2^r}b).$$

The lower bound of Theorem 2.1 follows from this trade-off and our procedure for converting an amortized hashing scheme into a hashing scheme with a worst-case cost guarantee for search operations.

Lemma 2.12 is proved in the same way as Lemma 2.11. The only part of the proof that changes due to amortization is Case 1 of the induction step. In this case the total refresh cost available during the wide phase is at most $10e_2 2^{2^{r-2}} w/\alpha$. We choose $c_0 > 320$ so that this quantity is less than $W_1/32$, which is the probable refresh cost incurred by the first batch. The rest of the proof remains as before.

Chapter 3

The Deque Conjecture

Splay is an algorithm for searching binary search trees, devised by Sleator and Tarjan, that reorganizes the tree by means of rotations. Sleator and Tarjan conjectured that Splay is, in essence, the fastest algorithm for processing any sequence of search operations on a binary search tree, using only rotations to reorganize the tree. Tarjan proved a special case of this conjecture, called the *Scanning Theorem,* and conjectured a more general special case, called the *Deque Conjecture.*

In this chapter, we prove tight bounds for some combinatorial problems involving rotation sequences on binary trees, derive a result that is a close approximation to the Deque Conjecture, and give two new proofs of the Scanning Theorem[1].

3.1 Introduction

We review the Splay Algorithm, its conjectures and previous works on them, and describe our results.

3.1.1 The Splay Algorithm and Its Conjectures

Splay is a simple, efficient algorithm for searching binary search trees, devised by Sleator and Tarjan [28]. A *splay* at an element x of a binary search tree first locates the element in the tree by traversing the path from the root of the tree to the element (called the *access path* of the element) and then transforms the tree by means of rotations in order

[1] The work of this chapter was reported in [30].

to speed up future searches in the vicinity of the element. The splay transformation moves element x to the root of the tree along its access path by repeating the following step (See Figure 3.1):

> **Splay step.**
> Let p and g denote, respectively, the parent and the grandparent of x.
> **Case 1.** p is the root: Make x the new root, by rotating the edge $[x, p]$.
> **Case 2.** $[x, p]$ is a left edge (*i.e.* an edge to a left child) and $[p, g]$ is a right edge, or vice versa: Rotate $[x, p]$; Rotate $[x, g]$.
> **Case 3.** Either both $[x, p]$ and $[p, g]$ are left edges, or both are right edges: Rotate $[p, g]$; Rotate $[x, p]$.

Sleator and Tarjan proved that Splay is, upto a constant factor, as efficient as the more complex traditional balanced tree algorithms for processing any sequence of binary search tree operations. They also showed that Splay actually behaves even faster on certain special kinds of sequences and conjectured that Splay is, upto a constant factor, the fastest rotation-based binary search tree algorithm for processing any sequence of searches on a binary search tree. We state this conjecture and some closely-related conjectures:

Conjecture 3.1 (Dynamic Optimality Conjecture [28]) *Let s denote an arbitrary sequence of searches of elements in a given n-node binary search tree. Define $\chi(s)$ equal to the minimum cost of executing sequence s on the tree using an algorithm that performs searches, incurring a cost equal to (1+the distance of the element from the root) on each search, and transforms the tree by means of single rotations, incurring unit cost per single rotation. Splay takes $O(n + \chi(s))$ time to process s.*

Conjecture 3.2 (Deque Conjecture [33]) *Deque operations on a binary tree transform the tree by inserting or deleting nodes at the left or right end of the tree. We perform deque operations on a binary tree using Splay as follows (See Figure 3.2):* POP *splays at the leftmost node and removes it from the tree;* PUSH *inserts a new node to the left, making the old tree its right subtree;* EJECT *and* INJECT *are symmetric operations performed at the right end. Splay takes $O(m + n)$ time to process a sequence of m deque operations on an n-node binary tree.*

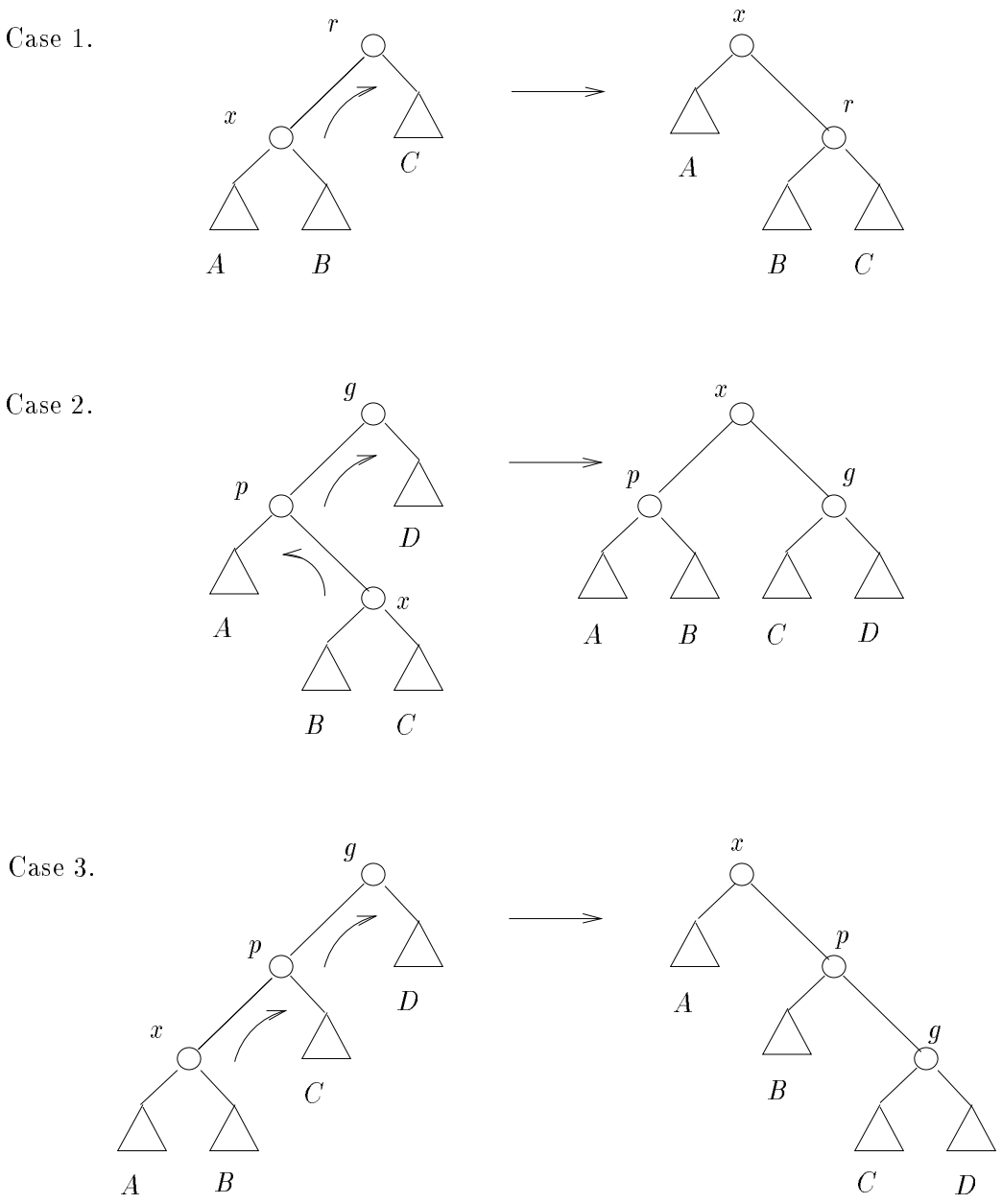


Figure 3.1: A splay step.

Conjecture 3.3 (Right Turn Conjecture [33]) *Define a* right 2-turn *on a binary tree to be a sequence of two right single rotations performed on the tree in which the bottom node of the first rotation coincides with the top node of the second rotation (See Figure 3.3). In a sequence of right 2-turns and right single rotations performed on an n-node binary tree, there are only $O(n)$ right 2-turns.*

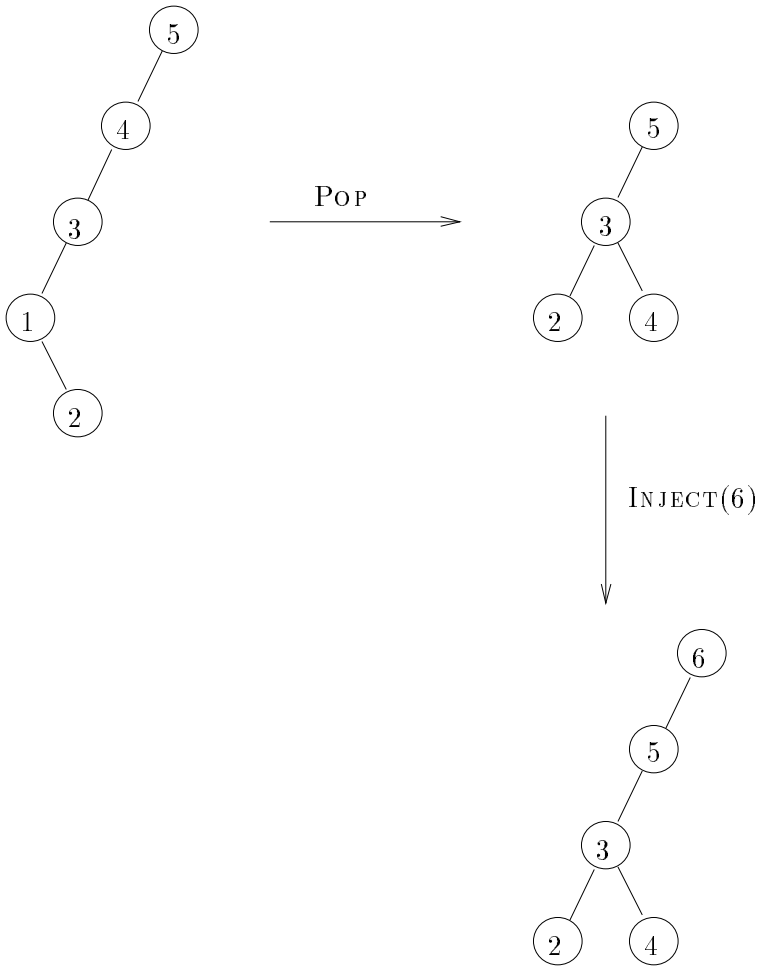


Figure 3.2: The deque operations.

The conjectures are related as follows. A stronger form of the Dynamic Optimality Conjecture that allows update operations as well as search operations implies the Deque Conjecture. The Right Turn Conjecture also implies the Deque Conjecture.

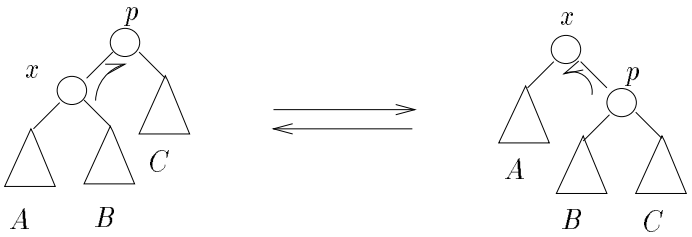
3.1.2 Terminology

We define the basic terminology used in the chapter. A *binary search tree* over an ordered universe is a binary tree whose nodes are assigned elements from the universe in *symmetric order:* that is, for any node x assigned an element e, the elements in the left subtree of x are lesser than e and the elements in the right subtree of x are greater than e. The path between the root and the leftmost node in a binary tree is called the *left path.* A tree in which the left path is the entire tree is called a *left path tree.* The edge between a node and its left child in a binary tree is called a *left edge.* The paths in a binary tree that comprise only left edges are called *left subpaths.* The *left depth* of a node in a binary tree is defined to be the number of left edges on the path between the node and the root. The terms *right path tree, right path, right edge, right subpath,* and *right depth* are defined analogously. A *single rotation* of an edge $[x, p]$ in a binary tree is a transformation that makes x the parent of p by transferring one of the subtrees of x to p (See Figure 3.3). A single rotation is called *right* or *left,* respectively, according to whether $[x, p]$ was originally a left edge or a right edge. A *rotation* on a binary tree is a sequence of single rotations performed on the tree. A rotation is called *left* or *right,* respectively, if it consists solely of left single rotations or solely of right single rotations. A *double rotation* on a binary tree is a sequence of two single rotations performed on the tree that have a node in common (as, for instance, by a splay step). A *left path rotation* on a binary tree is a right single rotation performed on the left path of the tree. A *right path rotation* is defined analogously.

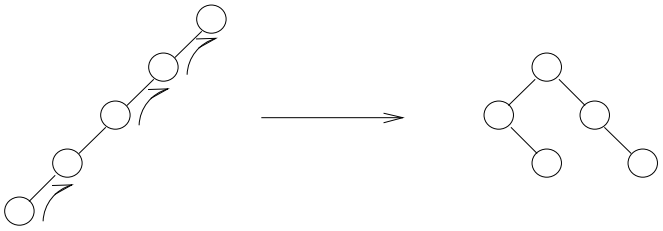
We define the Ackerman hierarchy of functions $\{A_i | i \geq 0\}$, its inverse hierarchy $\{\hat{\alpha}_i | i \geq 0\}$, and inverse functions $\bar{\alpha}$ and α of the Ackerman function as follows:

$$
\begin{aligned}
A_0(j) &= 2j \quad \text{for all } j \geq 1 \\
A_1(j) &= 2^j \quad \text{for all } j \geq 1 \\
A_i(j) &= \begin{cases} A_{i-1}(2) & \text{if } i \geq 2 \text{ and } j = 1 \\ A_{i-1}(A_i(j-1)) & \text{if } i \geq 2 \text{ and } j \geq 2 \end{cases} \\
\hat{\alpha}_i(n) &= \min\{k \geq 1 | A_i(k) \geq n\} \quad \text{for all } n \geq 1 \\
\bar{\alpha}(n) &= \min\{k \geq 1 | A_k(1) \geq n\} \quad \text{for all } n \geq 1 \\
\alpha(m, n) &= \min\{k \geq 1 | A_k(\lfloor m/n \rfloor) > \log n\} \quad \text{for all } m \geq n \geq 1
\end{aligned}
$$

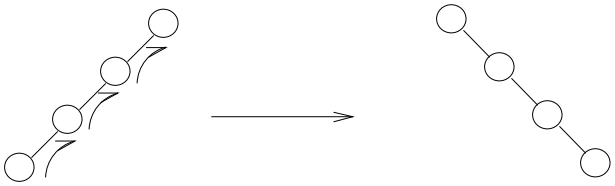
i. A single rotation



ii. A right 3-twist



iii. A right 3-turn



iv. A right 3-cascade

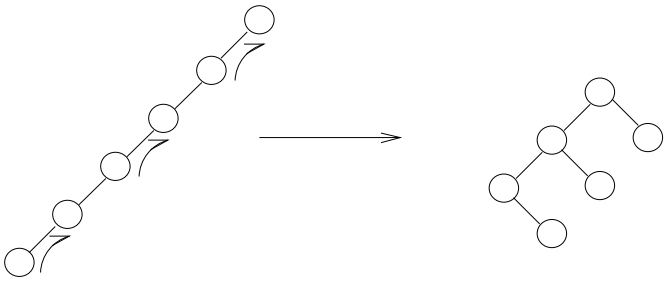


Figure 3.3: The various types of rotations.

