UNIVERSITY OF CALIFORNIA,
IRVINE

An Assortment of Sorts: Three Modern Variations on the Classic Sorting Problem

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

William Eric Devanny

2017

DEDICATION

To my parents, Caroline and Earl, and to Mikaela for their support and willingness to listen to my bad jokes.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

I would like first and foremost like to thank Professor David Eppstein and Professor Michael Goodrich for advising me during my graduate. I would also like to thank Professor Sandy Irani for serving along with them on my dissertation and advancement committees. I would like to thank Professor Michael Dillencourt and Professor Long Chen for also serving on my advancement committee. All of the faculty in the UCI Center for Algorithms and Theory of Computation have my gratitude for their mentorship during my studies.

The UCI Theory graduate students, past, present, and future, also have my appreciation for their fellowship in our academic pursuits.

I would like to thank all of my coauthors: Michael Bannister, Ziv Bar-Joseph, Juan Besa, Jack Cheng, Vida Dujomvić, David Eppstein, Jason Ernst, Anthony Gitter, Michael Goodrich, Kris Jetviroj, Tsvi Kopelowitz, Marcel Schulz, Joe Simons, Bálint Tillman, Lowell Trott, David Wood, and Shan Zhong.

I am grateful to the University of California, Irvine and the Donald Bren School of Information and Computer Sciences for their funding support that enabled my research.

I would like to thank the NSF who supported me with an NSF Graduate Research Fellowship under grant DGE-1321846. The NSF also supported work presented in this dissertation under grants 1228639, 1526631, 1217322, 1618301, and 1616248. I would also like to thank the 3M Corporation for a gift that supported work in this dissertation. I would like to thank David Eppstein and Ian Munro for several helpful discussions concerning the topics of Chapter 3. I would like to thank DARPA for supporting work reported on in Chapter 4 under agreement no. AFRL FA8750-15-2-0092. The views expressed in this dissertation are those of myself and my coauthors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

I am very grateful ACM has granted me permission to include their copyrighted material in Chapter 3. At the beginning of Chapter 3, I acknowledge their permission, and I provide a full citation to the publication where it appeared.

CURRICULUM VITAE

William Eric Devanny

EDUCATION

Doctor of Philosophy in Computer Science **2017**
University of California, Irvine *Irvine, California*

Bachelor of Science in Computer Science **2012**
Carnegie Mellon University *Pittsburgh, Pennsylvania*

TEACHING EXPERIENCE

Lecturer **2017**
Pomona College *Claremont, California*

Teaching Assistant **2013–2015**
University of California, Irvine *Irvine, California*

Teaching Assistant **2011**
Carnegie Mellon University *Pittsburgh, Pennsylvania*

PUBLICATIONS

The online house numbering problem: min-max online list labeling **2017**
William E. Devanny, Jeremy Fineman, Michael T. Goodrich, Tsvi Kopelowitz
European Symposium on Algorithms

Parallel equivalence class sorting: algorithms, lower bounds, and distribution-based analysis **2016**
William E. Devanny, Michael T. Goodrich, Kris Jetviroj
Symposium on Parallelism in Algorithms and Architectures

The computational hardness of dK-series **2016**
Bálint Tillman, William E. Devanny, David Eppstein
NetSci

Scheduling autonomous vehicle platoons through an unregulated intersection **2016**
Juan José Besa Vial, William E. Devanny, David Eppstein, Michael T. Goodrich
Workshop on Algorithmic Approaches for Transportation Modelling, Optimization and Systems

Track layout is hard **2016**
Michael J. Bannister, William E. Devanny, Vida Dujmović, David Eppstein, David R. Wood
International Symposium on Graph Drawing

Windows into geometric events: Data structures for time-windowed querying of temporal point sets **2014**
Michael J. Bannister, William E. Devanny, Michael T. Goodrich, Joseph A. Simons, Lowell Trott
Canadian Conference on Computational Geometry

ERGMs are hard **2014**
Michael J. Bannister, William E. Devanny, David Eppstein
ArXiv

Small superpatterns for dominance drawing **2014**
Michael J. Bannister, William E. Devanny, David Eppstein
Analytic Algorithmics and Combinatorics

The Galois complexity of graph drawing: Why numerical solutions are ubiquitous for force-directed, spectral, and circle packing drawings **2014**
Michael J. Bannister, William E. Devanny, David Eppstein, Michael T. Goodrich
International Symposium on Graph Drawing

Superpatterns and universal point sets **2013**
Michael J. Bannister, Zhanpeng Chang, William E. Devanny, David Eppstein
International Symposium on Graph Drawing

DREM 2.0: Improved reconstruction of dynamic regulatory networks from time-series expression data **2012**
Marcel H. Schulz, William E. Devanny, Anthony Gitter, Shan Zhong, Jason Ernst, Ziv Bar-Joseph
BMC Systems Biology

ABSTRACT OF THE DISSERTATION

An Assortment of Sorts: Three Modern Variations on the Classic Sorting Problem

By

William Eric Devanny

Doctor of Philosophy in Computer Science

University of California, Irvine, 2017

Professor David Eppstein, Chair

Sorting is one of the most well studied algorithmic problems in Computer Science. It is a fundamental building block in many other algorithms. In this dissertation, we consider several variants of the classical sorting problem all motivated by modern challenges or technologies. We present algorithms to solve these problem variants and provide lower bounds when possible.

The online list labelling problem attempts to maintain integer labels for a dynamic ordered list. As new elements are inserted, old elements may need to be relabeled to make room in the label space. Previous work has looked at minimizing the total number of relabels that need to be performed. However we analyze the version of the problem where the goal is to minimize the maximum number of times any one element is relabeled. We call this the *online house numbering problem.* This problem is motivated by the modern solid-state memories which have a limited write life. We provide two solutions to the house numbering problem: one that comes within a logarithmic factor of the optimal label space size with optimal maximum relabelings and one that has optimal label space size, but is a logarithmic factor off of the optimal maximum relabelings.

Sorting can also mean to split a set of elements into groups of similar elements. Cryptographic handshakes, where two parties securely identify if they belong to a privileged group, motivate studying this form of sorting that we call *equivalence class sorting*. Instead of sorting with a $<$ operator, our goal is to use an $\equiv$ operator to group a set of elements into their equivalence classes. We prove tight lower bounds that match the runtime of previously known algorithms as well as provide algorithms for performing equivalence class sorting in several models of parallel computation.

Classical sorting algorithms output the sorted order for a given input list. When the data is continually changing or “evolving”, the output of a classical algorithm cannot be guaranteed to be accurate. So we consider a new model for algorithms called the evolving data model. In this model, every time a comparison is performed, two elements that are adjacent in the underlying order are swapped. No algorithm can ever compute the exact correct order of the elements in such an evolving list. Instead the goal is to, over time, converge to be as close to the correct order as possible. We show that simply repeatedly running insertion sort achieves the best possible $O(n)$ inversions relative to the underlying order with exponentially high probability.

Chapter 1

Introduction

Sorting predates the modern computer. In fact, an automated sorting machine was used to count United States census data in 1880 [59]. Some of the earliest computer programmers worked on writing sorting algorithms [64]. Since then sorting has become a foundational topic of study. Knuth [59] makes the claim that "*every* important aspect of programming arises somewhere in the context of sorting or searching!"

Formally given an input list of n elements l, the sorting problem is to output the list containing the same set of elements permuted into an order respecting some $<$ operator i.e., if $i < j$, then in the output $l[i] < l[j]$. Ford and Johnson [41] introduced the comparison model in which algorithms are permitted to compare list elements using the $<$ operator and are assessed based on the worst case or expected number of these comparisons that they perform. The elements of the input list are often called *keys* and are contained in some larger set of possible keys called a *universe*. Each key may carry some auxiliary information. For example in a contact list, entries may be sorted by last name while also being associated with emails, phone numbers, and other personal information. Oftentimes the keys to be sorted are integers where the $<$ operator is the natural numeric order or strings where the $<$ operator is lexicographic order such as in a dictionary.

As every academic who has had to return a stack of graded papers to students knows, sorting is a useful subroutine to speed up future processing. In an unsorted list, looking up an element may require $O(n)$ time, but after sorting a list, one can use binary search to find an element in $O(\log n)$ time [29]. Similarly, finding the element of a given rank in an unsorted list may require $O(n)$ steps, but in a sorted list it only takes $O(1)$ time [29]. Aside from these two relatively simple applications, sorting algorithms are used as black boxes in algorithms in computational geometry [46], graph theory [63], and nearly every other computer science subfield.

Over the course of an undergraduate computer science education, students learn several classical sorting algorithms including bubble sort, selection sort, insertion sort, merge sort, quicksort, and heapsort [29,44]. Although relatively simple, these sorting algorithms illustrate important algorithmic design and analysis techniques such as loop invariants, recursion, worst-case analysis, and the usage of basic data structures. The principles in these algorithms will also be applied frequently in more advanced areas of study. In fact, Chapter 5.5 of Knuth [59] describes sophisticated applications which apply the ideas used in each of these classical sorting algorithms. Even bubble sort finds an application in two-tape sorting.

There is a well known comparison based sorting lower bound that $\Omega(n \log n)$ comparisons are required by any algorithm in the worst case. The classical sorting algorithms can be divided into those that achieve the optimal $\Theta(n \log n)$ run time (quicksort, merge sort, and heapsort) and those that do not (bubble sort, selection sort, and insertion sort). However interestingly in some cases "slower" algorithms may perform much better. When there are $o(n \log n)$ inversions in the input list, insertion sort has an asymptotically better runtime than the $\Theta(n \log n)$ algorithms quicksort, merge sort, and heapsort.

Because the comparison based model ignores the availability of certain powerful computing techniques as well as the difficulties that arise when sorting on real hardware, many variants of the classical sorting problem have been studied.

External memory sorting When the data to be sorted is too large to fit into main memory on a computer, the classical sorting algorithms can become infeasibly slow. Instead new techniques are required to minimize the amount of reading from and writing to the external memory. Agarwal [1] implemented a merge sort based algorithm to compare how the run time was divided up between reading, writing, and non-i/o operations.

Integer Sorting Instead of exclusively using the $<$ operator to interact with the input elements, using other operations on the elements may be advantageous in certain contexts. For example, when the list elements are integers in some polynomial range, radix sort uses bit-level operations to achieve $O(n\frac{\log U}{\log n})$ operations, where U is the size of the universe, breaking the $\Omega(n \log n)$ operation comparison sorting lower bound when $U = poly(n)$. Fusion trees were used by Fredman and Willard [42] to sort integers in $O(n \log n / \log \log n)$ time which is independent of U. Han and Thorup [48] improved this bound further and managed to use just $O(n\sqrt{\log n \log n})$ time.

Parallel sorting As computers have become more sophisticated the prospect of using multiple processors in parallel to solve problems has become viable. In this context, the goal is to sort a list of numbers using some number p of processors with, depending on the exact model, some form of shared access to memory. Shiloach and Vishkin [75] and Hirschberg [49] both describe parallel algorithms to sort in $O(k \log n)$ rounds using $n^{1+1/k}$ processors which all have complete access to the shared memory. Cole [26] gives a parallel merge sort using only n processors and $O(\log n)$ time. The AKS sorting network [2] is also a parallel sorting algorithm using n processors that also takes $O(\log n)$ time.

Online list labeling The online list labeling problem [15, 32, 60] can be thought of as a form of online sorting. Elements are assigned integer labels respecting the $<$ operator. When new elements arrive and are placed into the sorted order, they must receive a label between the labels of their two neighbors. If there is no such label available, then the labels for elements must be reallocated. Using a label space of $[1, 2^n]$ allows n elements to be inserted without a single relabel being required. So online list labeling data structures optimize two competing goals: minimize the number of relabelings and minimize the size of the label space used. Several papers give solutions when the label space is constrained to be of size $O(n)$ [15, 18, 80, 81]. When the label space is polynomial in size, $O(\log n)$ relabels suffice [60] and are required [23].

1.1 Results

In this dissertation, we consider three variants of the sorting problem and will provide algorithms and lower bounds.

House numbering problem In Chapter 2, we analyze the house numbering problem which is a modification of online list labeling. Some modern memories such as flash memory only support a limited number of writes [19, 71, 82]. Using naive online list labeling solutions with these memories could lead to a particular memory cell being overwritten repeatedly. Supposing the elements are stored in flash memory, we consider the slightly different problem of minimizing the maximum number of times any single element is relabeled. The conventional online list labeling data structures cannot be easily adapted to this goal.

We give two data structures for the house numbering problem. Our first guarantees no element will be relabeled more than $O(\log^2 n)$ times using an $O(\log n)$-bit label space and our second uses at most $O(\log n)$ relabels for any element with an $O(\log^2 n)$-bit label space.

Thus each solution is optimal in one parameter while being within a logarithmic factor of optimal in the other parameter.

The first of our two data structures uses a tree on the label space and moves elements throughout the tree always maintaining the correct relative order of elements. As an element undergoes relabels it is moved higher up in the tree and is thus less likely to be moved in the future. Interestingly there is no immediately obvious reason why the tree should not grow to an unbounded height. Proving that the elements stay somewhat low in the tree and so only use a small number of labels requires a particularly sophisticated potential argument.

Equivalence class sorting Jayapaul *et al.* [56] considered a slightly unconventional notion of sorting where instead of using a $<$ operator, they use an $\equiv$ operator. Obviously the goal cannot be to place the elements in order. So instead we attempt to group elements into their equivalence classes and our goal is to minimize the number of equivalence tests we need to perform. We call this the equivalence class sorting problem. This problem naturally arises when several individuals wish to use cryptographic handshakes to identify themselves as belonging to certain groups. Jayapaul *et al.* [56] showed that while in general beating $\Omega(n^2)$ equivalence tests is impossible, it is possible to equivalence class sort using $O(n^2/\ell)$ equivalence tests, where ℓ is the size of the smallest equivalence class. They also showed that even when ℓ is known, one cannot sort using fewer than $\Omega(n^2/\ell^2)$ equivalence tests.

In Chapter 3, we prove lower bounds for the equivalence class sorting problem and provide some parallel algorithms. We prove that any algorithm must use at least $\Omega(n^2/\ell)$ equivalence tests implying the algorithm of Jayapaul *et al.* [56] is optimal. When k the number of equivalence classes is constant with respect to n, we show that we can sort the input using just a constant number of parallel rounds of equivalence tests. However when k is non-constant, we are only able to sort in $O(k + \log\log n)$ or $O(k \log n)$ tests depending on the power of our parallel computation model. We also provide some experimental results on how long it takes to sort when the elements are drawn from a few common discrete distributions.

Sorting evolving data Motivated by the idea that underlying data might change during the execution of an algorithm, a new evolving data framework was introduced by Anagnostopoulos *et al.* [7]. Imagine that while processing a particularly long list of web pages several pages are deleted and several new pages are created. The output of such a process could be highly inaccurate. Instead if we use an algorithm that repeatedly resamples the input, we can provide a guarantee on the accuracy of our output. While generally sorting can be performed very quickly (so the data might not have changed very much), the foundational nature of sorting means it is interesting to investigate how to sort evolving data.

In the evolving data model for sorting, the order underlying the input elements slowly changes over time. Whenever the algorithm performs a comparison, an adjacent pair of elements in the underlying order are swapped. Algorithms are judged based on the number of inversions between the list they maintain and the underlying order. Anagnostopoulos *et al.* [7] were able to show that in this model no algorithm can achieve fewer than $\Omega(n)$ inversions with high probability and gave a quicksort based algorithm that achieves $O(n \log \log n)$. In Chapter 4, we show that simply repeating insertion sort forever guarantees the optimal $O(n)$ inversions with exponentially high probability.

If the random swaps in the underlying order and the work of insertion sort can be considered separately it is relatively easy to see that this algorithm works well. However understanding the interplay between the algorithm and the running swaps requires some care. We ultimately show that after t steps of insertion sort, the number of inversions insertion cannot fix is bounded by the sum of squares of t balls thrown uniformly at random into n bins. So if there are a linear number of inversions, then with high probability a linear number of steps will be performed and most of the steps will fix inversions.

Chapter 2

The Online House Numbering Problem: Min-Max Online List Labeling

2.1 Introduction

In this chapter we study a new version of the fundamental *online monotonic list labeling problem* [15, 32, 60] (OMLL), where the goal is to maintain labels for a dynamic ordered list of at most n elements that, due to monotonicity requirements of appropriate applications, must have (integer) labels that strictly increase in the direction of the ordering. When a new element is inserted into the list, either between two existing elements or at an endpoint of the list, we must assign a label to the new element that is consistent with the order of the list. To avoid labels becoming too long, algorithms for list-labeling problems relabel elements from time to time to maintain the ordering using relatively few bits for the labels. There are several common variants of OMLL that differ in the number of bits allowed for each label.

For example, the special case of allowing $\log n + O(1)$ bits[1] is known as the file-maintenance problem [15,18,80,81] as the labels can be viewed as corresponding to addresses in a size-$O(n)$ array.

Solutions for OMLL are used as foundational building blocks in several areas of computer science, ranging from cache-oblivious data structures [16,17,22] to distributed computing [36], and they play a central role in deamortization [21,32,34]. As an illustration of the data structure's central role in the field, it is used as a black box in order-maintenance data structures [32,60], which themselves are used as black boxes throughout computer science (for examples see [5,6,27,33,37,39,61]).

The focus of previous work solving the OMLL problem in the RAM model has been on minimizing the worst-case or amortized number of relabels per update. For example, when using $O(\log n)$ bits per label the worst-case number of relabels per update is known to be $O(\log n)$ [60], which is tight [23]. A particular element, however, can be relabeled as many as $\Omega(n)$ times during a sequence of n insertions using existing algorithms. This chapter considers the goal of minimizing the maximum number of times an element in the list is relabeled, while using only a small number of bits per label. We refer to this version of OMLL as the *online house numbering problem*, since it captures the challenges that take place when maintaining a strictly increasing numbering for a collection of houses representing their order along a road. When a new house is built between any two existing houses (or at either end of the row of houses), this new house needs to be assigned a house number. If no such integer house number is available, however, then other houses need to be renumbered (or relabeled) to make room for a number for the new house. Formally stated, the online house numbering

[1]Unless another base is indicated, all logarithms in this thesis are base 2.

problem is to maintain a labelling of an initially empty ordered list subject to n operations of the form, $\mathsf{insert}(x, a)$: insert x immediately after a in the ordered list. Remarkably, existing solutions for list labeling problems do not seem to lead to efficient solutions for the online house numbering problem.

Solutions to the online house numbering problem address label-update complexity, which is motivated from use of solid-state memories, like flash memory, that have an upper bound on the number of erasures that can occur for any memory cell [19, 71, 82]. For example, consider a database stored in flash memory with an ordered set of large records, where each record maintains a label respecting the order. Due to the use of memory with an erasure limit, the number of times that the label is changed must be minimized, since each relabeling entails rewriting that area in memory. A typical assumption in models for solid-state memories is that the algorithm or data structures also have access to a sublinear amount of additional *scratch space* for computational purposes (see Ben-Aroya and Toledo [14]), which is exempt from the erasure limits. In the context of our online house numbering, this would mean that each element in the data structure has a fixed record containing, e.g., the label and any other auxiliary information that is updated whenever a label changes (for our solution, we also store a counter as part of the record). Any additional components of the data structure must be restricted to the $o(n)$ scratch space.

There are two competing objectives that we consider in designing solutions for the online house numbering problem. The first objective is to minimize, over all elements in the list, the maximum number of times that the label of the element changes throughout the n insertions. Notice that with large labels, a trivial solution in which no relabels are needed is obtainable by assigning x the average of a and b, where b is the element succeeding x. This trivial solution requires $\Omega(n)$ bits per label, and so if each word of memory contains $\Theta(\log n)$ bits (which is a standard assumption), each label requires $\Omega(n/\log n)$ words. A large number of words directly impacts the efficiency of establishing the order of two elements, since comparing their labels entails scanning that many words. Thus, the second objective is to minimize the number of bits used in labels.

Since we are interested in minimizing two competing objectives, we express the complexities of our data structures using a pair of functions. A data structure supporting n insertions with $g(n)$ maximum relabels and using $h(n)$ bits per label is said to have complexity of $\langle g(n), h(n)\rangle$. Notice that $h(n) \geq \lceil \log n \rceil$ since n elements must be labeled. If one is interested in $h(n) = O(\log n)$ (constant number of words per label), then the OMLL lower bounds of [23] imply that $g(n) = \Omega(\log n)$. Thus, if there existed a solution for online house numbering with complexity $\langle O(\log n), O(\log n)\rangle$, it would be asymptotically optimal.

2.1.1 Our Results

In this chapter we describe two data structures that are close to the target bound of $\langle O(\log n), O(\log n)\rangle$, but each solution introduces an extra logarithmic factor in one of the functions. In a third solution, we investigate the dependence on the leading constant of $h(n)$ and provide a solution with complexity $\langle O(n^\epsilon), \log n + O(1/\epsilon)\rangle$. Our solutions, which can be adapted to work with $o(n)$ scratch space, establish the following results.

Theorem 2.1. *There exists a house numbering data structure with complexity* $\langle O(\log^2 n), O(\log n)\rangle$.

Theorem 2.2. *There exists a house numbering data structure with complexity* $\langle O(\log n), O(\log^2 n)\rangle$.

Theorem 2.3. *For any positive constant ϵ, there exists a house numbering data structure with complexity* $\langle O(n^\epsilon), \log n + O(1/\epsilon)\rangle$.

Proofs of Theorem 2.1 and Theorem 2.3 appear in Section 2.3. Theorem 2.2 is proved in Section 2.5. Our solution complexities exhibit an interesting feature: the online house numbering problem seems to exhibit a different tradeoff from OMLL, depending more strongly on the label lengths.

Overview of Challenges and Techniques. The main idea of our approach is that once a particular element has been relabeled many times, structural restrictions assure that this element will not be relabeled much in the future. To achieve this goal, we employ a tree-like structure similar to an (a, b)-tree (or B-tree) that stores at most $O(\gamma)$ elements in nodes of the tree, where $\gamma \geq 2$ is a parameter controlling the tradeoff between the two objectives. (For the purpose of this overview it is helpful to assume $\gamma = 2$.) Roughly speaking, the inorder traversal of this tree corresponds to the order of elements in the list. The elements in a node each have a local label, which is local to that node. The global label assigned to an element corresponds to the concatenation of the local labels on the path from the root to the element (with 0s padded at the end if the element is in an inner node). We require the local labels of elements to respect the order of elements in each node, thereby guaranteeing that the global labels respect the total order of the list. To simplify things when extending to non-constant γ, we employ (classic) file-maintenance data structures within each node for maintaining the local labels. Notice that changing the local label of an element also changes the global labels, which must be stored explicitly, of all elements in that element's subtree.

Our main strategy is to employ node splits to "promote" elements that have been relabeled too many times to higher levels in the tree. Promoting an element e that is currently in node u entails: splitting the elements of u around e into two new nodes, moving e into u's parent, and making the two new nodes children of u's parent. The intuition behind the promotions is that elements in higher nodes are less affected by insertions, and hence these elements need not be relabeled as often.

There are three reasons to promote an element: (1) if the element has been relabeled too many times since its last promotion, (2) if the node has too many elements, which would cause the performance of the file-maintenance black box to degrade, or (3) if the node becomes sufficiently imbalanced, according to a non-obvious weight-balance rule.

The key component of the analysis is ensuring that the height H of the data structure is well bounded, i.e., that elements are never promoted too far. The height H not only places an upper bound on the length of the labels, but in conjunction with the first trigger for element promotion also directly implies a bound on the number of times each element can be relabeled. We emphasize that the analysis bounding H leverages a potential argument in an atypical and non-obvious way, which we view as a surprising component of our data structure.

2.1.2 Related Prior Work

There is no prior work for the online house numbering problem, but it is closely related to the classic *file maintenance* and *online list labeling* problems for which several authors have shown how to achieve optimal polylogarithmic update times, in either worst-case or amortized senses (e.g., see [15, 18, 23, 32, 54, 60]).

Regarding algorithms in computational models that capture the challenges of solid-state memory, Ben-Aroya and Toledo [14] provide competitive analyses for several such algorithms, but they do not study OMLL problems as a specific topic of interest. See also the work of Irani *et al.* [53]. Subsequent work on efficiently implementing specific data structures and algorithms in such models includes methods for database algorithms [25] and hash tables [38].

2.2 Preliminaries

We prove Theorems 2.1, 2.2, and 2.3 using amortized analysis. Traditionally given a potential function Φ mapping data structure states to real numbers, the amortized cost of an operation x is $cost(x) + \Phi(D_{after}) - \Phi(D_{before})$ where $cost(x)$ is the actual cost of operation x, D_{before} is the state of the data structure before x, and D_{after} is the state of the data structure after x [77]. When summing the amortized costs of a sequence of operations, the $\Phi(D_{after})$ and

$\Phi(D_{before})$ terms telescope and cancel out. So the total cost of a sequence of N operations is at most the actual cost of the operations plus the potential in the final data structure state minus the starting potential. Our arguments do not use a potential function to analyze time costs. Instead our potential function is designed so that we can bound the height of a tree. First, we prove insertions will increase the potential of the tree by a small amount and promotions will have no effect on the potential. Second, we show that a tall tree requires a large amount of potential. Therefore for a given number of insertions, there is a maximal height to the tree.

In our house numbering data structures, we make use of instances of file maintenance data structures. The following lemma highlights the features that our algorithms leverage. Here, a file-maintenance data structure corresponds to an array, where placing an element in the ith slot in the array corresponds to assigning a label of i to the element.