The following table concretizes this definition:

i	0	1	2	3	4
$A_i(j)$	$2j$	2^j	$2^{2^{\cdot^{\cdot^{2}}}\}j}$		
$\hat{\alpha}_i(n)$	$\lceil n/2 \rceil$	$\lceil \log n \rceil$	$\leq \log^* n$	$\leq \log^{**} n$	$\leq \log^{***} n$
$\{n \mid \bar{\alpha}(n) = i\}$		$[1,2]$	$[3,4]$	$[5,16]$	$[17, 2^{2^{\cdot^{\cdot^{2}}}\}16}]$

3.1.3 Previous Works

Previous works on the Dynamic Optimality Conjecture have been mostly directed towards resolving its corollaries. Tarjan [33] proved that Splay requires linear time to sequentially scan the nodes of an n-node binary tree in symmetric order. This theorem, called the *Scanning Theorem,* is a corollary of all of the above conjectures. He also extended his proof to a proof of the Deque Conjecture when all the output operations are performed at one end of the tree. Lucas [22] obtained an $O(n\bar{\alpha}(n))$ upper bound for the Deque Conjecture when all the operations are output operations and the initial tree is a simple path between the leftmost and rightmost nodes. Building upon the work of Cole et al. [11], Cole [9,10] recently proved Sleator and Tarjan's Dynamic Finger Conjecture [28] for the Splay Algorithm which is a corollary of the Dynamic Optimality Conjecture. Wilber [36] gave two elegant techniques for lower-bounding $\chi(s)$. The techniques yield optimal lower bounds for some special sequences (such as $\chi(s) = \Omega(n \log n)$ for the bit-reversal permutation), but it is not clear how tight these lower bounds are for general sequences.

A related combinatorial question that has been studied is, how many single rotations are needed, in the worst case, to transform one n-node binary tree into another n-node binary tree? Culik and Wood [12] noted that $2n - 2$ rotations suffice and, later, Sleator et al. [29] derived the optimal bound of $2n - 6$ rotations for all sufficiently large n.

3.1.4 Our Results

Our work is directed towards resolving the Deque Conjecture. A good understanding of the powers of various types of rotations on binary trees would equip us with the necessary tools to tackle the conjecture. We prove almost tight upper and lower bounds on the maximum numbers of occurrences of various types of right rotations in a sequence of

right rotations performed on a binary tree. We study the following types of rotations (See Figure 3.3):

Right twist: For all $k \geq 1$, a right k-twist is a sequence of k right single rotations performed along a left subpath of a binary tree, traversing the subpath top-down.

Right turn: For all $k \geq 1$, a right k-turn is a right k-twist that converts a left subpath of k edges in a binary tree into a right subpath.

Right cascade: For all $k \geq 1$, a right k-cascade is a right k-twist that rotates every other edge lying on a left subpath of $2k-1$ edges in a binary tree.

A *right twist sequence* is a sequence of right twists performed on a binary tree. Define $Tw_k(n)$, $Tu_k(n)$ and $C_k(n)$, respectively, to be the maximum numbers of occurrences of k-twists, k-turns and k-cascades in a right twist sequence performed on an n-node binary tree. These numbers are well defined since a tree is transformed into a right path after $\binom{n}{2}$ right single rotations. We derive the following bounds for $Tw_k(n)$, $Tu_k(n)$ and $C_k(n)$:

	Upper bound	Lower bound
$Tw_k(n)$	$O(kn^{1+1/k})$	$\Omega(n^{1+1/k}) - O(n)$
$Tu_k(n)$	$\begin{cases} O(n\hat{\alpha}_{\lfloor k/2 \rfloor}(n)) & \text{if } k \neq 3 \\ O(n \log\log n) & \text{if } k = 3 \end{cases}$	$\begin{cases} \Omega(n\hat{\alpha}_{\lfloor k/2 \rfloor}(n)) - O(n) & \text{if } k \neq 3 \\ \Omega(n \log\log n) & \text{if } k = 3 \end{cases}$
$C_k(n)$		

The bounds for $Tu_k(n)$ and $C_k(n)$ are tight if $k \leq 2\bar{\alpha}(n) - 5$ and the bounds for $Tw_k(n)$ are nearly tight. The Right Turn Conjecture is refuted by the lower bound of $\Omega(n \log n)$ for $Tu_2(n)$[2]. We apply the upper bound for cascades to derive an $O((m+n)\bar{\alpha}(m+n))$ upper bound for the Deque Conjecture.

Another approach to the Deque Conjecture is to find new proofs of the Scanning Theorem that might naturally extend to the Deque Conjecture setting. We obtain a simple potential-based proof that solves Tarjan's problem [33] of finding a potential-based proof of the theorem, and an inductive proof that generalizes the theorem. The new proofs enhance our understanding of the Scanning Theorem, but, so far, have not led to a proof of the Deque Conjecture.

[2]S.R.Kosaraju has independently proved that $Tu_2(n) = \theta(n \log n)$. While his upper bound proof differs from ours, the lower bound constructions match.

The chapter is organized as follows. In Section 3.2, we prove the bounds for $Tw_k(n)$, $Tu_k(n)$ and $C_k(n)$. In Section 3.3, we derive the upper bound for the Deque Conjecture. In Section 3.4, we describe the new proofs of the Scanning Theorem.

3.2 Counting Twists, Turns, and Cascades

The two subsections of this section derive the upper and lower bounds for $Tw_k(n)$, $Tu_k(n)$ and $C_k(n)$.

3.2.1 Upper Bounds

All our upper bound proofs are based on a recursive divide-and-conquer strategy that partitions the binary tree on which the right twist sequence is performed into a collection of vertex-disjoint subtrees, called *block trees.* The root and some other nodes within each block are labeled *global* and the global nodes of all of the block trees induce a new tree called the *global tree.* Each rotation on the original tree effects a similar rotation either on one of the block trees or on the global tree. This allows us to inductively count the number of rotations of each type in the sequence.

We need the notion of *blocks* in binary trees [33]. Consider an n-node binary tree B whose nodes are labeled from 1 to n in symmetric order. A *block* of B is an interval $[i,j] \subseteq [1,n]$ of nodes in B. Any block $[i,j]$ of B induces a binary tree $B|_{[i,j]}$, called the *block tree* of block $[i,j]$, which comprises exactly the nodes i to j. The root of $B|_{[i,j]}$ is the lowest common ancestor of nodes i and j in B. The left child of a node x in $B|_{[i,j]}$ is the highest node in the left subtree of x in B which lies in block $[i,j]$. The right child of a node in $B|_{[i,j]}$ is defined analogously. Notice that, for the subtree rooted at any node of B, the highest node of the subtree which lies in block $[i,j]$ is unique whenever it exists: if two equally highest nodes exist, then their lowest common ancestor in the subtree would be higher than the two nodes, resulting in a contradiction. How does a rotation on B affect a block tree $B|_{[i,j]}$? If both of the nodes involved in the rotation are in $B|_{[i,j]}$, then the rotation translates into a rotation on $B|_{[i,j]}$ involving the same pair of nodes. Otherwise, $B|_{[i,j]}$ is not affected.

The functions Tw_k, Tu_k and C_k are superadditive:

Lemma 3.1 *For all $k \geq 1$ and $m \geq n \geq 1$, we have:*

a. $\lfloor m/n \rfloor Tw_k(n) \leq Tw_k(m)$,

b. $\lfloor m/n \rfloor Tu_k(n) \leq Tu_k(m)$, *and*

c. $\lfloor m/n \rfloor C_k(n) \leq C_k(m)$.

Proof. We prove Part a.; Parts b. and c. are similar. Given a right twist sequence S for an n-node binary tree B that comprises $Tw_k(n)$ right k-twists, construct a new tree of size $\lfloor m/n \rfloor n \leq m$ by starting with a copy of B and successively inserting a new copy of B as the right subtree of the rightmost node in the current tree $\lfloor m/n \rfloor - 1$ times. Since S can be performed on each of the copies of B one after another, there exists a right twist sequence with $\lfloor m/n \rfloor Tw_k(n)$ k-turns for a tree of size m. Part a. follows immediately. □

The upper bound for twists is the simplest to derive. Define $L_i(j) = \binom{i+j-1}{i}$ for all $i \geq 1$ and $j \geq 1$. The upper bound for $Tw_k(n)$ for n of the form $L_k(j)$ is given by:

Lemma 3.2 $Tw_k(L_k(j)) \leq k\binom{k+j-1}{k+1}$ *for all $k \geq 1$ and $j \geq 1$.*

Proof. We use double induction on k and j.

Case 1. $k = 1$ or $j = 1$: Straightforward.

Case 2. $k \geq 2$ and $j \geq 2$: The tree is partitioned into a left block of $L_{k-1}(j)$ nodes and a right block of $L_k(j-1)$ nodes. A right twist sequence on the tree translates into corresponding right twist sequences on the left and right block trees. We classify the k-twists in the original sequence into three categories and count the number of k-twists of each type separately. In the first type of k-twist, the lowest $k-1$ single rotations involve only left block nodes. Such a k-twist translates into a $(k-1)$-twist on the left block tree. Applying induction to the induced right twist sequence performed on the left block tree, we see that there are at most $(k-1)\binom{k+j-2}{k}$ k-twists of the first type in the original right twist sequence. Similarly, the number of k-twists that involve only right block nodes is at most $k\binom{k+j-2}{k+1}$. Consider a k-twist that does not belong to these two categories. The highest single rotation of such a twist must involve only right

block nodes; also, the lowest node involved in the twist must be a left block node. This implies that the highest node of the twist is a right block node that leaves the left path of its block as a result of the twist. Right rotations never add nodes to a block's left path, so the number of k-twists in the last category is at most the initial size of the left path of the right block $\leq L_k(j-1) = \binom{k+j-2}{k}$. It follows that the total number of right k-twists in the right twist sequence is bounded by

$$\begin{aligned}
&(k-1)\binom{k+j-2}{k} + k\binom{k+j-2}{k+1} + \binom{k+j-2}{k} \\
&= k\binom{k+j-2}{k} + k\binom{k+j-2}{k+1} \\
&= k\binom{k+j-1}{k+1}.
\end{aligned}$$

□

A simple calculation using the above lemma and Lemma 3.1a gives the upper bound for $Tw_k(n)$ for all n:

Theorem 3.1 *$Tw_k(n) \leq kn^{1+1/k}$ for all $k \geq 1$ and $n \geq 1$.*

Proof. Fix k and define $j = \min\{i | n \leq \binom{k+i}{k}\}$. Then we have

$$\begin{aligned}
T_k(n) &\leq T_k(\binom{k+j}{k})/\lfloor \binom{k+j}{k}/n \rfloor \quad \text{(By Lemma 3.1a)} \\
&\leq 2k\binom{k+j}{k+1}n/\binom{k+j}{k} \quad \text{(By Lemma 3.2)} \\
&\leq kn^{1+1/k}.
\end{aligned}$$

□

We derive the upper bounds for turns and cascades. It is easy to see that $Tu_1(n) = C_1(n) = \binom{n}{2} \leq n\hat{\alpha}_0(n)$. Let us prove that $Tu_2(n) = O(n\hat{\alpha}_1(n))$. Consider any right twist sequence performed on a binary tree B. We divide B into a left block $[1, \lfloor n/2 \rfloor]$ and a right block $[\lfloor n/2 \rfloor + 1, n]$. Every 2-turn either involves nodes from only one block (*intrablock*) or involves nodes from both blocks (*interblock*). An intrablock 2-turn effects a 2-turn in the corresponding block tree and gets counted in the right twist sequence for

the block tree. Every interblock 2-turn either adds a node to the right path of the left block tree or deletes a node from the left path of the right block tree (See Figure 3.4). Right rotations never remove nodes from a block's right path or add nodes to a block's left path, so the number of interblock 2-turns is at most $n-2$. This leads to the following recurrence for $Tu_2(n)$:

$$Tu_2(n) \leq \begin{cases} Tu_2(\lfloor n/2 \rfloor) + Tu_2(\lceil n/2 \rceil) + n - 2 & \text{if } n \geq 3 \\ 0 & \text{if } 1 \leq n \leq 2 \end{cases}$$

Solving the recurrence yields the desired bound for $Tu_2(n)$:

$$Tu_2(n) \leq n\lceil \log n \rceil - 2^{\lceil \log n \rceil} - n + 2 \leq n\hat{\alpha}_1(n).$$

With a slight modification the same proof works for 2-cascades also. An interblock 2-cascade either decreases the size of the left path of the right block tree or increases the number of left block nodes whose left depth relative to the block is at most 1 (See Figure 3.5). Right rotations never increase the left depth of a node, so the number of interblock 2-cascades is at most $n-3$. The bound $C_2(n) \leq n\hat{\alpha}_1(n)$ follows.