Lemma 2.1. *For any capacity η, there exists a file maintenance data structure with the following properties:*

- *The data structure assigns to each element a slot in the range $[1, 4\eta]$. Slots are such that a is before b if and only if in the total order a's slot is before b's slot.*
- *If the data structure has at most η elements then it can be split into two data structures with each element being moved at most once.*
- *Starting from an empty data structure, or a data structure that is the output of a split, as long as the number of elements in the structure does not exceed η, the amortized number of elements that are moved to a new slot per insertion is $O(\log^2 \eta)$.*

Proof. A data structure by Itai et al. satisfies these conditions [54]. □

Using the notation of the statement of Lemma 2.1, a file maintenance data structure f is characterized by a *capacity* (i.e., η), a *slot range* (i.e., $[1, 4\eta]$), and an amortized *moving cost*

cost(η) (i.e., $O(\log^2 \eta)$), which are all static. The capacity specifies how many elements can be inserted while still maintaining the *cost*(η) bound. In addition, we define the *usage* of f, denoted *usage*(f), to be the number of elements currently inside f.

2.3 A Generic House Numbering Data Structure

We describe our data structure in terms of several variables, namely κ_1, κ_2, κ_3, π_1, and π_2. These will allow us to balance various overheads in the data structure and shall be fixed in the analysis. Additionally, our house numbering data structure is parameterized by a value $\gamma \geq 2$, which controls a tradeoff between the label lengths and the number of relabels performed. Setting γ to a constant attains the $\langle O(\log^2 n), O(\log n)\rangle$ data structure. When considering a data structure for flash memory, the data structure itself, in addition to the actual list, should reside in scratch space.

The tree. Our house-numbering structure is a perfect rooted $(4\kappa_1\gamma + 1)$-ary tree T. Each internal node u in the tree maintains an instance f_u of a $\kappa_1\gamma$-capacity file-maintenance data structure (à la Lemma 2.1) and the leaves of the tree are associated with space for a single element. The leaves store the actual elements e in the tree, but leaves may be empty. Each element e also maintains a *relabel counter* $c(e)$.

The internal nodes store (conceptual) copies of elements, which we call *representatives*, that have been promoted to a higher level in the tree. We refer to all the copies of a particular element e as *the representatives* e. Representatives are analogous to duplicate keys in internal nodes of a B$^+$ tree, with each non-empty node containing exactly one representative that has been promoted to the parent node.

Each file-maintenance data structure f_u assigns slots in the range $[1, 4\kappa_1\gamma]$ to the representatives in node u. Equivalently, the file-maintenance structure specifies how to store the representatives in a size-$\kappa_1\gamma$ array, starting from slot 1. We use the 0th slot in the array for a special *dummy representative* d_u^-, which corresponds to the only representative in node u that has also been promoted to the parent. (As such, d_u^- is a representative of the leftmost left element in u's subtree.) Note that since the slot storing the dummy representative is not part of the file-maintenance structure f_u, the dummy representative never moves from slot 0. The i-th slot in f_u corresponds to the i-th child of u in an inorder tree walk. For representative r in f_u let $s(r)$ denote the slot in f_u that is assigned to r.

Without yet worrying about precisely how elements are labelled, we state a property about how they must appear in T. Naturally, the order property constrains the way we label elements.

1. *Order Property:* *An inorder traversal of T encounters the elements in their house numbering order.*

We use T_u to denote the subtree rooted at node u. Let v be the parent of u in T. Since u is represented as a slot s in the slot range of f_v, we will abuse notation and sometimes denote T_u by T_s. A subtree is empty if it contains no elements.

The labels. The label for an element e, denoted $\ell_T(e)$, is based on the root-to-leaf path down to the leaf node containing e. In particular, labels are base-$(4\kappa_1\gamma + 1)$ numbers with a number of digits equal to the height of the tree. Consider the path $u_1, u_2, \ldots, u_{H_T+1}$ down to the leaf containing e, where u_1 is the root, u_{H_T+1} is the leaf containing e, and H_T is the height of T. For $1 \leq i \leq H_T$ let s_i denote the slot of u_{i+1} in f_{u_i}. Then e's label is the concatenation of digits $s_1, s_2, \ldots, s_{H_T}$. An example of determining the labels of elements is depicted in Figure 2.1. The label of each element uses $\lceil \log(4\kappa_1\gamma + 1) \rceil$ bits per level of T for a total of $H_T \lceil \log(4\kappa_1\gamma + 1) \rceil$ bits.

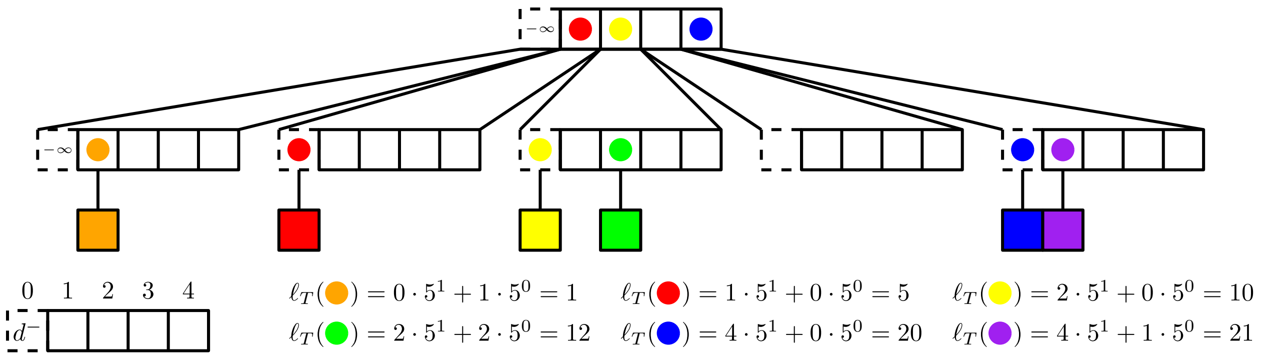

−∞
−∞
0 1 2 3 4
d^-
$\ell_T(●) = 0 \cdot 5^1 + 1 \cdot 5^0 = 1$
$\ell_T(●) = 1 \cdot 5^1 + 0 \cdot 5^0 = 5$
$\ell_T(●) = 2 \cdot 5^1 + 0 \cdot 5^0 = 10$
$\ell_T(●) = 2 \cdot 5^1 + 2 \cdot 5^0 = 12$
$\ell_T(●) = 4 \cdot 5^1 + 0 \cdot 5^0 = 20$
$\ell_T(●) = 4 \cdot 5^1 + 1 \cdot 5^0 = 21$

Figure 2.1: This tree illustrates how elements are labelled based on their representative's node's file maintenance labels and the node labels of their parents. Empty leaf nodes are omitted and we note that the root node is violating the Capacity Property.

Notice that by construction and the Order Property, the labels of elements respect the order of the elements in the house numbering.

Relabeling and subtrees. To maintain the Order Property, whenever a representative r is moved from slot s to slot s' in f_u, all of the elements and file-maintenance representatives in T_s are moved to the same exact location, but in $T_{s'}$. The following property will guarantee that this movement does not violate the Order Property.

2. *Representative Property:* *The representatives for an element e induce a path from the parent of the leaf containing e to the highest representative. Each representative of e except for the highest one is the dummy representative of its corresponding node.*

Following the Representative Property, we refer to the highest representative of an element e as the canonical representative of e, and denote this representative by $r(e)$.

2.3.1 Insertions

We now discuss the implementation of the $\mathsf{insert}(x, a)$ operation. Let u be the parent of the leaf node containing a, and assume for now that $usage(f_u) < \kappa_1\gamma$. A new representative r

of x is inserted into f_u immediately after the representative representing a in f_u (possibly causing elements in f_u to change slots). Because this insertion is into a file maintenance data structure, this insertion may cause some movement of other elements. Element x is placed into the leaf node corresponding to the slot assigned to r in f_u.

The insertion respects the Order Property, so x receives a valid label. The insertion causes some number of other representatives in f_u to be relabeled and also increases the usage of f_u. Eventually f_u will reach capacity. The capacity of the file maintenance instances needs to be respected and so when f_u reaches capacity we move around representatives in T to create room (thereby guaranteeing again that f_u is below capacity before the next insertion). This is captured by the following property.

3. *Capacity Property:* *For any internal node u in T, usage$(f_u) < \kappa_1\gamma$.*

In order to maintain the Capacity Property, the data structure employs the *promotion* of canonical representatives to higher nodes in T. The promotion procedure is detailed in Section 2.3.2.

Relabel counters. The relabel counter of an element is incremented whenever the label of the element is changed. To prevent any one element from being relabeled too many times, we enforce a bound on the relabel counter. Recall that the *cost*(η) function is defined in Section 2.2.

4. *Counter Property:* *For any element e, $c(e) < \kappa_2\, cost(\kappa_1\gamma)$.*

In order to enforce the Counter Property, whenever the counter of element e reaches its threshold, $r(e)$ is moved one level higher in the tree by a *promotion operation*, which we describe shortly, and sets $c(e) = 0$.

Notice that moving a representative to a new slot higher up in T will tend to relabel more elements compared to moving a representative to a new slot in a lower node in T. This presents a subtle challenge. Consider two representatives in f_u where u is relatively high up in T, such that one representative r has every file maintenance instance in $T_{s(r)}$ half full while the other representative r' has every file maintenance instance in $T_{s(r')}$ just below capacity. This implies that the ratio of the number of elements contained in $T_{s(r)}$ to the number of elements contained in $T_{s(r')}$ is exponentially small in the height of u. The consequence of this imbalance is that insertions of elements into the lighter subtree $T_{s(r)}$ can cause frequent promotions into f_u, each time causing r' to move to a new slot in f_u. When r' moves to a new slot, all of the elements in $T_{s(r')}$ must be relabeled. This imbalance creates some difficulties in keeping a tab on the complexities of the data structure. To overcome these difficulties, we enforce a requirement on the data structure to have the following property, which helps ensure a promotion does not relabel too many elements that are too high in the tree. This requirement restricts the total *weight* of elements in subtrees. For a representative r in f_u where u has height h in T, let $w(r) = \gamma^h$.

5. *Balance Property:* *For any node u, $\sum_{node\ v \in T_u} \sum_{canonical\ representative\ r \in f_v} w(r) < \kappa_3 \gamma^{height(u)}$.*

2.3.2 Promotions

For element e, the *promotion* of $r = r(e)$ from f_u to f_v, where v is a proper ancestor of u, is performed as follows. Let $\hat{r}$ be the representative in the slot in f_v that contains e in its subtree.

1. Insert r', which is a new representative of e, into f_v immediately after $\hat{r}$ in the order f_v is maintaining (this may cause some elements in f_u to change their slots).

2. Move any element in $T_{s(\hat{r})}$ that is after e (inclusive) and its representatives in $T_{s(\hat{r})}$ into identical locations in the subtree of $T_{s(r')}$.

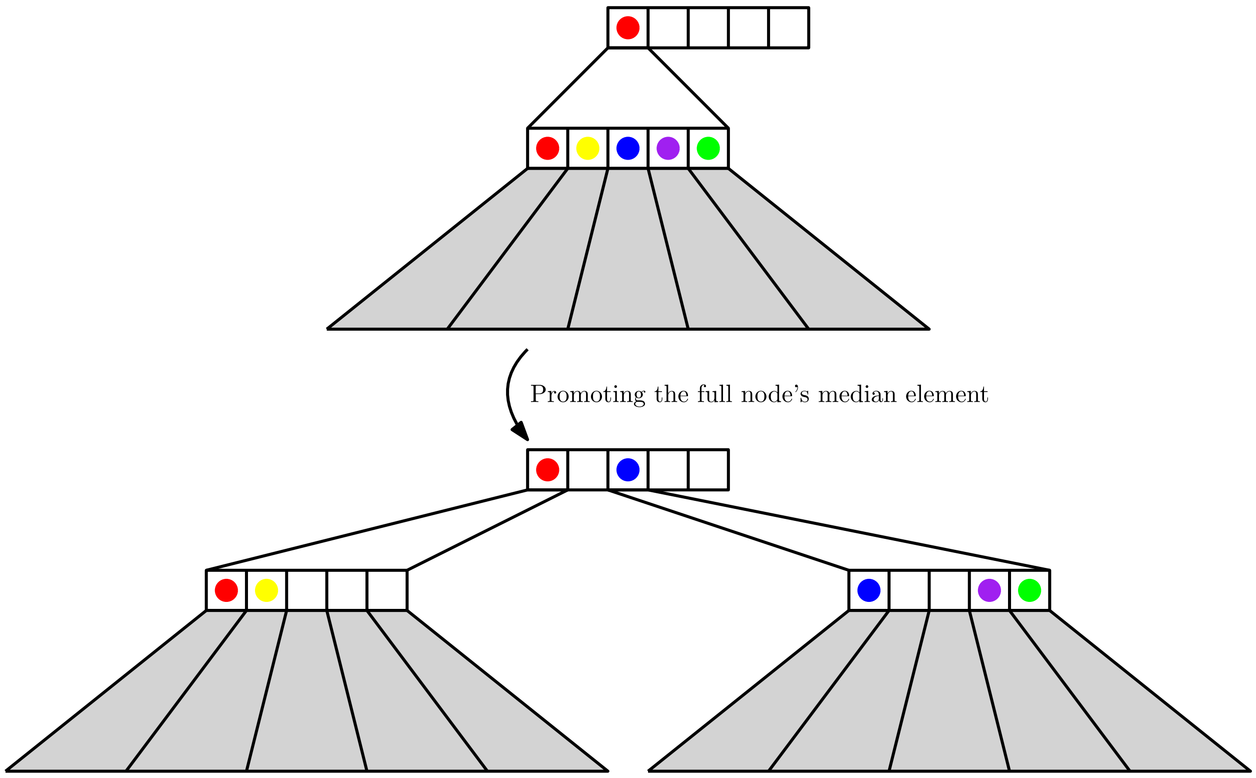


Figure 2.2: The blue median representative of the full file maintenance data structure is promoted dividing the subtree into two parts.

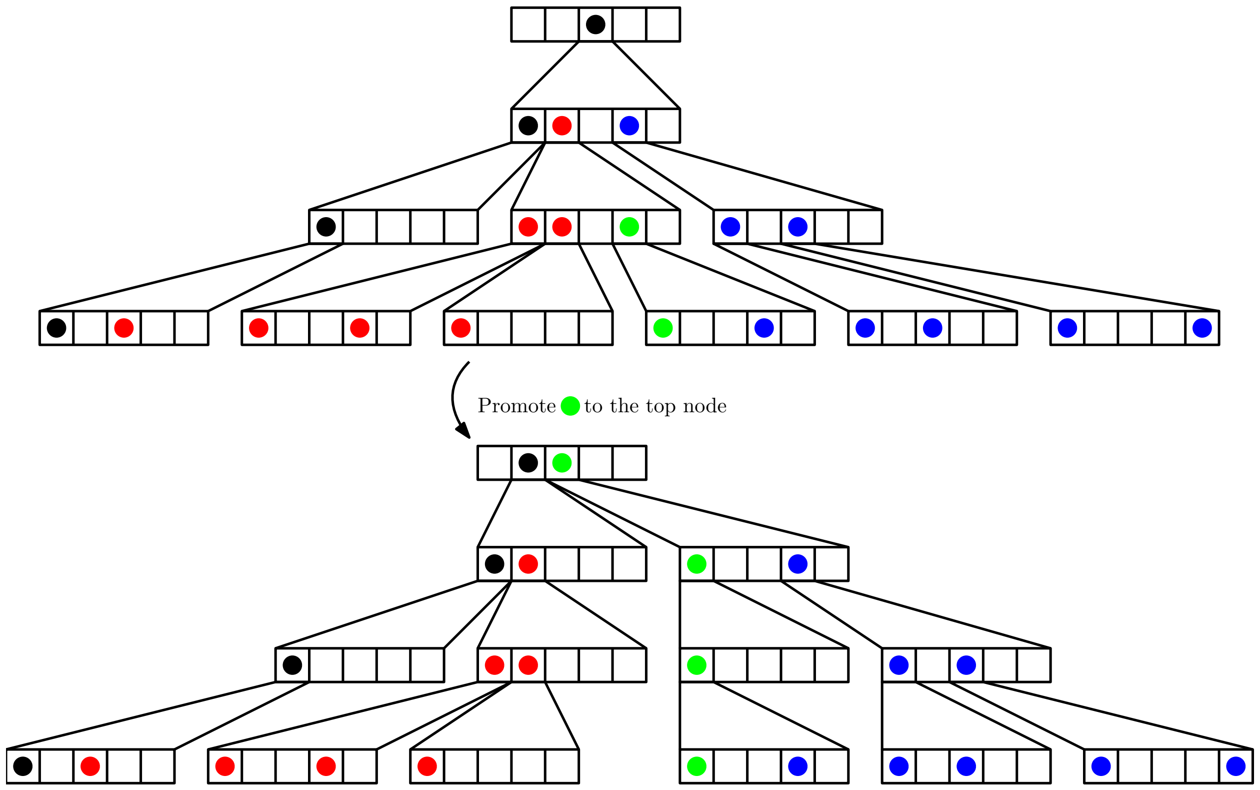


Figure 2.3: The black representative's subtree passed the threshold of the Balance Property and the weighted median representative, shown in green, is promoted. The subtrees of representatives less and greater than the median are copied and possibly shifted if they need a new root representative. Empty subtrees are omitted.

3. The previous step partitions some file maintenance instances into two pieces. In each such instance, respace the representatives it contains according to the split operation of Lemma 2.1. Notice that if the dummy representative is part of one piece, the data structure adds a new dummy representative in the other piece. This new dummy representative is a representative of e.

Examples of promotions are shown in Figures 2.2 and 2.3.

Lemma 2.2. *Promotions preserve the Order and Representative Properties.*

Proof. Before the promotion, the Order Property held and so an inorder traversal encountered the elements with label less than $\ell_T(e)$, then e, and then the elements with label greater than $\ell_T(e)$. After the promotion, the inorder traversal will traverse $T_{s(\hat{r})}$ which contains the elements less than e and then $T_{s(r')}$ which contains e followed by the elements greater than e. The two new subtrees preserved the original inorder traversal of their contained elements. So the inorder traversal before and after the promotion traverses the elements in T in the exact same order and the Order Property holds. The last step of a promotion ensures that the representatives of e still behave properly with respect to the Representative Property. Since we split along the path of elements less than and greater than e, no other path of representatives was altered and the Representative Property still holds. □

The only operations we perform on T are insertions of elements at leaves and promotions. Both of these preserve the Order and Representative Properties. Violations of the Capacity, Balance, and Counter Properties are fixed by promoting certain representatives. When a node u with parent v violates the Capacity Property, promote the median representative in f_u (which must be a canonical representative) into f_v. When the subtree of $s(r)$ becomes too heavy and violates the Balance Property, promote the weighted median canonical representative in the subtree of $s(r)$ into the node that contains r. When $c(e)$ passes the threshold of the Counter

Property, promote the canonical representative of e into the parent of its current node, and reset $c(e)$ to be zero. Figure 2.3 shows a promotion due to a violation of the Balance Property and Figure 2.2 shows a promotion due to a violation of the Capacity Property.

Promoting a representative may introduce new property violations. For example, suppose f_u for some internal node u contains $\kappa_1\gamma$ representatives and its parent v has exactly $\kappa_1\gamma - 1$ representatives. Promoting the median representative from f_u to f_v will cause the f_v to violate the Capacity Property.

The very rough pseudocode in Algorithms 1 and 2 describes the high level steps for insertions and promotions. We describe exactly how violations are processed next.

Algorithm 1 Insert x into the house numbering data structure immediately after element a.

```
1: function INSERT($x, a$)
2:     $u \leftarrow$ parent of $a$'s leaf
3:     insert a representative of $x$ into $f_u$
4:     move any relabeled elements to their new leaves
5:     place $x$ into the corresponding leaf of $u$
6:     repeatedly fix property violations using promote()
7: end function
```

Algorithm 2 Promote an element x into a node u

```
1: function PROMOTE($x, u$)
2:     $v \leftarrow$ node containing the canonical representative for $x$
3:     remove $x$ from $f_v$
4:     $a \leftarrow$ predecessor of $x$ in $f_u$
5:     insert a new canonical representative of $x$ after $a$ in $f_u$
6:     move the entire subtree of any relabeled elements
7:     split the subtree below $a$'s canonical representative into:
8:         - $T_1$ a subtree of elements $< x$ and $>= a$
9:         - $T_2$ a subtree of elements $>= x$
10:    place $T_1$ below $a$'s canonical representative
11:    place $T_2$ below $x$'s canonical representative
12: end function
```

Property violations. Since several properties may be violated at the same time, we employ the following prioritization for fixing these violations. We process the property violations by alternating between processing all violations of the Capacity and Balance Properties in a highest first fashion and then processing a single violation of the Counter Property. Algorithm 3 shows this procedure in pseudocode.

Algorithm 3 Process the violations in the tree

```
 1: function PROCESS_VIOLATIONS
 2:     while there is a violated property do
 3:         while there is a violation of the capacity or balance properties do
 4:             process the highest capacity or balance violation
 5:             (capacity violations have priority)
 6:         end while
 7:         if there is a violation of the counter property then
 8:             process one violation of the counter property
 9:         end if
10:     end while
11: end function
```

It is not yet clear that this processing terminates. We address this in Section 2.4. Whenever the initial insertion or a promotion causes an element to be relabeled, we increment the corresponding relabel counter.

During the processing of violations, a given relabel counter may be increased well past the bound in the Counter Property. But our potential argument only allows us to charge for relabelings that occur when the counter is at or below the threshold. To keep from relabeling the corresponding element each time an above-threshold counter is pushed even higher during a single (recursive) house numbering insertion, we perform the invariant violation processing on a logical copy of the data structure and only relabel elements with the final label. While a relabel counter may be incremented many times, any element is only relabeled at most once per insert operation.

2.4 Bounding the Height and Complexities

Let $H(n)$ denote an upper bound on the maximum possible height of a canonical representative in our house-numbering data structure after n elements are inserted. (We shall bound $H(n)$ as a function of γ in Lemma 2.6.) Then we can directly bound the length of labels at $H(n) \cdot \lceil \log(4\kappa_1\gamma + 1) \rceil$ bits. In particular, if κ_1 and γ are constants, we will prove that $H(n) = O(\log n)$ and hence the labels use $O(\log n)$ bits. Moreover, since we guarantee the Counter Property, each element e can be relabeled at most $\kappa_2\, cost(\kappa_1\gamma)$ times before $r(e)$ is moved up a level. So the maximum number of times that an element is relabeled is $O(\kappa_2 H(n)) = O(\kappa_2 \log n)$, assuming $\kappa_1\gamma$ is a constant and $H(n) = O(\log n)$.

The intuition behind our height analysis is as follows. Each insertion causes $cost(\kappa_1\gamma)$ representatives to be relabeled. Thus we need roughly κ_2 insertions to trigger enough relabels that a single representative could be promoted by the Counter Property. In other words, at most a $1/\kappa_2$ fraction of representatives are promoted due to insertions of elements and the Counter Property. This argument extends up the tree; promotions into height-h nodes can cause at most a $1/\kappa_2$ fraction of representatives to be promoted from height h. If this were the only effect, we would see $(1/\kappa_2)^h$ representatives promoted to height h.

This challenge turns out to be even more complex, since each promotion into a height-h node u also causes the elements in subtrees of any locally relabeled representatives in u to be completely relabeled. The Balance Property helps us to bound the total weight of representatives in these subtrees by $\kappa_3\gamma^h$. By increasing κ_2 enough, we effectively amortize the high number of relabelings due to moving a subtree against the geometrically decreasing number of promotions to that height, i.e., about $\kappa_3\gamma^h/\kappa_2^h$ per insertion. Since there are some "feedback" effects that arise from the interaction of fixing property violations, the analysis must proceed with care.

Before we turn to bounding the height, we prove a useful lemma.

Lemma 2.3. *If the Capacity, Balance, and Counter properties hold before an insertion, the processing of the resulting violations will never promote a representative into a node that:*

- *violates the Capacity Property or*
- *contains a representative whose subtree violates the Balance Property.*

Proof. Call a promotion into such a node an *invalid promotion.* We claim that in addition to never performing an invalid promotion, the violation processing maintains the property that violations of the Capacity and Balance Properties each occur at most once in each level of T. We call this the *Once Per Level Property.* When a representative is inserted or promoted into u: the usage of u is increased, the weight of every subtree of every representative on the path to the root is increased, and the relabel counters of any relabeled elements are increased. So an insertion or promotion will only introduce violations of the Capacity Property at u and violations of the Balance Property along the path from u to the root (and some other violations of the Counter Property). For example in Figures 2.2 and 2.3, each promotion can only create Counter Property violations lower in the subtree while it may introduce Capacity and Balance Property violations at the root. Hence the initial insertion or promotion from processing a Counter Property violation in a tree with no violations of the Capacity or Balance Properties is valid and will maintain the Once Per Level Property.

When the Once Per Level Property holds, there is some highest violation of each type. There is a highest node violating the Capacity Property and a highest node containing a representative violating the Balance Property. Let u be the higher of these two nodes. If both nodes have equal height, then let u be the highest node violating the Capacity Property. We consider the cases when u violates the Capacity Property or when u does not violate the Capacity Property and violates the Balance Property.