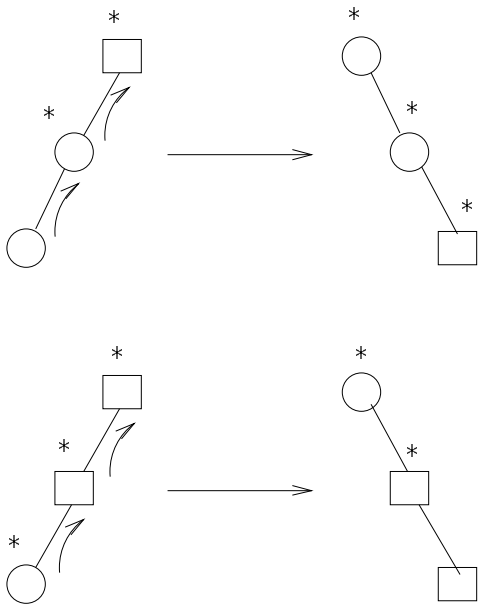


Figure 3.4: The two types of interblock right 2-turns. Circles denote left block nodes and squares denote right block nodes. The stars identify the nodes that lie on the left/right path of a block.

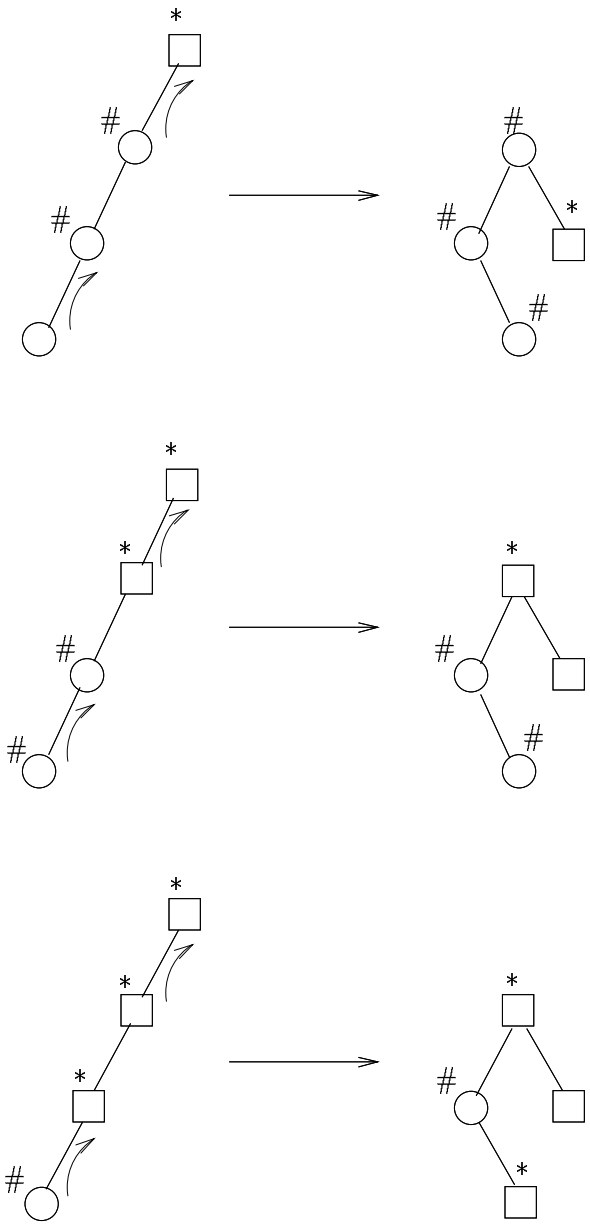


Figure 3.5: The three types of interblock right 2-cascades. The sharps identify the left block nodes that have a left depth of at most 1; the stars identify the right block nodes lying on the left path of their block.

In order to extend the above argument to k-turns and k-cascades for $k \geq 3$, we need an Ackerman-like hierarchy of functions $\{K_i | i \geq 1\}$:

$$\begin{aligned} K_1(j) &= 8j \quad \text{for all } j \geq 1 \\ K_2(j) &= 2^{4j} \quad \text{for all } j \geq 1 \\ K_i(j) &= \begin{cases} iK_{i-2}(\lfloor i/2 \rfloor) & \text{if } i \geq 3 \text{ and } j = 1 \\ K_i(j-1)K_{i-2}(K_i(j-1)/4)/2 & \text{if } i \geq 3 \text{ and } j \geq 2 \end{cases} \end{aligned}$$

The function K_i grows faster than the Ackerman function $A_{\lfloor i/2 \rfloor}$:

Lemma 3.3 *1.* $A_1^{(2)}(j) \leq K_3(j)$ *for all* $j \geq 1$.

2. $A_{\lfloor i/2 \rfloor}(j) \leq K_i(j)$ *for all* $i \neq 3$ *and* $j \geq 1$. □

The upper bound for $Tu_k(n)$ for n of the form $K_k(j)$ is given by:

Lemma 3.4 $Tu_k(K_k(j)) \leq 4jK_k(j)$, *for all* $k \geq 1$ *and* $j \geq 1$.

Proof. We use double induction on k and j.

Case 1. $1 \leq k \leq 2$: The lemma follows from the bounds $Tu_1(n) \leq n\hat{\alpha}_0(n)$ and $Tu_2(n) \leq n\hat{\alpha}_1(n)$.

Case 2. $k \geq 3$ and $j = 1$: We need to show that $Tu_k(K_k(1)) \leq 4K_k(1)$. Consider a binary tree B having $K_k(1)$ nodes on which a right twist sequence is performed. Divide B into a sequence of $K_{k-2}(\lfloor k/2 \rfloor)/2$ blocks of size $2k$ each. Each k-turn is of one of the following types:

Type A. All of the nodes involved in the k-turn belong to a single block: Since a block has only $2k$ nodes, there can be at most one such k-turn per block.

Type B. Some two nodes of the k-turn belong to a single block, but not all of the nodes of the turn are in that block: Let C denote the block tree of this block. The k-turn causes either an increase in the size of the right path of C, or a decrease in the size of the left path of C, or both. Hence the number of Type-B k-turns is at most $2K_k(1)$.

Type C. Each node of the k-turn belongs to a different block: To handle this case, we label the root of each block *global.* The global nodes in B induce a binary tree G, called the *global tree.* The root of G is identical to the root of B. The left child of a

node x in G is the highest global node in the left subtree of x in B. The right child of a node is defined similarly. It is easy to see that the left and right children of any node in G are unique. The effect on G of a rotation on B is analogous to the effect of such a rotation on a block tree of B: A rotation on B translates into a rotation on G if both of the nodes of the rotation are global; otherwise, G is unaffected. (If a rotation changes the root of a block then the global role passes from the old root to the new root but this does not affect the global tree.)

Suppose that the k-turn turns the left subpath $x_1 - x_2 - \cdots - x_{k+1}$ of B into a right subpath. Since all the x_is are from different blocks, the nodes $x_2, x_3, \ldots, x_k$ are all global. Therefore, the k-turn results in a $(k-2)$-turn on G (if x_1 or x_{k+1} is also global, then some right single rotations are also performed on G.) The number of $(k-2)$-turns that can be performed on G is at most

$$\begin{aligned} Tu_{k-2}(K_{k-2}(\lfloor k/2 \rfloor)/2) &\leq Tu_{k-2}(K_{k-2}(\lfloor k/2 \rfloor))/2 \quad \text{(By Lemma 3.1b)} \\ &\leq 2\lfloor k/2 \rfloor K_{k-2}(\lfloor k/2 \rfloor) \quad \text{(By the induction hypothesis)} \\ &< K_k(1). \end{aligned}$$

This gives an upper bound of $K_k(1)$ for the number of Type-C k-turns performed on B.

Summing together the above bounds for the three types of k-turns, we obtain a bound of

$$K_{k-2}(\lfloor k/2 \rfloor)/2 + 2K_k(1) + K_k(1) \leq 4K_k(1)$$

for the total number of k-turns in the right twist sequence. This completes Case 2.

Case 3. $k \geq 3$ and $j \geq 2$: We divide the binary tree on which the right twist sequence is executed into $K_k(j)/K_k(j-1)$ blocks of size $K_k(j-1)$ each. We split the k-turns into the three types defined in Case 2 and obtain the following tally for each type of turn:

Number of Type-A k-turns

$$\begin{aligned} &\leq \frac{K_k(j)}{K_k(j-1)} \cdot 4(j-1)K_k(j-1) \quad \text{(By the induction hypothesis)} \\ &\leq 4(j-1)K_k(j). \end{aligned}$$

Number of Type-B k-turns $\leq 2K_k(j)$.

Number of Type-C k-turns

$$\begin{aligned} &\leq Tu_{k-2}(K_{k-2}(K_k(j-1)/4)/2) \\ &\leq Tu_{k-2}(K_{k-2}(K_k(j-1)/4))/2 \quad \text{(By Lemma 3.1b)} \\ &\leq K_k(j-1)K_{k-2}(K_k(j-1)/4)/2 \quad \text{(By the induction hypothesis)} \\ &= K_k(j). \end{aligned}$$

Hence the total number of k-turns in the sequence is at most

$$4(j-1)K_k(j) + 2K_k(j) + K_k(j) < 4jK_k(j).$$

This finishes Case 3. □

Combining the above lemma with Lemmas 3.1b and 3.3, we obtain the upper bound for $Tu_k(n)$ for all k and n:

Theorem 3.2

$$Tu_k(n) \leq \begin{cases} 8n\hat{\alpha}_{\lfloor k/2 \rfloor}(n) & \text{if } k \neq 3 \\ 8n \log\log n & \text{if } k = 3 \end{cases} \quad \square$$

The upper bound for cascades is derived analogously:

Theorem 3.3

$$C_k(n) \leq \begin{cases} 8n\hat{\alpha}_{\lfloor k/2 \rfloor}(n) & \text{if } k \neq 3 \\ 8n \log\log n & \text{if } k = 3 \end{cases}$$

Proof. It suffices to prove Lemma 3.4 for $C_k(n)$: $C_k(K_k(j)) \leq 4jK_k(j)$, for all $k \geq 1$ and $j \geq 1$. Referring to the proof of Lemma 3.4, only the handling of Cases 2 and 3 has to be modified. Consider Case 2. As before, the blocks have size $2k$ each. The k-cascades are categorized as follows:

Type A. All nodes involved in the cascade belong to a single block: There is at most one Type-A cascade per block.

Type B. One of the cascade rotations involves a pair of nodes belonging to a single block, but not all of the nodes of the cascade are in that block: If the cascade rotates an edge that lies on the left path of some block, then the length of the left path of the block decreases by at least 1. Alternately, if the lowest three nodes involved in the cascade are from the same block, then the number of nodes in that block whose left depth is at most 1 increases. We conclude that the number of Type-B cascades falling under the above

categories is at most $2K_k(1) - K_{k-2}(\lfloor k/2 \rfloor)$. In every remaining Type-B k-cascade, only the lowest cascade rotation is intrablock and the lowest three nodes do not belong to the same block. Each such cascade behaves like a Type-C cascade in that it causes a $(k-2)$-cascade on the global tree (defined below) which accounts for it.

Type C. Each cascade rotation involves a pair of nodes belonging to different blocks: In this case for each block, in addition to the root of the block, we also label the left child of the root within the block global, if it exists; if the root has no left child, then the right child of the root is labeled global. Right rotations are propagated from the original tree to the global tree as described in Lemma 3.4 except in the following situation: When the edge joining the root and its left child, say l, in a block is rotated, the left child of l, say ll, now becomes global, and if ll is not adjacent to l in B, this results in a series of right single rotations on the global tree (See Figure 3.6). Under this definition of global tree, the $(k-2)$ interior rotations performed by any Type-C k-cascade are all global. Hence, each Type-C k-cascade translates into a right $(k-2)$-cascade and a sequence of right single rotations on the global tree. Therefore the total number of k-cascades in the sequence is at most

$$K_{k-2}(\lfloor k/2 \rfloor)/2 + 2K_k(1) - K_{k-2}(\lfloor k/2 \rfloor) + C_{k-2}(K_{k-2}(\lfloor k/2 \rfloor)) < 4K_k(1).$$

This completes Case 2 in the proof of Lemma 3.4 for cascades. Case 3 is handled similarly. □

3.2.2 Lower Bounds

The lower bound right twist sequences are inductively constructed by mimicking the divide-and-conquer strategy used to derive the upper bounds. The lower bound sequences always transform a left path tree into a right path tree. The tree is partitioned into a collection of vertex-disjoint block trees and a global tree is formed by selecting nodes from each block tree. The lower bound sequence for a tree is constructed by inductively constructing similar lower bound sequences for the block trees and for the global tree and weaving these sequences together. Actually, we first inductively construct a sequence of right twists as well as deletions having sufficiently many rotations of the given type and then remove the deletions to obtain the lower bound sequence.