In the first case, f_u violates the Capacity Property by containing $\kappa_1\gamma$ representatives. Because the parent of u is not violating either of the two properties, promoting the median of f_u is valid. That promotion may introduce violations at the parent of u or higher, but they will only be in levels of the tree where there were previously no violations. The violations that were either at u or below will be unaffected by the promotion (except for the one being processed). Therefore the Once Per Level Property still holds after the violation is processed.

In the second case, f_u does not violate the Capacity Property but it does have one representative violating the Balance Property. Because no other representative in f_u violates the Balance Property due to the Once Per Level Property, processing the violation is valid. By promoting the weighted median descendant into u, only f_u can be newly in violation of the Capacity Property and only representatives in ancestors of u can be newly in violation of the Balance Property. Both of these types of new violations are introduced at levels that did not contain a violation of that type before. The splitting of the subtree below into two pieces can only eliminate violations in the levels below u. So after validly processing this violation, the Once Per Level Property holds.

In either case, processing a violation does not make an invalid promotion and maintains the Once Per Level Property. Thus, the invariant processing never promotes a representative into a node violating the Capacity Property or containing a representative whose subtree violates the Balance Property. $\square$

Potential argument. To formalize the intuition outlined in the beginning of this section, we analyze our data structure using the following three potential functions, each of which corresponds to one of our properties:

- $\Phi_{fmds} = \pi_1 \sum_{internal\ nodes\ u} \max\left(2\gamma^{height(u)} \cdot usage(f_u) - \kappa_1 \gamma^{height(u)+1}, 0\right)$
- $\Phi_{counters} = \sum_e w(r(e))c(e)$

- $\Phi_{tree} = \pi_2 \sum_u \max\left(2 \sum_{node\ v \in T(u)} \sum_{canonical\ representative\ r \in f_v} w(r) - \kappa_3 \gamma^{height(u)}, 0\right)$

The total potential, $\Phi(T)$, is the sum of these three potential functions, that is $\Phi = \Phi_{fmds} + \Phi_{counters} + \Phi_{tree}$. Each potential function corresponds to one of our three properties and guarantees that when a property is violated we have sufficient potential to "pay" for the promotion.

The next few lemmas show how the potential functions work with the properties. The change in Φ due to a processing a violation can be separated into the two phases of a promotion. First, there is the decrease in potential when the promoted element's relabel counter is reset and the node containing the canonical representative of the element is split. Second, there is the increase in potential due to the insertion of the canonical representative of the element into a higher up node which results in relabeling many other elements, increasing that node's usage, and increasing the weight of every subtree containing the higher up node. Lemma 2.4 gives an upper bound on the increase in potential due to either an insertion or the second part of a promotion. Lemma 2.5 gives a lower bound on the decrease in potential due to the first part of a promotion. In conjunction these two Lemmas show that as long as the maximum height $H(n)$ is small, the lower bound on the decrease in potential is greater than the upper bound on the increase in potential. So a promotion results in a net decrease in potential and only insertions of new elements at the leaves increase the potential. Finally Lemma 2.6 contrasts the amount of potential gained from these insertions with the amount of potential needed to promote one representative to a height of $\log_\gamma n$. Because the former is strictly smaller, the height of the tree must be $H(n) < \log_\gamma n$.

Lemma 2.4. *During a promotion, the insertion of a representative into a file maintenance data structure at height h increases $\Phi(T)$ by at most $(2\pi_1 + \kappa_3\, cost(\kappa_1\gamma) + 2\pi_2\, height(T))\gamma^h$. Moreover, the insertion of an element increases $\Phi(T)$ by at most $2\pi_1 + \kappa_3\, cost(\kappa_1\gamma) + 2\pi_2\, height(T)$.*

Proof. Placing a canonical representative into a node u at height h causes the potential functions to change as follows:

- $\Delta(\Phi_{fmds}) \leq 2\pi_1\gamma^h$, because f_u had its size increased by 1
- $\Delta(\Phi_{counters}) \leq \kappa_3\gamma^h\, cost(\kappa_1\gamma)$, because $cost(\kappa_1\gamma)$ representatives in f_u are relabeled, each causing subtrees with total weight at most $\kappa_3\gamma^h$ to be relabeled.
- $\Delta(\Phi_{tree}) \leq 2\pi_2\, height(T)\gamma^h$, because each representative on the path to the root has the potential of its subtree increased by at most γ^h

The bound on $\Delta(\Phi_{counters})$ deserves some more elaborate explanation. Because by Lemma 2.3 we never promote a representative into a node that is violating the Capacity Property, there are at most $cost(\kappa_1\gamma)$ relabels by Lemma 2.3. Lemma 2.3 also implies we never promote into a node violating the Balance Property, and so incrementing the relabel counters of each subtree costs at most $\kappa_3\gamma^h$ potential. Combining these two facts gives us the stated bound. Note that the $cost(\kappa_1\gamma)$ term is an amortized upper bound across all of the file maintenance data structure insertions we perform. So one insertion of a canonical element may individually violate this inequality, but the increase can be amortized across the entire sequence of file maintenance insertions our data structure performs.

In total these sum up to $(2\pi_1 + \kappa_3\, cost(\kappa_1\gamma) + 2\pi_2\, height(T))\gamma^h$ which upper bounds the increase in all three potential functions. $\square$

Lemma 2.5. *If $height(T) \leq H(n)$, there exist settings of π_i's and κ_i's such that promoting a representative to fix a violation does not increase the total potential and $\kappa_2 = O(\gamma H(n))$.*

Proof. Depending on which violation caused the promotion, we must analyze the decrease in potential differently to account for the potential increase from Lemma 2.4 of

$(2\pi_1 + \kappa_3\, cost(\kappa_1\gamma) + 2\pi_2 H(n))\,\gamma^h$ where h is again the height of the node the promoted element is moved into.

- If a Capacity Property violation was processed, then Φ_{fmds} decreased by at least $\pi_1\kappa_1\gamma^h$.
- If a Counter Property violation was processed, then $\Phi_{counters}$ decreased by at least $\gamma^{h-1}\kappa_2\, cost(\kappa_1\gamma)$ due to the potential from $c(e)$.
- If a Balance Property violation was processed, then Φ_{tree} decreased by more than $\pi_2\kappa_3\gamma^h$ because of the subtree containing r.

To ensure the potential available is always at least the potential cost $\pi_1\kappa_1$, $\frac{\kappa_2\, cost(\kappa_1\gamma)}{\gamma}$, and $\pi_2\kappa_3$ must all be greater than $2\pi_1 + \kappa_3\, cost(\kappa_1\gamma) + 2\pi_2 H(n)$. Analyzing the system of inequalities leads to setting $\kappa_1 = 3$, $\kappa_2 = 72\gamma H(n)$, $\kappa_3 = 12H(n)$, $\pi_1 = 24\, cost(3\gamma)H(n)$, and $\pi_2 = 6\, cost(3\gamma)$.

Plugging these values back into the original formula, the potential increases by at most

$$(2(24\, cost(3\gamma)H(n)) + (12H(n))\, cost(3\gamma) + 2(6\, cost(3\gamma))H(n))\gamma^h = 72\, cost(3\gamma)H(n)\gamma^h.$$

On the other hand, the potential decrease is at least

- $(24\, cost(3\gamma)H(n))(3)\gamma^h$ in the case of a Capacity Property violation,
- $\gamma^{h-1}(72\gamma H(n))\, cost(3\gamma)$ in the case of a Counter Property violation, or
- $(6\, cost(3\gamma))(12H(n))\gamma^h$ in the case of a Balance Property violation.

In all three cases, the lower bound of the decrease in potential is equal to $72\, cost(3\gamma)\gamma^h H(n)$ and therefore it is at least the increase in potential due to a promotion. $\square$

Lemma 2.6. *For the same setting of π_i's and κ_i's as Lemma 2.5, and $\gamma \leq n$, after n insertions there are no promotions to height above $\log_\gamma n$.*

Proof. Initially the tree is empty and has height zero. By Lemma 2.5, setting $\log_\gamma n = H(n)$, until $height(T)$ exceeds $H(n)$ promoting a representative does not increase the potential. Thus, while the height bound holds the only mechanism for increasing the potential is by inserting a new element. By Lemma 2.4, the increase in potential from inserting an element is at most $72\,cost(3\gamma)\log_\gamma n$ and so after n insertions, the potential of the entire data structure is at most $72\,cost(3\gamma)n\log n$.

To complete the proof, we observe that a representative can only be promoted to height h if the total potential in the data structure is at least $72\,cost(3\gamma)\gamma^h \log_\gamma n$. Specifically, the proof of Lemma 2.5 shows that the potential has to decrease by at least this amount when performing the promotion, so the potential has to exist before the promotion. In order to reach a height of at least $\log_\gamma n + 1$, we would need at least $72\,cost(3\gamma)\gamma^{\log_\gamma n+1}\log_\gamma n > 72\,cost(3\gamma)n\log n$ potential. Thus, a height $\log_\gamma n$ structure cannot have enough potential. $\square$

Theorem 2.4. *There exists a house numbering data structure with complexity $\langle O(\gamma\log^2 n), \log_\gamma n \cdot \lceil\log(12\gamma+1)\rceil\rangle$.*

Proof. By Lemma 2.6, after n insertions there are no promotions into nodes at height higher than $\log_\gamma n$, so a tree of this height suffices. Thus, each label uses $\log_\gamma n$ "digits", where each digit uses $\lceil\log(12\gamma+1)\rceil$ bits, for a total of $\log_\gamma n \cdot \lceil\log(12\gamma+1)\rceil$ bits per label. For the relabel bound, by the Counter Property, each canonical representative is promoted at most $\kappa_2\,cost(3\gamma) = O(\gamma H(n))\,cost(3\gamma) = O(\gamma\log_\gamma n\cdot\log^2\gamma)$ times. Summing on all possible levels and applying Lemma 2.1, each element is only relabeled $O(\gamma\log^2 n)$ times. $\square$

Theorem 2.1 is a special case of Theorem 2.4 obtained by setting $\gamma = 2$.

2.4.1 Achieving $\langle O(n^\epsilon), \log n + O(1/\epsilon)\rangle$

Proof of Theorem 2.3. Setting $\gamma = n^\epsilon$ in Theorem 2.4, the number of bits used is $1/\epsilon\lceil\log(12n^\epsilon + 1)\rceil = \log n + O(1/\epsilon)$. The maximum relabel bound becomes $O(n^\epsilon \log^2(n^\epsilon)) = O(n^\epsilon \log^2 n)$. That is, when $\gamma = n^\epsilon$, it is an $\langle O(n^\epsilon \log^2 n), \log n + O(1/\epsilon)\rangle$ house numbering data structure. This bound is improved to $\langle O(n^\epsilon), \log n + O(1/\epsilon)\rangle$ by using the same solution with $\gamma = n^{\epsilon/2}$. □

2.5 A $\left\langle O(\log n), O(\log^2 n)\right\rangle$ house numbering data structure

Our $\left\langle O(\log n), O(\log^2 n)\right\rangle$ house numbering data structure also maintains a tree T' of nodes of instances of file maintenance data structures. In this data structure, the file maintenance data structure instances do not all use the same label range. Instead, the instance f_u for node u at height h is set to store at most $2^{h+1} - 1$ representatives. The data structure uses the same labelling method for the representatives and so the Representative Property holds. Level h uses $\log(2^{h+1}) = h + 1$ bits and the total number of bits used for a label is, again, going to depend on the height of the tree.

The insert operation. On an $\mathsf{insert}(x, a)$ operation insert x into the appropriate leaf bucket and its immediate parent's file maintenance data structure. If an attempted insertion would overflow a node, take every representative in the subtree rooted at that node and insert them into the parent. If this immediately overflows the parent, recursively repeat the process until a node is large enough. An example of this cascading overflow process can be seen in Figure 2.4.

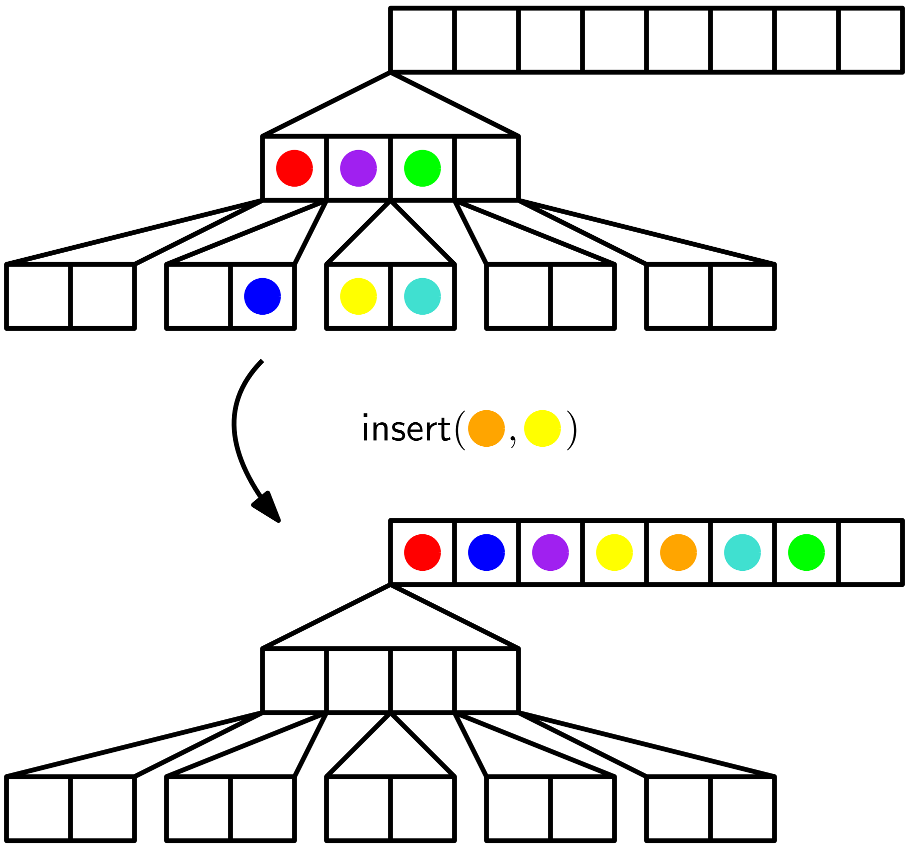


Figure 2.4: An insertion causes a cascade of overflowing buckets until a node is reached which can contain the whole subtree. Only canonical elements are drawn.

Lemma 2.7. *After n insertions the height of the tree is at most* $\log n$.

Proof. We only insert into a higher node when the previously highest node was filled up. A node at height $\log n$ has room for $2^{\log n+1} - 1 = 2n - 1$ representatives and so will never be overflowed. ☐

As representatives are moved throughout T', they cause relabels to other representatives. We say a relabeling of a representative is *caused by an insertion* if the representative was in the node being inserted into, was one of the representatives being inserted, or was in a descendant of that node.

Lemma 2.8. *For any h, over the course of n insertions, a representative undergoes at most two relabelings caused by insertions at level h.*

Proof. Each node has size one less than twice the size of its immediate children. So when a node overflows the parent is filled at least halfway after the insertions. Therefore a set of insertions into any node can only happen twice before that node is overfilled. Each of these insertions causes every descendant of the node to be relabeled. Once the node is overfilled every descendant will be inserted into a node at level $h + 1$ and can no longer be affected by insertions into nodes at lower levels. ☐

Proof of Theorem 2.2. By Lemma 2.7, the height of the tree is at most $\log n - 1$ and we use at most $\sum_{h=0}^{\log n} h = \binom{\log n+1}{2}$ bits. Because of Lemma 2.8, any representative is relabeled at most twice per level of the tree for a total of at most $2 \log n$ relabels. ☐

2.6 Conclusion

The house numbering problem is an interesting variant of the very well studied file maintenance and online list labelling problems. It poses some unique challenges that previous techniques do not solve. Our two data structures are able to come near optimal for the problem, but an $\langle O(\log n), O(\log n)\rangle$ house numbering data structure remains as an open problem.

Chapter 3

Parallel Equivalence Class Sorting *

3.1 Introduction

In the *Equivalence Class Sorting* problem, we are given a set, S, of n elements and an equivalence relation, and we are asked to group the elements of the set into their equivalence classes by only making pairwise equivalence tests (e.g., see [56]). For example, imagine a convention of n political interns where each person at the convention belongs to one of k political parties, such as Republican, Democrat, Green, Labor, Libertarian, etc., but no intern wants to openly express their party affiliation unless they know they are talking with someone of their same party. Suppose further that each party has a secret handshake that two people can perform that allows them to determine whether they are in the same political party (or they belong to different unidentified parties). We are interested in this chapter in the computational complexity of the equivalence class sorting problem in distributed and parallel settings, where we would like to minimize the total number of parallel

*This chapter is included with permission from ACM and appeared in W. E. Devanny, M. T. Goodrich, and K. Jetviroj, "Parallel Equivalence Class Sorting: Algorithms, Lower Bounds, and Distribution-Based Analysis," Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, 2016 © 2016 ACM. [31]

comparison rounds and/or the total number of comparisons needed in order to classify every element in S.

An important property of the equivalence class sorting problem is that it is not possible to order the elements in S according to some total ordering that is consistent with the equivalence classes. Such a restriction could come from a general lack of such an ordering or from security or privacy concerns. Additionally hashing elements by their equivalence class is not permitted. For example, consider the following applications:

- *Generalized fault diagnosis.* Suppose that each of n different computers are in one of k distinct malware states, depending on whether they have been infected with various computer worms. Each worm does not wish to reveal its presence, but it nevertheless has an ability to detect when another computer is already infected with it (or risk autodetection by an exponential cascade, as occurred with the Morris worm [69]). But a worm on one computer is unlikely to be able to detect a different kind of worm on another computer. Thus, two computers can only compare each other to determine if they have exactly the same kinds of infections or not. The generalized fault diagnosis problem, therefore, is to have the n computers classify themselves into k malware groups depending on their infections, where the only testing method available is for two computers to perform a pairwise comparison that tells them that they are either in the same malware state or they are in different states. This is a generalization of the classic fault diagnosis problem, where there are only two states, "faulty" or "good," which is studied in a number of interesting papers, including one from the very first SPAA conference (e.g., see [11–13, 43, 72, 73]).
- *Group classification via secret handshakes.* This is a cryptographic analogue to the motivating example given above of interns at a political convention. In this case, n agents are each assigned to one of k groups, such that any two agents can perform a cryptographic "secret handshake" protocol that results in them learning only whether

they belong to the same group or not (e.g., see [24, 55, 76, 83]). The problem is to perform an efficient number of pairwise secret-handshake tests in a few parallel rounds so that each agent identifies itself with the others of its group.

- *Graph mining.* Graph mining is the study of structure in collections of graphs [28]. One of the algorithmic problems in this area is to classify which of a collection of n graphs are isomorphic to one another (e.g., see [70]). That is, testing if two graphs are in the same group involves performing a graph isomorphism comparison of the two graphs, which is a computation that tends to be nontrivial but is nevertheless computationally feasible in some contexts (e.g., see [9]).

Note that each of these applications contains two important features that form the essence of the equivalence class sorting problem:

1. In each application, it is not possible to sort elements according to a known total order, either because no such total order exists or because it would break a security/privacy condition to provide such a total order.
2. The equivalence or nonequivalence between two elements can be determined only through pairwise comparisons.

There are nevertheless some interesting differences between these applications, as well, which motivate our study of two different versions of the equivalence class sorting problem. Namely, in the first two applications, the comparisons done in any given round in an algorithm must be disjoint, since the elements themselves are performing the comparisons. In the latter two applications, however, the elements are the objects of the comparisons, and we could, in principle, allow for comparisons involving multiple copies of the same element in each round. For this reason, we allow for two versions of the equivalence class sorting problem:

- *Exclusive-Read (ER)* version. In this version, each element in S can be involved in at most a single comparison of itself and another element in S in any given comparison round.
- *Concurrent-Read (CR)* version. In this version, each element in S can be involved in multiple comparisons of itself and other elements in S in any comparison round.

In either version, we are interested in minimizing the number of parallel comparison rounds and/or the total number of comparisons needed to classify every element of S into its group.

Because we expect the number of parallel comparison rounds and the total number of comparisons to be the main performance bottlenecks, we are interested here in studying the equivalence class sorting problem in Valiant's parallel comparison model [78], which only counts steps in which comparisons are made. This is a synchronous computation model that does not count any steps done between comparison steps, for example, to aggregate groups of equivalent elements based on comparisons done in previous steps.

3.1.1 Related Prior Work

In addition to the references cited above that motivate the equivalence class sorting problem or study the special case when the number of groups, k, is two, Jayapaul *et al.* [56] study the general equivalence class sorting problem, albeit strictly from a sequential perspective. For example, they show that one can solve the equivalence class sorting problem using $O(n^2/\ell)$ comparisons, where ℓ is the size of the smallest equivalence class. They also show that this problem has a lower bound of $\Omega(n^2/\ell^2)$ even if the value of ℓ is known in advance.

The equivalence class sorting problem is, of course, related to comparison-based algorithms for computing the majority or mode of a set of elements, for which there is an extensive set of prior research (e.g., see [3, 4, 35, 74]). None of these algorithms for majority or mode result in efficient parallel algorithms for the equivalence class sorting problem, however.

In some contexts, it may be possible to hash elements according to their equivalence class. For example, when counting word or item frequencies, a well known linear time solution is to use a hash table. If the equivalence classes can be totally ordered, sorting the elements by their equivalence class leads to an $O(n \log n)$ time solution.

3.1.2 Our Results

In this chapter, we study the equivalence class sorting (ECS) problem from a parallel perspective, providing a number of new results, including the following:

1. The CR version of the ECS problem can be solved in $O(k + \log \log n)$ parallel rounds using n processors, were k is the number of equivalence classes.
2. The ER version of the ECS problem can be solved in $O(k \log n)$ parallel rounds using n processors, were k is the number of equivalence classes.
3. The ER version of the ECS problem can be solved in $O(1)$ parallel rounds using n processors, for the case when ℓ is at least λn, for a fixed constant $0 < \lambda \leq 0.4$, where ℓ is the size of the smallest equivalence class.
4. If every equivalence class is of size f, then solving the ECS problem requires $\Omega(n^2/f)$ total comparisons. This improves a lower bound of $\Omega(n^2/f^2)$ by Jayapaul *et al.* [56].
5. Solving the ECS problem requires $\Omega(n^2/\ell)$ total comparisons, where ℓ is the size of the smallest equivalence class. This improves a lower bound of $\Omega(n^2/\ell^2)$ by Jayapaul *et al.* [56].
6. In Section 3.4, we study how to efficiently solve the ECS problem when the input is drawn from a known distribution on equivalence classes. In this setting, we assume

n elements have been sampled and fed as input to the algorithm. We establish a relationship between the mean of the distribution and the algorithm's total number of comparisons, obtaining upper bounds with high probability for a variety of interesting distributions.

7. We provide the results of several experiments to validate the results from Section 3.4 and study how total comparison counts change as parameters of the distributions change.

Our methods are based on several novel techniques, including a two-phased compounding-comparison technique for the parallel upper bounds and the use of a new coloring argument for the lower bounds.

3.2 Parallel Algorithms

In this section, we provide efficient parallel algorithms for solving the equivalence class sorting (ECS) problem in Valiant's parallel model of computation [78]. We focus on both the exclusive-read (ER) and concurrent-read (CR) versions of the problem, and we assume we have n processors, each of which can be assigned to one equivalence comparison test to perform in a given parallel round. Note, therefore, that any lower bound, $T(n)$, on the total number of comparisons needed to solve the ECS problem (e.g., as given by Jayapaul *et al.* [56] and as we discuss in Section 3.3), immediately implies a lower bound of $\Omega(T(n)/n)$ for the number of parallel rounds of computation using n processors per round. For instance, these lower bounds imply that the number of parallel rounds for solving the ECS problem with n processors must be $\Omega(n/\ell)$ and $\Omega(k)$, respectively, where k is the number of equivalence classes and ℓ is the size of the smallest equivalence class.

With respect to upper bounds, recall that Jayapaul *et al.* [56] studied the ECS problem from a sequential perspective. Unfortunately, their algorithm cannot be easily parallelized, because the comparisons performed in a "round" of their algorithm depend on the results from other comparisons in that same round. Thus, new parallel ECS algorithms are needed.

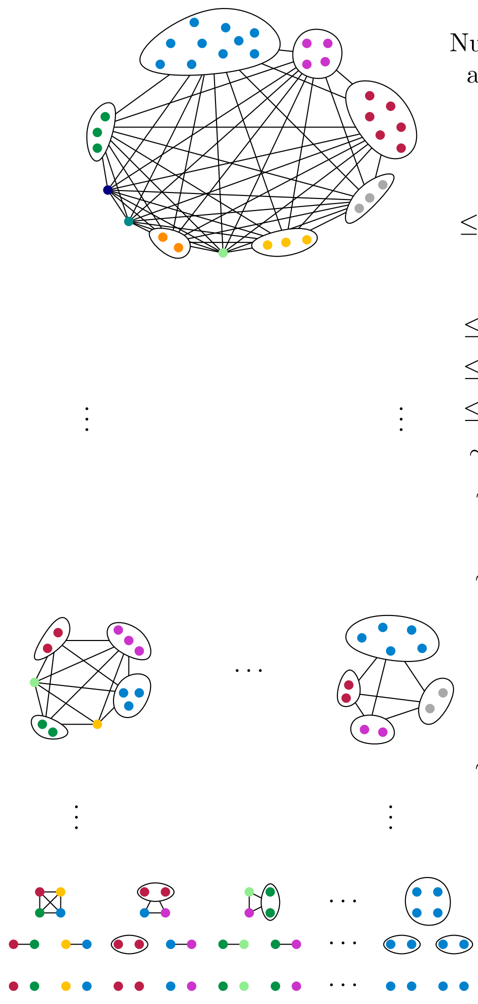
3.2.1 Algorithms Based on the Number of Groups

In this subsection, we describe CR and ER algorithms based on knowledge of the number of groups, k.