We need some definitions. For all positive integers k, a *right k-twist-deletion sequence*

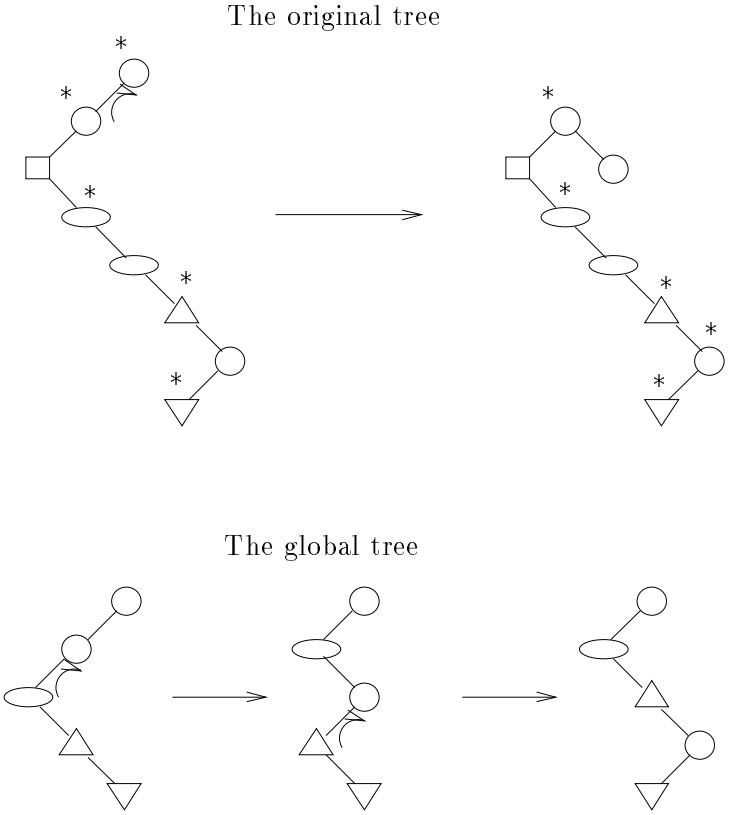


Figure 3.6: The effect of a right single rotation involving the root of a block and its left child within the block on the global tree. Circles denote the nodes of the block; other symbols denote the nodes from other blocks. The starred nodes in the original tree are the global nodes.

is defined to be an intermixed sequence of right single rotations, right k-twists and deletions of the leftmost node performed on a binary tree. *Right k-turn-deletion sequences* and *right k-cascade-deletion sequences* are defined analogously. Consider a right twist that is performed on some left subpath $x_0 - x_1 - \cdots - x_l$ of a binary tree, where x_0 is the lowest node on the subpath. x_0 is called the *base* of the twist. If x_k is the left child of a node y (say) in the tree, then the twist is called an *apex twist* and y is the *apex* of the twist. Otherwise, the twist is called *apexless.*

The lower bound for $Tw_k(n)$ for n of the form $L_k(j) = \binom{k+j-1}{k}$ is given by:

Lemma 3.5 *$Tw_k(L_k(j)) \geq \binom{k+j-1}{k+1}$ for all $k \geq 1$ and $j \geq 1$.*

Proof. For any pair of positive integers k and j, we inductively construct a right k-twist-deletion sequence for a left path tree of $L_k(j)$ nodes having the following properties:

1. The sequence deletes all the nodes from the tree.
2. A right k-twist always involves the leftmost node of the tree.
3. A deletion always deletes the root of the tree.
4. The sequence has exactly $\binom{k+j-1}{k+1}$ k-twists.

The removal of the deletions from the sequence would yield a right twist sequence having the desired number of k-twists.

Case 1. $k = 1$ or $j = 1$: Easy.

Case 2. $k \geq 2$ and $j \geq 2$: Divide the left path tree into a lower block of size $L_{k-1}(j)$ and an upper block of size $L_k(j-1)$. Recursively perform a right $(k-1)$-twist-deletion sequence, say $\hat{S}$, on the lower block tree. For each $(k-1)$-twist in $\hat{S}$, first rotate the edge joining the root of the lower block with its parent and then perform the $(k-1)$-twist on the block. This is equivalent to a k-twist involving the leftmost node of the tree. Each deletion in $\hat{S}$ is modified by first making the deleted node the root of the tree using right rotations and then deleting the node. By property 4 of $\hat{S}$, the number of $(k-1)$-twists in $\hat{S}$ is exactly $\binom{k+j-2}{k}$. The initial depth of the root of the lower block equals

$L_k(j-1) = \binom{k+j-2}{k}$. Since each $(k-1)$-twist in $\hat{S}$ reduces the depth of the root of the lower block by 1 and no other operation in $\hat{S}$ affects the depth, it is always possible to rotate the root of the lower block just before the execution of any $(k-1)$-twist in $\hat{S}$. The construction is completed by recursively performing a right k-twist-deletion sequence, say $\bar{S}$, on the upper block.

The sequence obviously satisfies properties 1–3. The total number of k-twists performed by the sequence equals

$$\begin{aligned}
&(\text{the number of } (k-1)\text{-twists in } \hat{S}) + (\text{the number of } k\text{-twists in } \bar{S}) \\
&= \binom{k+j-2}{k} + \binom{k+j-2}{k+1} \quad (\text{By the induction hypothesis}) \\
&= \binom{k+j-1}{k+1}.
\end{aligned}$$

This proves property 4. □

Combining Lemma 3.1a with the above lemma yields:

Theorem 3.4 *$Tw_k(n) \geq n^{1+1/k}/2e - O(n)$ for all $k \geq 1$ and $n \geq 1$.* □

We construct the lower bound sequences for turns. As in the upper bound proof, we need a new Ackerman-like hierarchy of functions. Define:

$$\begin{aligned}
B_1(j) &= j \quad \text{for all } j \geq 1 \\
B_2(j) &= 2^j - 1 \quad \text{for all } j \geq 1 \\
B_i(j) &= \begin{cases} 1 & \text{if } i \geq 3 \text{ and } j = 1 \\ ((i+1)jB_i(j-1)+1)B_{i-2}((i+1)jB_i(j-1)) & \text{if } i \geq 3 \text{ and } j \geq 2 \end{cases}
\end{aligned}$$

The function B_i grows essentially at the same rate as the Ackerman function $A_{\lfloor i/2 \rfloor}$:

Lemma 3.6 *1. $B_3(j) \leq A_1^{(2)}(2j)$, for all $j \geq 1$.*

2. $B_i(j) \leq A_{\lfloor i/2 \rfloor}(3j)$ for all $i \neq 3$ and $j \geq 1$. □

The lower bound for $Tu_k(n)$ for n of the form $B_k(j)$ is given by:

Lemma 3.7 *$Tu_k(B_k(j)) \geq (1/2)(j-3)B_k(j)$ for all $k \geq 1$ and $j \geq 1$.*

Proof. For any pair of positive integers k and j, we inductively construct a right k-turn-deletion sequence for the left path tree of $B_k(j)$ nodes having the following properties:

1. The sequence deletes all the nodes from the tree.
2. A right k-turn always involves the leftmost node of the tree.
3. A deletion always deletes the root of the tree.
4. The sequence comprises at least $(1/2)(j-3)B_k(j)$ apex k-turns. Further, if $k \geq 3$, there are no apexless k-turns in the sequence.
5. For any node x, the number of apex k-turns with base x is at most j.
6. For any node x, the number of apex k-turns with apex x is at most j.

Case 1. $k = 1$: The sequence repeatedly rotates the leftmost node to the root and deletes it.

Case 2. $k = 2$: Divide the left path tree into a lower left subpath comprising $2^{j-1}-1$ nodes, a middle node, and an upper left subpath comprising $2^{j-1}-1$ nodes. Recursively perform a right 2-turn-deletion sequence on the lower subpath. Modify each deletion in this sequence as follows: Perform a 2-turn on the subpath defined by the deleted node, say x, its parent (the middle node), and its grand parent; make x the root of the tree by successively rotating the edge joining it and its parent; delete x from the tree (this also deletes x from the lower subpath.) Next, delete the middle node which is currently the root of the tree. Finally, recursively perform a right 2-turn-deletion sequence on the upper subpath (See Figure 3.7).

This sequence performs $(j-2)2^{j-1}+1$ 2-turns of which exactly $j-1$ are apexless. Therefore the number of apex 2-turns is at least $(j-3)2^{j-1} \geq (1/2)(j-3)B_2(j)$. This proves property 4. The remaining properties are easy to check.

Case 3. $k \geq 3$ and $j = 1$: Just delete the only node in the tree.

Case 4. $k \geq 3$ and $j \geq 2$: Let $s = (k+1)jB_k(j-1)$. We inductively construct the sequences of operations performed on the block trees of the tree:

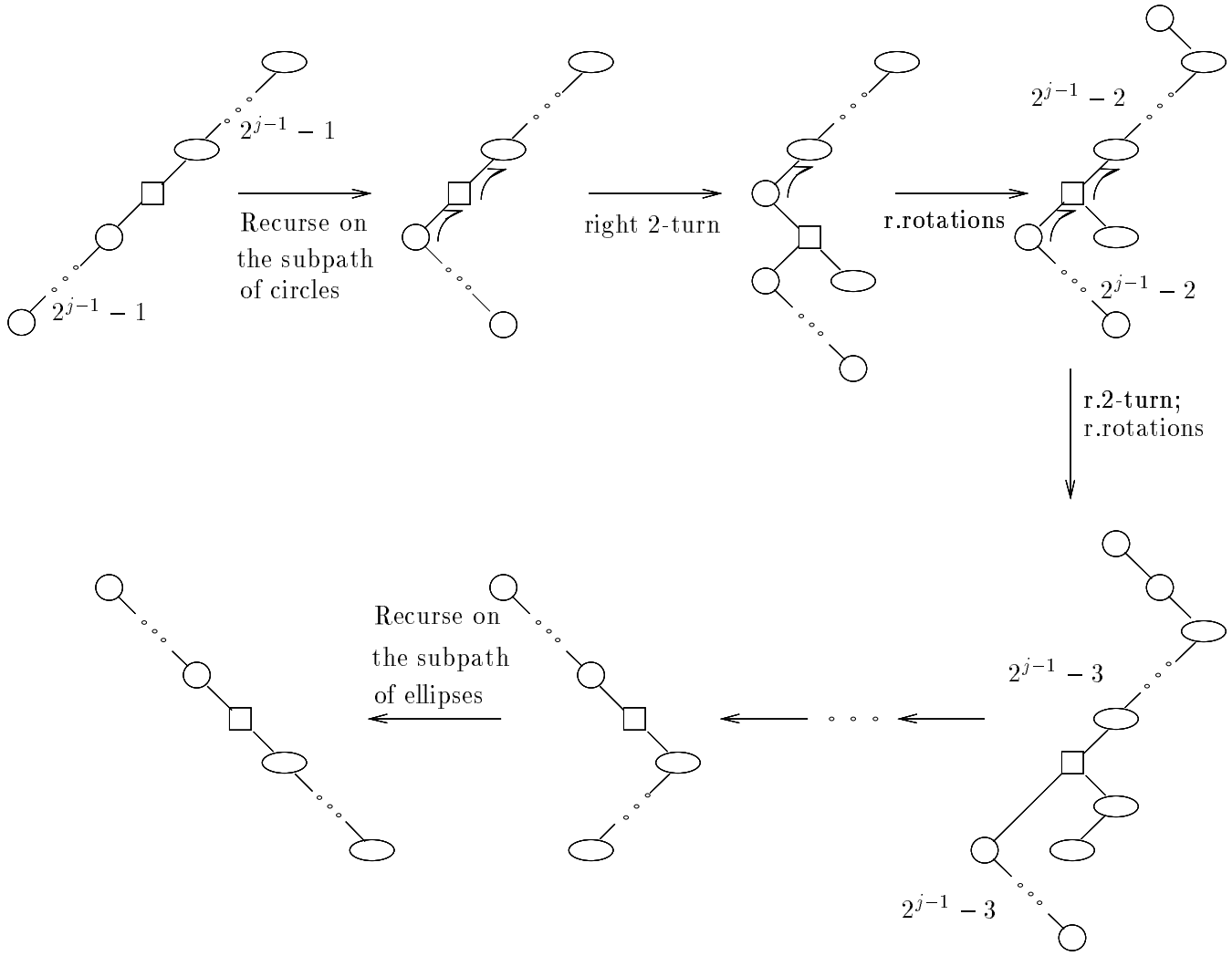


Figure 3.7: The lower bound construction for right 2-turns. The construction recursively transforms a left path tree of size $2^j - 1$ into a right path tree.

Lemma 3.8 *There exists a right k-turn-deletion sequence for a left path tree of size $s+1$ satisfying properties 1 and 2 and the following properties:*

$\bar{3}$. *A deletion that is not the last operation in the sequence always deletes the left child of the root.*

$\bar{4}$. *The sequence comprises at least $(1/2)(j-4)s+j$ right k-turns all of which are apex turns.*

$\bar{5}$. *For any node x, the number of apex k-turns with base x is at most $j-1$.*

$\bar{6}$. *For any node x, the number of apex k-turns with apex x is at most $j-1$.*

$\bar{7}$. *The root of the tree is always the rightmost node.*

Proof. Divide the nodes of the tree excluding the root into a sequence of $(k+1)j$ blocks of size $B_k(j-1)$ each. Perform a right k-turn-deletion sequence obeying properties 1–6 on the lowest block. (The inductive hypothesis implies the existence of such a sequence.) Denote this sequence by S. Each deletion in S except the last is modified by rotating the deleted node up the tree until it is adjacent to the root and then deleting it. The deletion of the last node in the block, say x_1, is implemented differently. x_1 is rotated up the tree until it is in contact with the root of the next higher block, say x_2. x_2 is rotated upwards in a similar fashion in order to make it adjacent to the root of the next higher block, say x_3. In this manner we create a left subpath $x_1 - x_2 - \cdots - x_{k+1}$ containing the roots of the lowest $k+1$ blocks. Next, a right k-turn is performed on this subpath and then x_1 is rotated up the tree and deleted. Following this, S is executed on the blocks of nodes $x_2, x_3 \ldots, x_{k+1}$ in succession. Each deletion is modified by first making the deleted node adjacent to the root and then deleting it. At the conclusion of this sequence of operations, all the nodes in the lowest $k+1$ blocks of the tree have been deleted and at least $(1/2)(j-4)(k+1)B_k(j-1)+1$ apex k-turns have been performed. The above sequence of operations is repeated on each group of $k+1$ consecutive blocks, choosing the lowest group of blocks currently in the tree each time. The final operation of the sequence deletes the root.