If two sets of elements are partitioned into their equivalence classes, merging the two partitions of elements into the equivalence class partition for the union of the two sets requires at most k^2 equivalence tests by simply performing a comparison between every pair of equivalence classes: one from the first partition and one from the partition. This idea leads to the following algorithm, which uses a two-phased compounding-comparison technique to solve the ECS problem:

- Initialize a list of n sets containing the individual input elements and associate with each set the trivial partition containing the single element in the set.
- While the number of processors per partition is less than $4k^2$, merge pairs of equivalence class partitions by performing all k^2 pairwise comparisons between the k equivalence classes in the two partitions.
- While there is more than one equivalence class partition remaining, if m is the number of processors per partition remaining, then let $c = \frac{m}{k^2}$ and merge c partitions together simultaneously by performing all k^2 pairwise comparisons between the k equivalence classes for each of the $\binom{c}{2}$ pairs of partitions.

We analyze this algorithm in the following two lemmas and we illustrate it in Figure 3.1.



Number of answers	Processors per answer	Answer size	Rounds needed	Answer reduction factor	
1	n	k			
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
$\leq n/2^{2^i}k^2$	$\geq 2^{2^i}k^2$	$\leq k$	1	$\sim 2^{2^i}$	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	Second while loop phase
$\leq n/2^8k^2$	$\geq 2^8k^2$	$\leq k$	1	$\sim 2^8$	
$\leq n/2^4k^2$	$\geq 2^4k^2$	$\leq k$	1	2^4	
$\leq n/2^2k^2$	$\geq 2^2k^2$	$\leq k$	1	2^2	
$\sim n/2k^2$	$\sim 2k^2$	$\leq k$	1	2	
$\sim n/k^2$	$\sim k^2$	$\leq k$	1	2	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
$\sim n/2k$	$\sim 2k$	$\leq k$	$k/2$	2	
$\sim n/k$	$\sim k$	$\leq k$	k	2	First while loop phase
$\sim 2n/k$	$\sim k/2$	$\leq k/2$	$k/2$	2	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
n/4	4	≤ 4	4	2	
$n/2$	2	≤ 2	2	2	
n	1	1	1	2	

Figure 3.1: A visualization of the parallel algorithm with a table on the right keeping track of relevant numbers for each loop iteration.

Lemma 3.1. *The first while loop takes $O(k)$ rounds of parallel comparisons to complete.*

Proof. In each round of parallel comparisons, the number of equivalence classes in a partition at most doubles until it reaches the upper bound of k. In loop iteration $i \leq \lceil \log k \rceil$, the partitions are size at most 2^i and there are 2^i processors per set. Therefore it takes at most 2^i rounds to merge two partitions. So the number of rounds to reach loop iteration $\lceil \log k \rceil$ is $O(k)$.

For loop iterations $\lceil \log k \rceil < i < \lceil \log k \rceil^2$, the partitions are size at most k, but there are still 2^i processors per partition. Therefore the number of processors available doubles each loop iteration. The total number of parallel comparison rounds needed for these iterations is also $O(k)$, as it forms a geometric sum that adds up to be $O(k)$. This part of the algorithm is illustrated in the bottom half of Figure 3.1. □

Lemma 3.2. *The second while loop takes $O(\log \log n)$ rounds to complete.*

Proof. When entering the second while loop, there are many more processors per partition than needed to merge just two partitions. If a set has access to ck^2 processors, then a group of $\binom{c}{2}$ partitions can merge into one partition in a single round. This means that if there are $n/(ck^2)$ sets at the start of a round, then we merge groups of $c^2/2$ partitions into one partition and there are $n/(c^3k/2)$ partitions remaining. Because $c \geq 4$ by the condition of the first while loop, in iteration i of the second while loop, there are at most $n/(2^{2^i}k)$ partitions. And so the second while loop will terminate after $O(\log \log n)$ rounds with the single partition for the entire input. This is illustrated in the top half of Figure 3.1. □

Combining these two lemmas, we get the following.

Theorem 3.1. *The CR version of the equivalence class sorting problem on n elements and k equivalence classes can be solved in $O(k + \log\log n)$ parallel rounds of equivalence tests, using n processors in Valiant's parallel comparison model.*

Proof. Lemmas 3.1 and 3.2. □

We also have the following.

Theorem 3.2. *The ER version of the equivalence class sorting problem on n elements and k equivalence classes can be solved in $O(k \log n)$ parallel rounds of equivalence tests, using n processors in Valiant's parallel comparison model.*

Proof. Merging two equivalence class partitions for the ER version of the ECS problem model will always take at most k rounds. Repeatedly merging equivalence partitions will arrive at one partition in $\log n$ iterations. So equivalence class sorting can be done in $O(k \log n)$ parallel rounds of equivalence tests. □

3.2.2 Algorithms Based on the Smallest Group Size

In this subsection, we describe ER algorithms based on knowledge of ℓ, the size of the smallest equivalence class. We assume in this section that $\ell \geq \lambda n$, for some constant $\lambda > 0$, and we show how to solve the ECS problem in this scenario using $O(1)$ parallel comparison rounds. Our methods are generalizations of previous methods for the parallel fault diagnosis problem when there are only two classes, "good" and "faulty" [11–13, 43]. Let us assume, therefore, that there are at least 3 equivalence classes. We begin with a theorem from Goodrich [43].

Theorem 3.3 (Goodrich [43]). *Let V be a set of n vertices, and let $0 < \gamma, \lambda < 1$. Let $H_d = (V, E)$ be a directed graph defined by the union of d independent randomly-chosen*[1] *Hamiltonian cycles on V (with all such cycles equally likely). Then, for all subsets W of V of λn vertices, H_d induces at least one strongly connected component on W of size greater than $\gamma\lambda n$, with probability at least*

$$1 - e^{n[(1+\lambda)\ln 2 + d(\alpha \ln \alpha + \beta \ln \beta - (1-\lambda)\ln(1-\lambda))] + O(1)},$$

where $\alpha = 1 - \frac{1-\gamma}{2}\lambda$ and $\beta = 1 - \frac{1+\gamma}{2}\lambda$.

In the context of the present chapter, let us take $\gamma = 1/4$, so $\alpha = 1-(3/8)\lambda$ and $\beta = 1-(5/8)\lambda$. Let us also assume that $\lambda \le 0.4$, since we are considering the case when the number of equivalence classes is at least 3; hence, the smallest equivalence class is of size at most $n/3$.

Unfortunately, standard approximations for the natural logarithm are not strong enough for us to employ the above probability bound and achieve our desired result for small values of λ. So instead we use the following inequalities, which hold for x in the range $[0, 0.4]$ (e.g., see [62]), and are based on the Taylor series for the natural logarithm:

$$-x - \frac{x^2}{2} - \frac{x^3}{2} \le \ln(1-x) \le -x - \frac{x^2}{2} - \frac{x^3}{4}.$$

These bounds allow us to bound the term, $t = \alpha \ln \alpha \; + \; \beta \ln \beta - (1-\lambda)\ln(1-\lambda)$, in the

[1]That is, H_d is defined by the union of cycles determined by d random permutations of the n vertices in V, so H_d is, by definition, a simple directed graph.

above probability of Theorem 3.3 (for $\gamma = 1/4$) as follows:

$$\begin{aligned}
t &= (1-\frac{3}{8}\lambda)\ln(1-\frac{3}{8}\lambda) \;+\; (1-\frac{5}{8}\lambda)\ln(1-\frac{5}{8}\lambda) \\
&\quad - (1-\lambda)\ln(1-\lambda) \\
&\le (1-\frac{3}{8}\lambda)\left(-\frac{3}{8}\lambda - \frac{1}{2}\left(\frac{3}{8}\lambda\right)^2 - \frac{1}{4}\left(\frac{3}{8}\lambda\right)^3\right) \\
&\quad + (1-\frac{5}{8}\lambda)\left(-\frac{5}{8}\lambda - \frac{1}{2}\left(\frac{5}{8}\lambda\right)^2 - \frac{1}{4}\left(\frac{5}{8}\lambda\right)^3\right) \\
&\quad - (1-\lambda)\left(-\lambda - \frac{\lambda^2}{2} - \frac{\lambda^3}{2}\right) \\
&\le -\frac{3743}{8192}\lambda^4 + \frac{19}{256}\lambda^3 - \frac{15}{64}\lambda^2,
\end{aligned}$$

which, in turn, is at most

$$-\frac{\lambda^2}{8},$$

for $0 < \lambda \le 0.4$. Thus, since t is negative for any constant $0 < \lambda \le 0.4$, we can set d large enough (depending on λ) so that $dt + (1+\lambda)\ln 2) < 0$ and Theorem 3.3 holds with high probability.

Our ECS algorithm, then, is as follows:

1. Construct a graph, H_d, as in Theorem 3.3, as described above, with d set to a constant so that the theorem holds for the fixed λ in the range $(0, 0.4]$ that is given. Note that this step does not require any comparisons; hence, we do not count the time for this step in our analysis (and the theorem holds with high probability in any case).

2. Note that H_d is a union of d Hamiltonian cycles. Thus, let us perform all the comparisons in H_d in $2d$ rounds. Furthermore, we can do this set of comparisons even for the ER

version of the problem. Moreover, since d is $O(1)$, this step involves a constant number of parallel rounds (of $O(n)$ comparisons per round).

3. For each strongly connected component, C, in H_d consisting of elements of the same equivalence class, compare the elements in C with the other elements in S, taking $|C|$ at a time. By Theorem 3.3, $|C| \geq \lambda n/8$. Thus, this step can be performed in $O(1/\lambda) = O(1)$ rounds for each connected component; hence it requires $O(1)$ parallel rounds in total. Moreover, after this step completes, we will necessarily have identified all the members of each equivalence class.

We summarize as follows.

Theorem 3.4. *Suppose S is a set of n elements, such that the smallest equivalence class in S is of size at least λn, for a fixed constant, λ, in the range $(0, 0.4]$. Then the ER version of the equivalence class sorting problem on S can be solved in $O(1)$ parallel rounds using n processors in Valiant's parallel comparison model.*

This theorem is true regardless of whether or not the true λ is known. If the true value of λ is not known, it is possible to repeatedly run the ECS algorithm starting with an arbitrary constant of 0.4 for λ and halving the constant whenever any of the final equivalence classes are smaller than λn for the current value of λ. Once the value is less than the unknown true λ, the algorithm will succeed and the number of rounds will be independent of n and a function of only the constant λ.

As we show in the next section, this performance is optimal when $\ell \geq \lambda n$, for a fixed constant $\lambda \in (0, 0.4]$.

3.3 Lower Bounds

The following lower bound questions were left open by Jayapaul *et al.* [56]:

- If every equivalence class has size f, is the total number of comparisons needed to solve the equivalence class sorting problem $\Theta(n^2/f)$ or $\Theta(n^2/f^2)$?
- Is the total number of comparisons for finding an element in the smallest equivalence class $\Theta(n^2/\ell)$ or $\Theta(n^2/\ell^2)$?

Speaking loosely these lower bounds can be thought of as a question of how difficult it is for an element to locate its equivalence class. The $\Theta(n^2/f)$ and $\Theta(n^2/\ell)$ bounds can be interpreted as saying the average element needs to compare to at least one element in most of the other equivalence classes before it finds an equivalent element. Because there must be $\binom{x}{2}$ comparisons between x equivalence classes, the $\Theta(n^2/f^2)$ and $\Theta(n^2/\ell^2)$ bounds say we do not need too many more comparisons then the very minimal number needed just to differentiate the equivalence classes. It seems unlikely that so few comparisons are required and we prove that this intuition is correct by proving lower bounds of $\Omega(n^2/f)$ and $\Omega(n^2/\ell)$ comparisons.

Note that these lower bounds are on the total number of comparisons needed to accomplish a task, that is they bound the work a parallel algorithm would need to perform. By dividing by n, they also give simple bounds on the number of rounds needed in either the ER or CR models.