It is obvious that the right k-turn-deletion sequence constructed above satisfies properties 1,2,$\bar{3}$ and $\bar{7}$. Since there are j groups of $k+1$ blocks each, the total number of

apex k-turns executed by the sequence is at least $(1/2)(j-4)s+j$. Further, by property 4 of S, the sequence performs only apex k-turns. This proves property $\bar{4}$. Properties $\bar{5}$ and $\bar{6}$ are easy to show using properties 5 and 6 of sequence S. □

We construct a right k-turn-deletion sequence for the left path tree of size $B_k(j)$ satisfying the six properties. The tree is partitioned into $B_{k-2}(s)$ blocks of size $s+1$ each. The root of each block is labeled global. The global nodes form a global tree as described in the proof of Lemma 3.4. By the induction hypothesis, there exists a right $(k-2)$-turn-deletion sequence, say $\bar{S}$, for the global tree, satisfying properties 1-6. We construct the right k-turn-deletion sequence, denoted S, for the original tree by mapping each global tree operation in $\bar{S}$ onto a sequence of original tree operations, preserving the correspondence between the two trees. The following invariants define the relationships between the two trees:

A. Let B denote the block containing the leftmost node of the tree and let x denote the root of B. Suppose that d nodes have been deleted from B so far. Then,

   i. The number of apex $(k-2)$-turns performed so far on the global tree that had x as their base is exactly d.

   ii. Denote by $\hat{S}$ the right k-turn-deletion sequence constructed by Lemma 3.8 that deletes all the nodes from the left path tree of size $s+1$. Let T denote the tree that results when the prefix of $\hat{S}$ up to the d^{th} deletion is executed on the left path tree. The block tree of B equals T.

B. Consider any block B that does not contain the leftmost node of the tree. Let x denote the root of B. The block tree of B is a left path tree which is divided into the root and two subpaths. The nodes in the lower subpath, called *black nodes,* are the nodes in B that have participated in a k-turn. The nodes in the upper subpath are called *white nodes.*

   i. If b denotes the number of black nodes currently in B, then exactly b of the apex $(k-2)$-turns performed so far on the global tree had x as their apex.

C. If x is a global node with a right child y in the global tree, y is also the right child of x in the original tree.

D. If x is a global node with a left child y in the global tree, there is a left subpath $x = x_0 - x_1 - \ldots - x_{k+1} = y$ in the original tree such that $x_1, x_2, \ldots, x_k$ are the set of white nodes in the block of x.

Each global tree operation in $\bar{S}$ is simulated as follows:

Right rotation: Suppose that a global tree edge $[x, y]$, such that y is a left child of x, is rotated. In the original tree we repeatedly rotate the edge connecting y and its parent until x becomes the right child of y. Only invariants C and D are affected by the rotations. It is not hard to see that both these invariants are true after the last rotation.

Deletion: Suppose that a global node x is deleted. Since x is the root of the global tree, it is also the root of the original tree. Let d denote the number of nodes deleted so far from the block of x. We perform $\hat{S}$ (the sequence constructed by Lemma 3.8) on the block tree of x starting immediately after the d^{th} deletion. Each deletion is modified so that the deleted node is first made the root of the tree and then deleted. Invariant A.ii ensures that this is valid and that this will result in the deletion of all the nodes in the block of x from the tree. Therefore this sequence of operations reestablishes the correspondence between the global tree and the original tree.

Apexless $(k-2)$-turn: Break up the turn into a sequence of $k-2$ global rotations and simulate each global rotation as specified above.

Apex $(k-2)$-turn: Suppose that a global tree subpath $x_1 - x_2 - \cdots - x_{k-1}$ is turned and that x_1 is the base of the turn. Let x_0 denote the leftmost node in the block of x_1 and let x_k denote the parent of x_{k-1} in the original tree. We create the subpath $x_0 - x_1 - \ldots - x_k$ in the original tree and perform a k-turn on this subpath. This is implemented as follows:

1. Let d denote the number of nodes deleted so far from the block of x_1. Execute the segment of sequence $\hat{S}$ between the d^{th} and the $(d+1)^{\text{st}}$ deletions (excluding the deletions) on the block tree of x_1. By Lemma 3.8, property $\bar{3}$, this makes node x_0 the left child of x_1.
2. Rotate x_1 up the tree until its parent is x_2. Continuing in this fashion, rotate the nodes $x_2, x_3, \ldots, x_{k-1}$ upwards, creating a left subpath $x_0 - x_1 - \cdots - x_k$.

3. Perform a k-turn on the subpath $x_0 - x_1 - \cdots - x_k$.
4. Rotate x_0 up the tree, making it the root, and delete it.
5. Since x_k has become black due to the k-turn, the edge joining x_k and its left child is repeatedly rotated until the left child of x_k is not a global node.

Invariant B.i and property 6 of $\bar{S}$ guarantee that x_k is white at the beginning of this sequence of operations. Similarly, invariant A.i and property 5 of $\bar{S}$ guarantee that x_0 is well defined. Observe that all invariants are true at the end of the simulation.

The sequence of operations performed on the original tree during the simulation of $\bar{S}$ constitutes sequence S.

S deletes all the nodes in the original tree since $\bar{S}$ deletes all the nodes in the global tree. This proves that S satisfies property 1.

Properties 2 and 3 of S are apparent from the simulation procedure.

By Lemma 3.8, at least $(1/2)(j-4)s + j$ apex k-turns (*local turns*) are performed during the execution of $\hat{S}$ on any particular block. Hence the total number of local turns summed over all blocks is at least $(1/2)(j-4)sB_{k-2}(s) + jB_{k-2}(s)$. The number of turns involving global nodes (*global turns*) equals the number of $(k-2)$-turns in $\bar{S}$ which, by the induction hypothesis, is at least $(1/2)(s-3)B_{k-2}(s)$. Therefore the total number of k-turns in S is at least

$$\begin{aligned} & (1/2)(j-4)sB_{k-2}(s) + jB_{k-2}(s) + (1/2)(s-3)B_{k-2}(s) \\ = \;& ((1/2)(j-3)s + j - (3/2))B_{k-2}(s) \\ > \;& (1/2)(j-3)B_k(j). \end{aligned}$$

Evidently, every k-turn in S has an apex. This proves property 4.

For any node x, there is at most one global turn with x as the base since x is deleted from the tree immediately after the turn. By Lemma 3.8, there are at most $j-1$ local turns with x as the base. We conclude property 5. Property 6 is proved analogously. □

Combining the above lemma with Lemma 3.1b yields:

Theorem 3.5

$$Tu_k(n) \geq \begin{cases} (1/12)n\hat{\alpha}_{\lfloor k/2 \rfloor}(n) - O(n) & \text{if } k \neq 3 \\ (1/8)n \log\log n - O(n) & \text{if } k = 3 \end{cases}$$ □

The lower bound for cascades is given by:

Theorem 3.6

$$C_k(n) \geq \begin{cases} (1/12)n\hat{\alpha}_{\lfloor k/2 \rfloor}(n) - O(n) & \text{if } k \neq 3 \\ (1/8)n \log\log n - O(n) & \text{if } k = 3 \end{cases}$$

Proof. We modify the lower bound proof for $Tu_k(n)$ given above. Define:

$$\begin{aligned} B'_1(j) &= j \quad \text{for all } j \geq 1 \\ B'_2(j) &= 3.2^j - 2 \quad \text{for all } j \geq 1 \\ B'_i(j) &= \begin{cases} 1 & \text{if } i \geq 3 \text{ and } j = 1 \\ (4ijB'_i(j-1)+3)B'_{i-2}(4ijB'_i(j-1)) & \text{if } i \geq 3 \text{ and } j \geq 2 \end{cases} \end{aligned}$$

It is easy check that Lemma 3.6 holds for the new hierarchy $\{B'_i\}$. We prove the analogue of Lemma 3.7 for $C_k(n)$, which states that $C_k(B'_k(j)) \geq (1/2)(j-3)B'_k(j)$ for all $k \geq 1$ and $j \geq 1$. We construct a right k-cascade-deletion sequence that converts a left path tree of size $B'_k(j)$ into a right path tree and satisfies the analogues of properties 1–6 for cascades.

Case 1. $k = 1$ or $j = 1$: Easy.

Case 2. $k = 2$: Divide the left path tree into a lower subpath and an upper subpath, each having $3.2^{j-1} - 2$ nodes, and two middle nodes. The right 2-cascade-deletion sequence is constructed by recursing on the lower and upper subpaths in turn and performing a 2-cascade involving the deleted node and the middle nodes for each deletion in the first recursive step. The sequence comprises $(3j-4)2^{j-1} - j + 2 \geq (1/2)(3.2^j - 2)(j-3)$ apex 2-cascades and satisfies all the properties.

Case 3. $k \geq 3$ and $j \geq 2$: Let $s = 4kjB'_k(j-1)$. A *p,q-zigzag tree* is a tree that is constructed from a p-node left path tree by inserting a q-node left path tree as the right subtree of the leftmost node. We extend Lemma 3.8 to cascades and construct a right k-cascade-deletion sequence, say $\hat{S}$, for a $3,s$-zigzag tree that comprises $(1/2)(j-4)s+2j$ apex k-cascades. Each deletion in the sequence, except for the last two deletions, deletes the leftmost grandchild of the root. The proof divides the tree into $2j$ groups of $2k$ blocks each, each block, in turn, of size $B'_k(j-1)$, and recursively performs a right k-cascade-deletion sequence on each block, choosing the blocks in bottom-to-top order. A k-cascade is performed on the roots of the blocks within each group, yielding $2j$ extra cascades.

The tree is partitioned into $B'_{k-2}(s)$ blocks of size $s+3$ each. The global tree is constructed from the roots of the blocks and a right $(k-2)$-cascade-deletion sequence, say $\bar{S}$, satisfying properties 1–6 is recursively performed on it. The simulation of $\bar{S}$ on the original tree maintains the invariants A (with expression $s+3$ replacing $s+1$), C and D and the following modification of invariant B:

$\bar{\text{B}}$. Consider any block B that does not contain the leftmost node of the tree. Let x denote the root of B. Block B induces a connected subtree in the original tree. Further, if exactly b of the $(k-2)$-cascades performed so far on the global tree had x as their apex, then the block tree of B is a $(s-b+3), b$-zigzag tree.

The simulation of global tree operations other than apex $(k-2)$-cascades is as before. Consider an apex $(k-2)$-cascade in $\bar{S}$ involving a global tree subpath $x_1 - x_2 - \cdots - x_{2k-4}$, such that x_1 is the base of the cascade. Let y_1 and y_2 denote, respectively, the leftmost node in the block of x_1 and the left child of x_1. Let z_1 and z_2 denote, respectively, the parent and the grandparent of x_{2k-4} in the original tree. The global tree cascade is simulated on the original tree by creating the subpath $y_1 - y_2 - x_1 - x_2 - \cdots - x_{2k-4} - z_1 - z_2$ in the original tree, performing a k-cascade on this subpath, and finally deleting y_1 from the tree. We verify that the resulting sequence, say S, satisfies property 4:

$$\begin{aligned} \#\ k\text{-cascades in } S &= \#\ \text{local cascades} + \#\ \text{global cascades} \\ &\geq ((1/2)(j-4)s + 2j)B'_{k-2}(s) + (1/2)(s-3)B'_{k-2}(s) \\ &> (1/2)(j-3)B'_k(j). \end{aligned}$$

The rest of the properties of S are easy to check. This completes the proof of Lemma 3.7 for cascades. The theorem follows. □

3.3 An Upper Bound for the Deque Conjecture

In this section, we show that Splay takes $O((m+n)\bar{\alpha}(m+n))$ time to process a sequence of m deque operations on an n-node binary tree. We reduce a deque operation sequence to right cascade sequences on auxiliary trees and apply the upper bounds for cascades.

Define the *cost* of a deque operation or a right twist operation to be the number of single rotations performed by the operation.

The cost of a set of right cascades in a right twist sequence is given by:

Lemma 3.9 *Consider an arbitrary right twist sequence executed on an n-node binary tree. The total cost of any m right cascades in the sequence equals $O((m+n)\alpha(m+n,n))$.*

Proof. Let $l = 2\alpha(m+n,n)+2$. Split each of the m right cascades into a sequence of right l-cascades followed by a sequence of at most $l-1$ rotations. By Theorem 3.3, the total number of right l-cascades is at most $8n\hat{\alpha}_{\lfloor l/2 \rfloor}(n)$. This yields a bound of $(m + 8n\hat{\alpha}_{\lfloor l/2 \rfloor}(n))l$ for the number of rotations performed by the m right cascades. We bound $\hat{\alpha}_{\lfloor l/2 \rfloor}(n)$ as follows:

$$\begin{aligned} A_{\alpha(m+n,n)+1}(\lfloor m/n \rfloor + 2) &= A_{\alpha(m+n,n)}(A_{\alpha(m+n,n)+1}(\lfloor m/n \rfloor + 1)) \\ &\geq A_1(A_{\alpha(m+n,n)}(\lfloor (m+n)/n \rfloor)) \\ &\geq n. \end{aligned}$$

Therefore $\hat{\alpha}_{\lfloor l/2 \rfloor}(n) = \hat{\alpha}_{\alpha(m+n,n)+1}(n) \leq \lfloor m/n \rfloor + 2$. The lemma follows. □

Remark. Hart and Sharir [19] proved a result similar to Lemma 3.9 concerning sequences of certain path compression operations on rooted ordered trees. Their result can be derived from the analogue of Lemma 3.9 for turns by interpreting turns in a binary tree as path compressions on the rooted ordered tree representation of the binary tree. It is interesting that they also use ideas similar to blocks and global tree in their proof.

We estimate the cost of a sequence of deque operations performed at one end of a binary tree that also has left and right path rotations:

Lemma 3.10 *Consider an intermixed sequence of* POPs, PUSHs, *left path rotations and right path rotations performed on an arbitrary n-node binary tree. The total cost of* POP *operations equals $O((m+n)\bar{\alpha}(m+n))$, where m denotes the number of* POPs *and* PUSHs *in the sequence.*

Proof. We simplify the sequence through a series of transformations without undercounting POP rotations.

Simplification 3.1 *The first operation of the sequence is a* POP.

Transformation. Delete the operations preceding the first POP from the sequence and modify the initial tree by executing the deleted prefix of the sequence on it. □

Simplification 3.2 *The sequence does not contain* PUSH *operations.*

Transformation. For each PUSH operation, insert a node into the initial tree as the symmetric order successor of the last node that was popped before the PUSH. The PUSH operation itself is implemented by just rotating its corresponding node to the root through right rotations. □

Define a PARTIALPOP to be a sequence of arbitrarily many right 2-turns performed on the leftmost node of a binary tree followed by deletion of the node.

Simplification 3.3 *The sequence consists of only* PARTIALPOP*s and left path rotations; the lemma is true if the total cost of* PARTIALPOP *operations equals* $O((m+n)\bar{\alpha}(m+n))$*, where* m *denotes the number of* PARTIALPOP*s in the sequence and* n *denotes the size of the initial tree.*

Transformation. Normalize the tree by rotating the nodes on the right path across the root into the left path and consider the resulting sequence. □

Simplification 3.4 *The sequence comprises only right cascades; the lemma is true if the total cost of any* m *cascades in the sequence equals* $O((m+n)\bar{\alpha}(m+n))$*.*

Transformation. Instead of deleting nodes at the end of PARTIALPOPs, rotate them upwards to the right path. □

The lemma follows from Simplification 4 and Lemma 3.9. □

The upper bound for the Deque Conjecture is given by:

Theorem 3.7 *The cost of performing an intermixed sequence of* m *deque operations on an arbitrary* n*-node binary tree using splay equals* $O((m+n)\bar{\alpha}(m+n))$*.*

Proof. Divide the sequence of operations into a series of epochs as follows: The first epoch comprises the first $\max\{\lfloor n/2 \rfloor, 1\}$ operations in the sequence. For all $i \geq 1$, if the tree contains k nodes at the end of epoch i, then epoch $i+1$ consists of the

next $\max\{\lfloor k/2 \rfloor, 1\}$ operations in the sequence. The last epoch might consist of fewer operations than specified. It suffices to show that the cost of an epoch that starts with a k-node tree is $O(k\bar{\alpha}(k))$, since the sum of the sizes of the starting trees over all epochs is $O(m+n)$.

Consider an epoch whose initial tree, say T, has $k \geq 2$ nodes. Divide T into a left block of $\lfloor k/2 \rfloor$ nodes and a right block of $\lceil k/2 \rceil$ nodes. This partitioning ensures that neither block gets depleted before the epoch completes. The total cost of PUSHs and INJECTs is 0, since only rotations contribute to the operation cost. We show that the total cost of POPs is $O(k\bar{\alpha}(k))$. The same proof will apply to EJECTs.

A POP on T translates into a POP on the left block. The effect of a POP on the right block is a series of left path rotations. It is easy to see that the total number of single rotations performed by a POP is at most

(the number of single rotations performed by the POP on the left block) +
2(the number of left path rotations performed on the right block) + 2.

A PUSH operation on the tree propagates as a PUSH on the left block. An EJECT performs only right path rotations on the left block. An INJECT does not affect the left block. Hence by Lemma 3.10, the total number of single rotations performed by all the POPs on the left block equals $O(2\lfloor k/2 \rfloor \bar{\alpha}(2\lfloor k/2 \rfloor)) = O(k\bar{\alpha}(k))$. A left path rotation on the right block decreases the size of the left path of the block by 1. The initial size of this path is at most $\lceil k/2 \rceil$ and the size increases by at most 1 per deque operation. Therefore the total number of left path rotations performed on the right block due to POPs is at most $k+1$. This leads to an $O(k\bar{\alpha}(k))$ upper bound on the total cost of all the POPs performed during the epoch. □

3.4 New Proofs of the Scanning Theorem

In the two subsections of this section, we describe a simple potential-based proof of the Scanning Theorem and an inductive proof that generalizes the theorem.

3.4.1 A Potential-based Proof

The proof rests on the observation that a certain subtree of the binary tree, called the *kernel tree,* which is mainly involved in the splay operations always has a very nice shape. As the nodes of the original tree are accessed using splays, the kernel tree evolves through insertions and deletions of nodes at the left end, and left path cascades caused by the splays. Each node of the kernel tree is assigned a *unimodal potential function,* that is, a potential function that initially steadily increases to a maximum value and then steadily decreases once the node has progressed sufficiently through the kernel tree. The nice shape of the kernel tree guarantees that most of the nodes involved in each splay are in their potential decrease phase, enabling their decrease in potentials to pay for all the rotations and the small increase in potentials of the nodes in their potential increase phase.

We need some definitions. A binary tree is called *rightist* if the depths of the leaves of the tree increase from left to right. The *left* and *right heights* of a binary tree are defined, respectively, to be the depths of the leftmost and rightmost nodes. The *right inner height* of a node x is defined to be the depth of the successor of x within x's subtree if x has a right subtree and 0 otherwise.

We are ready to describe the proof. At any time during the sequence of splays, the set of nodes in the current tree that have been involved in a splay rotation form a connected subtree of the tree, called the *kernel tree,* whose root coincides with the root of the right subtree of the tree. Initially, the kernel tree is empty. The sequence of splays on the the original tree propagates into an intermixed sequence of n PUSHs and n POPs on the kernel tree, where a PUSH inserts a new node at the bottom of the left path of the tree and a POP splays at and deletes the leftmost node of the tree. Our goal is to show that the cost of the sequence of operations on the kernel tree equals $O(n)$. The theorem would then follow immediately.

The argument focuses on the sequence of operations on the kernel tree. Since the kernel tree is created by a sequence of PUSHs and POPs, it satisfies the following two properties:

1. The subtrees hanging from the left and right paths of the tree are rightist.
2. If T_1 and T_2 are subtrees hanging from the left path with T_1 to the left of T_2, then

$$\text{rightheight}(T_1) \leq \text{leftheight}(T_2).$$

This can be easily shown using induction.

We use the following potential function. The potential of the kernel tree equals the sum of the potentials of all its nodes. The potential of a node consists of an *essential* component and a *nonessential* component. For any node x, let $ld(x)$ and $rih(x)$ denote, respectively, the left depth and the right inner height of x. The essential potential of x equals $\min\{\lceil \log ld(x) \rceil, rih(x)\}$ unless x is on the right path in which case its essential potential equals 0. The essential potential of a node is a unimodal function of time, since the potential first monotonely increases from 0 until the node's right inner height overtakes the logarithm of its left depth and then monotonely decreases. The nonessential potential of x equals 2 units if x is not on the right path and x's left child has the same right inner height as x, and equals 0 otherwise.

We compute the amortized cost of kernel tree operations. PUSH has amortized cost 2, to provide for the nonessential potential that may be needed by the parent of the inserted node. Consider a POP. Let x denote the lowest node on the left path such that $\lceil \log ld(x) \rceil \leq rih(x)$. Every double rotation of a splay step that involves two nodes with identical right inner heights is paid for using the nonessential potential of the node leaving the left path. The number of remaining double rotations is at most $\lfloor ld(x)/2 \rfloor + \lceil \log(ld(x)+1) \rceil$. The first term accounts for the double rotations involving two proper ancestors of x and the second term accounts for the double rotations involving the descendents of x. Each latter category rotation increases the potential by at most 1, contributing to a net increase of at most $\lceil \log(ld(x)+1) \rceil$ units of potential. The halving of the left depths of the ancestors of x caused by the splay operation decreases the potential by exactly $ld(x) - 1$. The amortized cost of POP is therefore bounded by

$$\lfloor ld(x)/2 \rfloor + 2\lceil \log(ld(x)+1) \rceil - (ld(x) - 1) \leq 5.$$

We conclude that at most $7n$ double rotations, hence at most $15n$ single rotations, are performed by the sequence of operations on the kernel tree. This proves the Scanning Theorem.

3.4.2 An Inductive Proof

In this section, we describe an inductive proof of a generalization of the Scanning Theorem. The proof technique is similar to the method used to derive the upper bounds in Section 3.2.1. The binary tree is partitioned into blocks of constant size so that the total number of single rotations within blocks is linear. The induction is applied to a global tree consisting of a constant fraction of the tree nodes. Since a splay on the original tree translates into a much weaker rotation on the global tree, we have to incorporate the strength of the rotations into the inductive hypothesis.

We state the result. For any positive integer k and real number d, such that $1 \leq d \leq n$, a right k-twist is called *d-shallow* if the lowest node involved in the twist has a left depth of at most dk. Let $S^{(d)}(n)$ denote the maximum number of single rotations performed by d-shallow right twists in any right twist sequence executed on an n-node binary tree. We prove that $S^{(d)}(n) = O(dn)$. The Scanning Theorem follows from $S^{(2)}(n) = O(n)$.

We estimate the number of d-shallow right twists in a right twist sequence:

Lemma 3.11 *For any $d \geq 1$, the total number of d-shallow right twists in any right twist sequence is at most $4dn$.*

Proof. Consider any d-shallow right twist that rotates a sequence of edges, say $[x_1, y_1]$, $[x_2, y_2]$, ..., $[x_k, y_k]$, such that the left depths of the sequence of nodes x_k, y_k, x_{k-1}, y_{k-1}, ..., x_1, y_1 is nonincreasing. Let $ld(z)$ and $ld'(z)$ denote, respectively, the left depths of any node z before and after the twist. For all $i \in [\lceil k/2 \rceil, k]$, we have

$$\frac{ld'(x_i)}{ld(x_i)} = 1 - \frac{i}{ld(x_i)} \leq 1 - \frac{i}{kd} \leq 1 - \frac{1}{2d}.$$

In order to pay unit cost for a twist, we charge each node x_i, such that $i \in [\lceil k/2 \rceil, k]$, $\min\{2d/ld(x_i), 1\}$ debits. Let us prove that the total charge is at least 1. If $ld(x_{\lceil k/2 \rceil}) \leq 2d$, then $x_{\lceil k/2 \rceil}$ is charged 1 debit. Otherwise, we have $1 \geq 2d/ld(x_i) \geq 2/k$ for all $i \geq \lceil k/2 \rceil$. Since $\lfloor k/2 \rfloor + 1$ nodes are each charged at least $2/k$ debits, the net charge to all the nodes is at least 1.

Now, we bound the total charge to a node over the entire sequence. Call a node *deep* if its left depth is greater than $2d$ and *shallow* otherwise. Suppose that a node receives

a sequence of charges $2d/L_k, 2d/L_{k-1}, \ldots, 2d/L_0$ while it is deep. Then

$$L_i > \frac{2d}{(1-1/2d)^i} \quad \text{for all } i \geq 0.$$

Therefore the total charge to a node while it remains deep is at most

$$(1-1/2d)^k + (1-1/2d)^{k-1} + \cdots + 1 \leq 2d.$$

A node receives at most $2d$ debits while it is shallow. This implies that any node is charged at most $4d$ debits, giving a bound of $4dn$ for the total number of d-shallow right twists. □

The upper bound for $S^{(d)}(n)$ is given by:

Theorem 3.8 *$S^{(d)}(n) \leq 87dn$ for all $d \geq 1$ and $n \geq 1$.*

Proof. The proof uses induction on n.

Case 1. $n \leq 174d$: $S^{(d)}(n) \leq \binom{n}{2} \leq 87dn$.

Case 2. $n > 174d$: Divide the tree into a sequence of $\lceil n/K \rceil$ blocks such that each block except the first contains exactly $K = 29d$ nodes. The first block may contain fewer nodes. In each block except the first, the nodes with preorder numbers 1 to $4d$ within the block are global. The first block does not contain any global nodes. Notice that the global nodes of a block form a connected subtree within the block whose root coincides with the root of the block. Further, if the left path of the block contains more than $4d$ nodes then all global nodes lie on the left path of the block. Otherwise all nodes on the left path of the block are global. The global nodes in the tree form a global tree as in the previous upper bound constructions. The size of the global tree is at most $n/7.25$.

We analyze the effect of a original tree rotation on the global tree. An interblock right single rotation translates into a corresponding rotation on the global tree if both nodes of the rotation are global. Otherwise the global tree is not affected. The analysis of an intrablock right single rotation involves the following cases:

Local-local: The global tree is unaffected.

Local-global: Again, the global tree is not affected, but the global role is transferred from the global node to the local node.

Global-global: Let $[x, p]$ denote the rotated edge such that p is the parent of x. If the left subtree of x within the block contains only global nodes, then the rotation simply propagates to the global tree. Otherwise, p's global role is transferred to the node, say x', in p's block that had preorder number $4d + 1$ initially. Let p' denote the lowest ancestor of x' in the original tree which is global. The effect of the transfer of global role on the structure of the global tree is to contract edge $[x, p]$ and add a new edge $[x', p']$. We show that the same transformation is realizable through a series of right single rotations in the global tree. These rotations are performed by traversing the path from p to p' in the global tree as follows (See Figure 3.8):

> Start at edge $[p, x]$ and repeat the following operation until the last edge on the path is reached: If the next edge on the path is a left edge, move to the next edge; otherwise, rotate the current edge and move to the next edge after the rotation. Finally, if x' belongs to the right subtree of p' in the original tree, rotate the last global tree edge traversed.
>
> **Remark.** The operation performs all the rotations within the subtree of the global tree rooted at x. This is seen as follows. If $x = p'$, then no rotations are performed on the global tree. Otherwise, p' lies in the left subtree of x. Hence the successor of edge $[p, x]$ on the global tree path from p to p' is a left edge. This implies that the operation does not rotate edge $[p, x]$. Therefore all the rotations performed by the operation occur in the subtree of the global tree rooted at x.

At any during this traversal, contracting the current edge results in a tree that is identical to the tree obtained by contracting the edge $[x, p]$ in the initial tree, so it follows that the above series of rotations on the global tree correctly simulates the rotation of edge $[x, p]$.

In summary, a right single rotation of an edge $[x, p]$, such that p is the parent of x, either does not affect the global tree, or translates into a rotation of the edge $[x, p]$ in the global tree, or translates into a sequence of right single rotations on the subtree of the global tree rooted at x. The rotation is called *global* if it results in a rotation of the corresponding edge in the global tree and *local* otherwise.

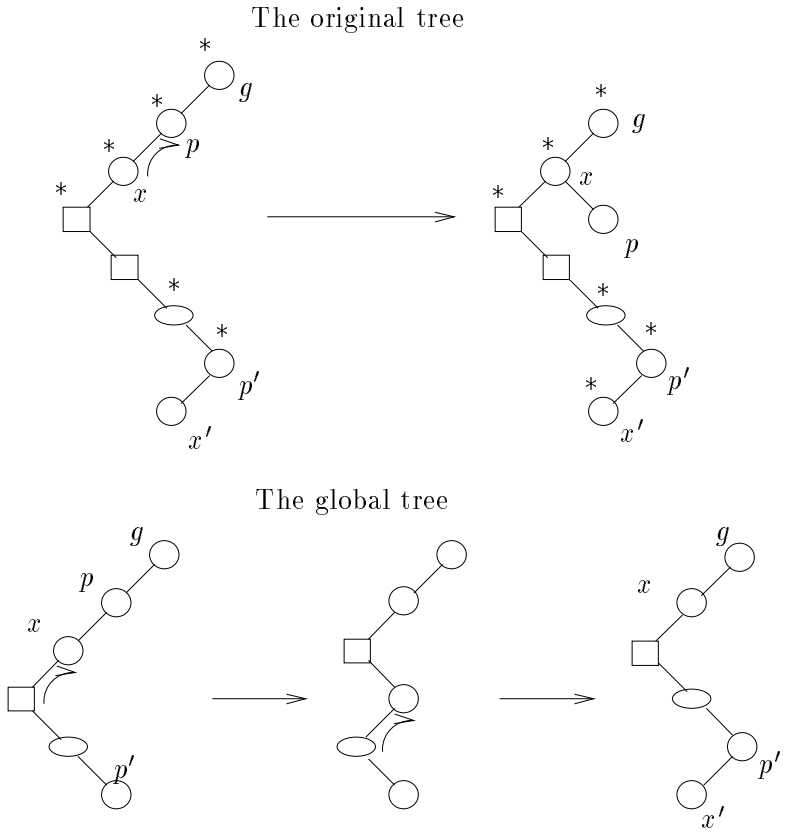


Figure 3.8: The transfer of global role in an intrablock global-global rotation. Circles denote the nodes of the block. The starred nodes of the original tree are the global nodes.

Consider the effect of a right twist in the original tree on the global tree. The sequence of single rotations on the global tree caused by the right twist comprises *l-rotations,* caused by local rotations in the twist, and *g-rotations,* caused by global rotations in the twist. The nodes involved in any l-rotation are distinct from the nodes involved in any previous g-rotation, hence we may transform the sequence of global tree rotations by moving each l-rotation before all the g-rotations without altering the net effect of the sequence on the global tree. The suffix of the sequence consisting of all the g-rotations defines a *global twist* on the global tree. In summary, the effect of a right twist in the original tree on the global tree is a right rotation followed by a global twist corresponding to the subsequence of global rotations in the twist.

We estimate the number of single rotations performed by d-shallow twists in a right twist sequence executed on the tree. Consider any d-shallow twist in the sequence. Define the *left path* of the twist to be the left path resulting from the contraction of the right edges on the access path of the lowest node involved in the twist. We classify the right single rotations performed by the twist as follows:

Type 1. Local, interblock rotation in which the top node is global: There is at most one Type-1 rotation per twist because the left subtree of the bottom node of the rotation consists of only local nodes.

Type 2. Local, interblock rotation in which the top node is local: The top node lies on the left path of its block and, since the node is local, it has $4d$ global ancestors within the block that lie on the left path of the twist. Notice that the top nodes of different Type-2 rotations belong to different blocks. Thus, if k_2 denotes the number of Type-2 rotations performed by the twist, then the left path of the twist contains at least $(4d+1)k_2$ edges. Since the number of edges on the left path of the twist is bounded by dk, we obtain that $k_2 \leq \lfloor k/4 \rfloor$.

Type 3. Local, intrablock rotation: For each Type-3 rotation, charge $(8/3)$ debits to the block in which the rotation is performed. If the number of Type-3 rotations is at least $(3k-4)/8$, the total charges to the blocks *plus* a charge of $(4/3)$ debits to the twist itself pays for all the rotations performed by the twist.

Type 4. Global rotation: Only the situation where the number of Type-3 rotations

is less than $(3k-4)/8$ needs to be considered. In this case at least

$$k - 1 - \lfloor k/4 \rfloor - (\lceil (3k-4)/8 \rceil - 1) = k - \lfloor k/4 \rfloor - \lfloor (3k+3)/8 \rfloor \geq 3k/8$$

global rotations are performed. Therefore the global twist performs at least $3k/8$ rotations on the global tree, and it is $(8d/3)$-shallow. If we charge each such global twist $(4/3)$ times the actual cost, then all the rotations can be paid for. This is seen as follows. Let k_3 and k_4 denote, respectively, the number of Type-3 and Type-4 rotations. Then, $k_3 + k_4 \geq 3k/4 - 1$. The total charge is $8k_3/3 + 4/3 + 4k_4/3$ which is minimized when $k_3 = 0$. When $k_3 = 0$, the total charge is at least $4/3 + (4/3)(3k/4 - 1) \geq k$.

Since each d-shallow twist is charged at most $4/3$ debits, the total charge to all the d-shallow twists is at most $16dn/3$ by Lemma 3.11. The total charge to a block of size s is at most $8\binom{s}{2}/3$. It follows that the total charge to all the blocks is at most $4nK/3 \leq 116dn/3$. By the inductive hypothesis, the total charge to all the $(8d/3)$-shallow global twists is at most $(4/3)(87)(8d/3)(n/7.25) = 128dn/3$. Therefore the sum total of all the charges is bounded by $16dn/3 + 116dn/3 + 128dn/3 \leq 87dn$, completing the induction step. □

Chapter 4

Testing Set Equality

The problem of maintaining a dynamic collection of sets under various operations arises in numerous applications. A natural application is the implementation of high-level programming languages like SETL that support sets and permit operations such as equality, membership, union, intersection, etc. on them. The general problem of efficiently maintaining sets under all of these operations appears quite difficult. This chapter describes a fast data structure for maintaining sets under equality-tests and under creations of new sets through insertions and deletions of elements[1].

4.1 Introduction

The *Set Equality-testing Problem* is to maintain a collection of sets over a finite, ordered universe under the following operations:

- EQUAL(S, T): Test if $S = T$.
- INSERT(S, x, T): Create a new set $T = S \cup \{x\}$.
- DELETE(S, x, T): Create a new set $T = S \backslash \{x\}$.

The collection initially contains just the empty set. We would like to devise a data structure for this problem that tests equality of sets in constant time and executes the remaining operations as fast as possible, under this constraint.

[1] The work of this chapter was reported in a joint-paper with Robert E. Tarjan [31].

If sets are represented by *unique* storage structures, then equality-testing of a pair of sets can be implemented in constant time by just checking whether they are represented by a single storage structure; *uniqueness* simply means that all the instances of a set are represented by a single storage structure. Following this natural approach, several people have devised unique storage representations for sets that allow constant time equality-tests and can be updated efficiently. Wegman and Carter [35] gave a randomized signature representation for sets that can be updated in constant time and constant space but errs with a small probability during equality-tests. Pugh [26] and Pugh and Tietelbaum [27] gave an error-free randomized binary trie representation for sets that can be updated in $O(\log n)$ expected time and $O(\log n)$ expected space, where n denotes the size of the updated set. Their data structures also support union and intersection of sets, although less efficiently. Yellin [40] gave a deterministic binary trie representation of sets that can be updated in $O(\log^2 m)$ time and $O(\log m)$ space, where m denotes the total number of updates.

We devise a deterministic data structure for the Set Equality-testing Problem requiring $O(\log m)$ amortized time and $O(\log m)$ space per update operation. The data structure is based on a solution to a more fundamental problem involving S-expressions. S-expressions [5] constitute the staple data type of programming language LISP. An *S-expression* is either an *atom* (signifying a number or a character string) or a pair of S-expressions. An atom S-expression is represented in storage by a node; a pair S-expression is represented by a node with left and right pointers that point to nodes representing the component S-expressions. We store S-expressions uniquely, i.e. all instances of an S-expression are represented by a single node. CONS(s_1,s_2) returns the S-expression (s_1.s_2). A *cascade* of CONS operations is a sequence of CONS operations in which the result of each CONS operation is an input to the next CONS operation. For instance,

$$\begin{aligned} s_1 &:= \textsc{Cons}(s_0, t_0) \\ s_2 &:= \textsc{Cons}(s_1, t_1) \\ &\vdots \\ s_f &:= \textsc{Cons}(s_{f-1}, t_{f-1}) \end{aligned}$$

is a cascade of f CONS operations. The S-expression problem in question is to devise a data structure for efficiently implementing cascades of CONS operations on uniquely stored S-expressions.

Unique storage of S-expressions makes CONS operations expensive. Given a pair of S-expressions, a CONS operation has to check whether there is a third S-expression in the collection with these S-expressions as its component S-expressions. Viewing the collection of S-expressions as a dictionary, this is equivalent to performing a search operation, possibly followed by an insertion, on the dictionary. Single CONS operations can be implemented in $O(\sqrt{\log F})$ time and $O(1)$ amortized space or, alternately, in $O(1)$ time and $O(F^{\epsilon})$ space, where F denotes the total number of CONS operations performed and ϵ is any positive constant. This implementation is based on Willard's data structure [37] for maintaining a dictionary in a small universe. Universal hashing [8] and dynamic perfect hashing [13] offer alternate implementations that require $O(1)$ randomized amortized time and $O(1)$ amortized space per CONS operation.

We develop a data structure that performs a cascade of f CONS operations in $O(f + \log m_c)$ amortized time, where m_c denotes the total number of cascades performed. The total space used is proportional to the number of distinct S-expressions present. This means that CONS operations can be implemented in constant amortized time and constant space in situations where these operations occur in long cascades. Our set-equality-testing data structure is an immediate corollary of this result. When sets are represented by binary tries, an update operation translates into a cascade of at most $\log m$ CONS operations and requires $O(\log m)$ amortized time using this data structure. Many list-oriented functions in functional languages (LISP, for instance) involve cascades of CONS operations and can be implemented efficiently using this method; function APPEND is a typical example:

$$
\begin{array}{lcl}
\textsc{Append}([v_1, v_2, \ldots, v_k], [w_1, w_2, \ldots, w_l]) \equiv & & \\
\quad s_1 & := & \textsc{Cons}(v_k, [w_1, w_2, \ldots, w_l]) \\
\quad s_2 & := & \textsc{Cons}(v_{k-1}, s_1) \\
\quad \vdots & & \\
\quad \textbf{Result} & := & \textsc{Cons}(v_1, s_{k-1})
\end{array}
$$

The chapter is organized as follows. In Sections 4.2 and 4.3, we describe the data structure for equality-testing of sets and analyze its performance. In Section 4.4, we discuss directions for further work.

4.2 The Data Structure

We reduce the Set Equality-testing Problem to the problem of implementing cascades of CONS operations on uniquely stored S-expressions. The elements seen so far are numbered in serial order and define the current universe $U = [1, |U|]$. Each set is represented by a binary trie [21] in this universe. The binary trie representing a set S is an S-expression that stores the elements of S as atoms and is defined recursively. Let $2^p < |U| \leq 2^{p+1}$. A singleton set is represented by an atom and the empty set, by the atom NIL. If $|S| \geq 2$, then S is represented by a pair $(s_1.s_2)$, where s_1 and s_2 are, respectively, the S-expressions representing subsets $S \cap [1, 2^p]$ and $S \cap [2^p + 1, |U|]$ in their respective subuniverses. We store S-expressions uniquely so that two sets are equal if and only if their S-expressions are represented by a single node. A set update operation translates into a cascade of at most $\log |U| \leq \log m$ CONS operations, which can be implemented in $O(\log m)$ amortized time and $O(\log m)$ space using a method described below; m denotes the total number of update operations.

We describe an efficient data structure for performing cascades of CONS operations on uniquely stored S-expressions. The data structure requires $O(f + \log m_c)$ amortized time to perform a cascade of f CONS operations, where m_c denotes the total number of cascades performed. Consider the collection of nodes representing S-expressions. Number these nodes serially in their order of creation. A *parent* of a node v is defined to be a node that points to v. Each node v maintains a set parents(v) of all its parents. Each parent $p \in$ parents(v) is assigned a key equal to (serial#(w), b), where w is the other node (besides v) pointed to by p, and b equals 0 or 1 depending on whether the left pointer of p points to v or not. To perform a CONS operation on two nodes, v and w, we search the set parents(v) using the key (serial#(w), 0) and return the matching parent. If there is no matching parent, we create a new node p with pointers to v and w, set parents(p) to empty, insert p into parents(v) and parents(w), and return p. In a cascade of CONS operations, we implement each CONS operation by searching in the set of parents of the node returned by the previous CONS operation.

We represent each set parents(v) by a binary search tree and perform searches and insertions on the tree using the Splay Algorithm[2]. A search operation is followed by a

[2]The Splay Algorithm is described in Chapter 3, Section 3.1.1.

splay on the last-visited node during the search. A new element is inserted into the tree as follows. If the inserted element is larger than the current maximum element, insert it as the right child of the maximum element; this requires maintaining a pointer to the rightmost node in the tree. Otherwise insert the element into the tree in the standard top-down manner and then splay at the element. These two types of insertions are called *passive* and *active*, respectively. We implement passive insertions more efficiently since they are more numerous than active insertions.

4.3 The Analysis

The following theorem summarizes the performance of the data structure for CONS operations.

Theorem 4.1 *The amortized cost of a cascade of f* CONS *operations equals $O(f + \log m_c)$, where m_c denotes the total number of cascades performed on S-expressions.*

The key idea behind the proof of this theorem is to bound the cost of operations on a parent set using a strong form of Sleator and Tarjan's Static Optimality Theorem [28]. We focus on the graph induced by the S-expression nodes, write the static optimality expressions for all these nodes, and bound the sum of the static optimality expressions over all the nodes, using the fact that S-expression nodes have at most two children (even though they might have unboundedly many parents).

We state the lemmas used in proving the theorem. The following lemma uses the notion of blocks in a binary tree introduced in Chapter 3, Section 3.2.1, and occurs implicitly in the work of Cole et al. [11].

Lemma 4.1 *Consider a binary search tree whose elements have been assigned arbitrary nonnegative weights. Suppose that the tree is partitioned into blocks so that each block has a positive weight (the* weight *of a block equals the total weight of all the elements in it). Let n denote the number of elements in the tree and let n_b denote the number of blocks. The cost of a sequence of m splays performed on the roots of the blocks equals $O(m + n + \sum_{j=1}^{m} \log(W/w_j) + \sum_{i=1}^{n_b} \log(W/\bar{w}_i))$, where W = the total weight of all the elements, w_j = the weight of the block of the jth accessed element, and $\bar{w}_i$ = the weight of the ith block of the tree.*

Proof. Assign potentials to the nodes of the tree as described by Cole et al. [11] in Section 2, "Global insertions", and analyze the splays using their analysis of global insertions[3]. Their analysis yields the following conclusions: the amortized cost of a splay on the root of a block with weight w equals $O(1+\log(W/w))$; the drop in potential over the entire sequence equals $O(\sum_{i=1}^{n_b}\log(W/\bar{w}_i)+n)$. The result follows. □

The following lemma bounds the cost of the sequence of operations performed on a single parent set and it is the key idea underlying the analysis.

Lemma 4.2 *Consider a sequence of insertions and searches performed on an (initially empty) binary search tree using splays. Let*
f_i = *the number of searches of element* i,
F = *the total number of searches,*
n_a = *the number of active insertions, and*
n = *the total number of insertions.*
The cost of this sequence equals $O(n+n_a\log n_a+F+\sum_{f_i\geq 1} f_i\log(F/f_i))$.

Proof. We modify the sequence by preinserting all the elements into the initial tree according to their order of arrival (without splaying). On this tree, we perform the searches and simulate the insertions. Active insertions are simulated by splaying at the corresponding elements and passive insertions are simply ignored. We obtain a sequence of splays corresponding to active insertions and searches (*active* splays and *hot* splays, respectively). It suffices to bound the cost of this sequence.

We bound the cost of this sequence by partitioning the tree into blocks and applying Lemma 4.1. Partition the tree into blocks as follows. The elements accessed by active and hot splays are, respectively, called *active* and *hot.* Every active or hot element forms a singleton block. Each nonempty interval of nodes between consecutive singleton blocks forms a *passive* block. Choose an element from each passive block and call it the *block representative.* Note that n_a = the number of active elements. The weight of element i is defined by:

$$\begin{cases} f_i & \text{if the element is hot} \\ F/(n_a+1) & \text{if the element is active but not hot} \\ 0 & \text{if the element is in a passive block} \\ & \text{but not the representative} \end{cases}$$

[3] An account of this analysis can also be found in Cole [9], Section 4.

The representatives of n_a+1 of the passive blocks are each assigned a weight of $F/(n_a+1)$; the representatives of the remaining passive blocks are placed in one-to-one correspondence with the set of hot elements and assigned the weights of their mates. The total weight of the tree is at most $4F$. Applying Lemma 4.1, the cost of the sequence of splays equals

$$O(n_a + F + n + \sum_{f_i \geq 1} f_i \log(4F/f_i) + n_a \log(4(n_a+1)) + 2(\sum_{f_i \geq 1} \log(4F/f_i)) + (2n_a+1)\log(4(n_a+1))) = O(n + n_a \log n_a + F + \sum_{f_i \geq 1} f_i \log(F/f_i)).$$

□

Remark. The lemma is a strong form of Sleator and Tarjan's Static Optimality Theorem [28]. The term *static optimality* comes from the expression $\sum_i f_i \log(F/f_i)$ which gives the weighted path length of the optimal static binary tree whose leaves have weights $f_1, f_2, \ldots, f_n$. Their theorem applies only to sequences of searches in which all the elements of the tree are accessed at least once. The use of Cole et al.'s sharper analysis [11] yielded our stronger lemma.

The following graph inequality will help us to bound the sum of the static optimality expressions over the nodes of the S-expression graph, using the fact that the nodes of this graph have constant-bounded indegrees.

Lemma 4.3 *Consider a digraph $G = (V, E)$ and consider a collection of walks in G. Let*

F_e = *the number of traversals of edge e in the walks,*
F_v = $\sum_{(v,w)\in E} F_{(v,w)}$, *for any vertex v,*
W_v = *the number of walks originating at vertex v, and*
id_v = *indegree$(v)+1$.*

Then,

$$\sum_{(v,w)\in E} F_{(v,w)} \log(F_v/F_{(v,w)}) \leq \sum_{v\in V} F_v \log id_v + \sum_{F_v \geq 1} W_v \log F_v.$$

Proof. Let $\bar{F}_{(v,w)} = F_{(v,w)} -$ #walks with (v,w) as the last edge.

$$\sum_{(v,w)\in E} F_{(v,w)} \log(F_v/F_{(v,w)}) = \sum_{F_v \geq 1} F_v \log F_v + \sum_{(v,w)\in E} F_{(v,w)} \log(1/F_{(v,w)})$$

$$\begin{aligned}
&\leq \sum_{F_v \geq 1} W_v \log F_v + \sum_{(x,v)\in E} \bar{F}_{(x,v)} \log F_v + \sum_{(v,w)\in E} \bar{F}_{(v,w)} \log(1/\bar{F}_{(v,w)}) \\
&\quad (x \log(1/x) \text{ is decreasing in } [1/e, \infty]) \\
&= \sum_{F_v \geq 1} W_v \log F_v + \sum_{F_w \geq 1} \sum_{(v,w)\in E} \bar{F}_{(v,w)} \log(F_w/\bar{F}_{(v,w)}) \\
&\leq \sum_{F_v \geq 1} W_v \log F_v + \sum_{w\in V} F_w \log id_w \quad \text{(entropy inequality).} \quad \square
\end{aligned}$$

We are ready to prove the theorem.

Proof of Theorem 4.1. Consider a sequence of m_c cascades of CONS operations, comprising F CONS operations totally. The cost of a cascade of f CONS operations equals $O(f)$ plus the cost of operations performed on parent sets. During any cascade of CONS operations, there are at most two active insertions into parent sets. These insertions are performed when the first node is created by the cascade; all subsequent insertions are passive. Hence, out of at most $2F$ insertions into parent sets totally performed during the sequence of cascades, at most $2m_c$ insertions are active insertions. Applying Lemma 4.2 to the sequence of insertions and searches performed on each parent set and summing the costs over all parent sets, we see that the total cost of parent set operations equals

$$O(F + m_c \log m_c + \sum_{\text{nodes } v} \sum_{\substack{i \in \text{ parents}(v) \\ \wedge \ f_i \geq 1}} f_i \log(F(v)/f_i)),$$

where $F(v)$ denotes the total number of searches performed on parents(v) and f_i denotes the number of searches of element i among these. The double summation bounds the total cost of all searches performed on the parent sets. This summation can be bounded using Lemma 4.3. The collection of nodes at the end of the sequence of cascades induces a directed graph whose vertices are the S-expression nodes and whose edges go from nodes to their parents. The indegree of each vertex in this graph is at most 2. For each edge (v, w), define $F_{(v,w)}$ = the number of searches of node w performed on parents(v). Delete all edges e such that $F_e = 0$. Applying Lemma 4.3 to the resulting graph, we see that the summation is bounded by $F \log 3 + m_c \log m_c$. It follows that the cost of the sequence of cascades equals $O(F + m_c \log m_c)$. The theorem follows. $\square$

4.4 Directions for Further Work

The following open problems arise naturally in connection with this work:

1. Is there a data structure for implementing CONS operations in constant amortized time and constant amortized space, in general?

2. Prove (or disprove) that the problem of maintaining sets under the complete repertoire of set operations has no *efficient* solution. An *efficient* solution is one that implements all set operations in time polylogarithmic in the number of update operations.

3. The *Sequence Equality-testing Problem* [31] is to maintain a collection of sequences from a finite, ordered universe under equality-tests and under creations of new sequences through insertions and deletions of elements. There exists a data structure that performs equality-tests of sequences in constant time and updates sequences in about $\sqrt{n}$ time/space, where n denotes the length of the updated sequence. The problem can be solved in $O(\log m)$ time/space per update operation if either sequences are repetition-free or randomization and a small error are permitted; m denotes the number of update operations. The existence of a deterministic (or even an error-free randomized) data structure that updates sequences in polylogarithmic time/space, in general, remains open.

Bibliography

[1] A.V.Aho, J.E.Hopcroft, and J.D.Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading, Mass., 1974.

[2] A.V.Aho and D.Lee. Storing a dynamic sparse table. In Proc. 27th IEEE FOCS, 1986, 55-60.

[3] M.Ajtai. A lower bound for finding predecessors in Yao's cell probe model. Combinatorica 8, 3, 1988, 235-247.

[4] M.Ajtai, M.Fredman, and J.Komlos. Hash functions for priority queues. Information and Control 63, 1984, 217-225.

[5] J.Allen. Anatomy of LISP. McGraw Hill Publishing Co., 1978.

[6] J.Bentley and J.Saxe. Decomposable searching problems 1: Static-to-dynamic transformations. J. Algorithms 1, 1980, 301-358.

[7] N.Blum. On the single operation worst-case time complexity of the disjoint set union problem. SIAM J. Computing 15, 1986, 1021-1024.

[8] J.L.Carter and M.N.Wegman. Universal classes of hash functions. J. Comp. Sys. Sci., 18, 1979, 143-154.

[9] R.Cole. On the dynamic finger conjecture for splay trees. In Proc. 22nd ACM STOC, 1990, 8-17.

[10] R.Cole. On the dynamic finger conjecture for splay trees 2: Finger searching. Courant Institute Technical Report No. 472, 1989.

[11] R.Cole, B.Mishra, J.Schmidt, and A.Siegel. On the dynamic finger conjecture for splay trees 1: Splay-sorting ($\log n$)-block sequences. Courant Institute Technical Report No. 471, 1989.

[12] K.Culik II and D.Wood. A note on some tree similarity measures. Info. Process. Lett. 15, 1982, 39-42.

[13] M.Dietzfelbinger, A.Karlin, K.Mehlhorn, F.Meyer auf der Heide, H.Rohnert, and R.E.Tarjan. Dynamic perfect hashing: Upper and lower bounds. In Proc. 29th IEEE FOCS, 1988, 524-531.

[14] M.L.Fredman, J.Komlos, and E.Szemeredi. Storing a sparse table with $O(1)$ worst case access time. J. ACM 31, 3, 1984, 538-544.

[15] M.L.Fredman and M.E.Saks. The cell probe complexity of dynamic data structures. In Proc. 21st ACM STOC, 1989, 345-354.

[16] M.L.Fredman, R.Sedgewick, D.D.Sleator, and R.E.Tarjan. The pairing heap: a new form of self-adjusting heap. Algorithmica 1, 1986, 111-129.

[17] M.L.Fredman and D.E.Willard. Blasting through the information theoretic barrier with fusion trees. In Proc. 22nd STOC, 1990, 1-7.

[18] I.Galperin and R.L.Rivest. Scapegoat trees. Manuscript, December 1990.

[19] S.Hart and M.Sharir. Nonlinearity of Davenport-Schinzel sequences and of generalized path compression schemes. Combinatorica 6, 2, 1986, 151-177.

[20] W.Hoeffding. Probability inequalities for sums of bounded random variables. Journal of the American Statistical Association 58, 1963, 13-30.

[21] D.E.Knuth. The Art of Computer Programming 3: Sorting and Searching. Addison-Wesley, Reading, Mass., 1973.

[22] J.M.Lucas. Arbitrary splitting in splay trees. Rutgers University Tech. Rept. No. 234, June 1988.

[23] K.Mehlhorn. Data Structures and Algorithms 1: Sorting and Searching. Springer-Verlag, 1984.

[24] K.Mehlhorn, S.Naher, and M.Rauch. On the complexity of a game related to the dictionary problem. In Proc. 30th IEEE FOCS, 1989, 546-548.

[25] W.Paul and J.Simon. Decision trees and random access machines. Symposium uber logik and algorithmik, Zurich 1980; also in Mehlhorn [23, 85-97].

[26] W.Pugh. INCREMENTAL COMPUTATION AND THE INCREMENTAL EVALUATION OF FUNCTIONAL PROGRAMMING. Ph.D. Thesis, Cornell University, 1988.

[27] W.Pugh and T.Teitelbaum. Incremental computation via function caching. In Proc. 16th ACM POPL, 1989, 315-328.

[28] D.D.Sleator and R.E.Tarjan. Self-adjusting binary search trees. J. ACM, 32, 1985, 652-686.

[29] D.D.Sleator, R.E.Tarjan, and W.P.Thurston. Rotation distance, triangulations, and hyperbolic geometry. J. Amer. Math. Soc. 1, 3, 1988, 647-681.

[30] R.Sundar. Twists, turns, cascades, deque conjecture, and scanning theorem. In Proc. 30th IEEE FOCS, 1989, 555-559; On the deque conjecture for the splay algorithm. Combinatorica, to appear.

[31] R.Sundar and R.E.Tarjan. Unique binary search tree representations and equality-testing of sets and sequences. In Proc. 22nd ACM STOC, 1990, 18-25.

[32] R.E.Tarjan. DATA STRUCTURES AND NETWORK ALGORITHMS. Society for Industrial and Applied Mathematics, Philadelphia, Pa., 1983.

[33] R.E.Tarjan. Sequential access in splay trees takes linear time. Combinatorica 5, 1985, 367-378.

[34] R.E.Tarjan. Amortized computational complexity. SIAM J. Appl. Discrete Meth. 6, 1985, 306-318.

[35] M.N.Wegman and J.L.Carter. New hash functions and their use in authentication and set equality. J. Comp. Sys. Sci. 22, 1981, 265-279.

[36] R.Wilber. Lower bounds for accessing binary search trees with rotations. SIAM J. Computing 18, 1989, 56-67.

[37] D.E.Willard. New trie data structures which support very fast search operations. J. Comp. Sys. Sci., 28, 1984, 379-394.

[38] A.C.Yao. Should tables be sorted? J. ACM 28, 1981, 615-628.

[39] A.C.Yao. On the complexity of maintaining partial sums. SIAM J. Comput. 14, 2, 1985, 277-288.

[40] D.Yellin. Representing sets with constant time equality testing. IBM Tech. Rept., April 1990; In Proc. First Annual ACM-SIAM Symposium on Discrete Algorithms, 1990, 64-73.