With respect to such lower bound questions as these, let us maintain the state of an algorithm's knowledge about element relationships in a simple graph. At each step, the vertex set of this graph is a partition of the elements where each set is a partially discovered equivalence class for S. Thus, each element in S is associated with exactly one vertex in this graph at each step of the algorithm, and a vertex can have multiple elements from S associated with it. If

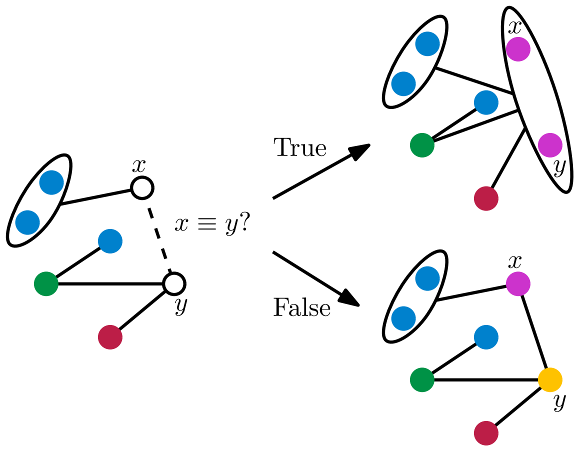


Figure 3.2: We test if x and y are in the same equivalence class. If they are, their vertices are contracted together. If they are not, an edge is added.

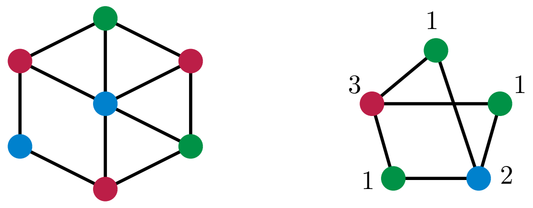


Figure 3.3: On the left we have a graph with an equitable 3-coloring and on the right we have a graph with a weighted equitable 3-coloring.

a pair of elements was compared and found to not be equal, then there should be an edge in between the two vertices containing those elements. So initially the graph has a vertex for each element and no edges. When an algorithm tests equivalence for a pair of elements, then, if the elements are not equivalent, the appropriate edge is added (if it is absent) and, if the elements are equivalent, the two corresponding vertices are contracted into a new vertex whose set is the union of the two. A depiction of this is shown in Figure 3.2. An algorithm has finished sorting once this graph is a clique and the vertex sets are the corresponding equivalence classes.

An *equitable k-coloring* of a graph is a proper coloring of a graph such that the size of each color class is either $\lfloor n/k \rfloor$ or $\lceil n/k \rceil$. A *weighted equitable k-coloring* of a vertex weighted graph is a proper coloring of a graph such that the sum of the weight in each color class is either $\lfloor n/k \rfloor$ or $\lceil n/k \rceil$. Examples of these can be seen in Figure 3.3. Equitable coloring was introduced by Meyer [66].

An adversary for the problem of equivalence class sorting when every equivalence class has the same size f (so f divides n) must maintain that the graph has a weighted equitable n/f-coloring where the weights are the size of the vertex sets. The adversary we describe here will maintain such a coloring and additionally mark the elements and the color classes in a special way. It proceeds as follows.

First, initialize a starting graph that consists of n vertices and no edges and arbitrarily color the graph in an equitable maner. For each comparison of two elements performed by a sorting algorithm, the adversary reacts based on the following case analysis:

- If either of the two elements is unmarked and this comparison would increase its degree to higher than $n/4f$, then the adversary marks it as having "high" element degree.
- If either of the two elements is still unmarked, the two elements currently have the same color, and there is another unmarked vertex such that it is not adjacent to a vertex with the color involved in the comparison and no vertex with its color is adjacent to the unmarked vertex in the comparison (i.e. we can have it swap colors with one of the vertices in the comparison), then the adversary swaps the color of that element and the unmarked element in the comparison.
- If either of the two elements is still unmarked, the two elements currently have the same color, and there is no other unmarked vertex with a different unmarked color not adjacent to the color of the two elements being compared, then the adversary marks all elements with the color involved in the comparison as having "high" color degree and marks the color as having "high" degree.

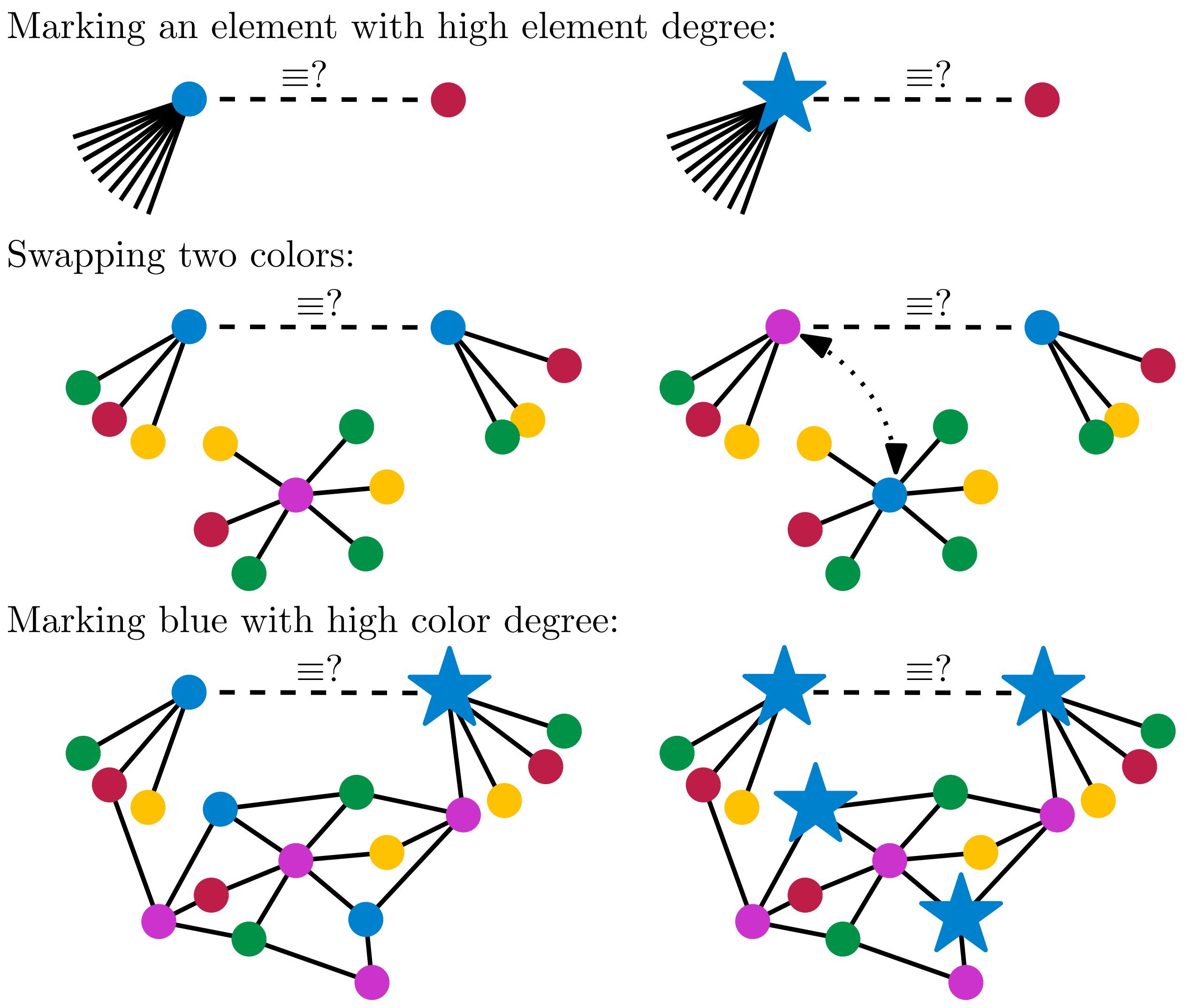

Marking an element with high element degree:
≡?
≡?
Swapping two colors:
≡?
≡?
Marking blue with high color degree:
≡?
≡?

Figure 3.4: Three cases of how the adversary works to mark vertices and swap colors. The dashed line indicates the two elements being compared. Marked vertices are denoted with stars.

- At this point, either both elements are marked and the adversary answers the algorithms comparison based on their color, or one of the elements is unmarked and they have different colors, so the adversary can reply “not equal” to the algorithm’s comparison.

At all times, the vertices that contain unmarked elements all have weight one, because the adversary only answers equivalent for comparisons involving two marked vertices. When a color class is marked, all elements in that color class are marked as having “high” color degree. A few of the cases the adversary goes through are depicted in Figure 3.4.

Lemma 3.3. *If $n/8$ elements are marked during the execution of an algorithm, then $\Omega(n^2/f)$ comparisons were performed.*

Proof. There are three types of marked vertices: those with "high" element degree marks, those with "high" color degree marks, and those with both marks.

The color classes must have been marked as having "high" degree when a comparison was being performed between two elements of that color class and there were no unmarked color candidates to swap colors with. Because one of the elements in the comparison had degree less than $n/4f$, only a quarter of the elements have a color class it cannot be swapped with. So if there were at least $n - n/4$ unmarked elements in total, then the elements in the newly marked color class must have been in a comparison $n/2$ times.

The "high" element degree elements were involved in at least $n/4f$ comparisons each. So if i color classes were marked and j elements were only marked with "high" element degree, then the marked elements must have been a part of a test at least $ni/2 + nj/4f \geq (i + j/f)n/4$ times. Once $fi + j \geq n/8$, then at least $n^2/64f$ equivalence tests were performed. □

Theorem 3.5. *If every equivalence class has the same size f, then sorting requires at least $\Omega(n^2/f)$ equivalence comparisons.*

Proof. When an algorithm finishes sorting, each vertex will have weight f and so the elements must all be marked. Thus, by Lemma 3.3, at least $\Omega(n^2/f)$ comparisons must have been performed. □

We also have the following lower bound as well.

Theorem 3.6. *Finding an element in the smallest equivalence class, whose size is ℓ, requires at least $\Omega(n^2/\ell)$ equivalence comparisons.*

Proof. We use an adversary argument similar to the previous one, but we start with ℓ vertices colored a special *smallest class color (scc)* and seperate the remaining $n - \ell$ vertices into $\lfloor (n-\ell)/(\ell+1) \rfloor$ color classes of size $\frac{n}{\lfloor (n-\ell)/(\ell+1) \rfloor}$ or $\frac{n}{\lfloor (n-\ell)/(\ell+1) \rfloor} + 1$.

There are two changes to the previous adversary responses. First, the degree requirement for having "high" degree is now $n/4\ell$. Second, if an scc element is about to be marked as having "high" degree, we attempt to swap its color with any valid unmarked vertex. Otherwise, we proceed exactly as before.

If an algorithm attempts to identify an element as belonging to the smallest equivalence class, no scc elements are marked, and there have been fewer than $n/8$ elements marked, then the identified element must be able to be swapped with a different color and the algorithm made a mistake. Therefore, to derive a lower bound for the total number of comparisons, it suffices to derive a lower bound for the number of equivalence tests until an scc element is marked.

The scc color class cannot be marked as having "high" color degree until at least one scc element has high element degree. However, as long as fewer than $n/8$ elements are marked, we will never mark an scc element with "high" degree. So at least $n/8$ elements need to be marked as having "high" element degree or "high" color degree and, by the same type of counting as in Lemma 3.3, $\Omega(n^2/\ell)$ equivalence tests are needed. □

3.4 Sorting Distributions

In this subsection, we study a version of the equivalence class sorting problem where we are given a distribution, D, on a countable set, S, and we wish to enumerate the set in order of most likely to least likely, $s_0, s_1, s_2, \ldots$. For example, consider the following distributions:

- Uniform: In this case, D is a distribution on k equivalence classes, with each equivalence class being equally likely for every element of S.

- Geometric: Here, D is a distribution such that the ith most probable equivalence class has probability $p^i(1-p)$. Each element flips a biased coin where heads occurs with probability p until it comes up tails. Then that element is in equivalence class i if it flipped i heads.

- Poisson: In this case, D is model of the number of times an event occurs in an interval of time, with an expected number of events determined by a parameter λ. Equivalence class i is defined to be all the samples that have the same number of events occurring, where the probability of i events occurring is

$$\frac{\lambda^i e^{-\lambda}}{i!} \ .$$

- Zeta: This distribution, D, is related to Zipf's law, and models when the sizes of the equivalence classes follows a power law, based on a parameter, $s > 1$, which is common in many real-world scenarios, such as the frequency of words in natural language documents. With respect to equivalence classes, the ith equivalence class has probability

$$\frac{i^{-s}}{\zeta(s)},$$

where $\zeta(s)$ is Riemann zeta function (which normalizes the probabilities to sum to 1).

So as to number equivalence classes from most likely to least likely, as $i = 0, 1, \ldots$, define $D_{\mathbb{N}}$ to be a distribution on the natural numbers such that

$$\Pr_{x \sim D_{\mathbb{N}}} [x = i] = \Pr_{y \sim D} [y = s_i] \, .$$

Furthermore, so as to "cut off" this distribution at n, define $D_{\mathbb{N}}(n)$ to be a distribution on the natural numbers less than or equal to n such that, for $0 \leq i < n$,

$$\Pr_{x \sim D_{\mathbb{N}}(n)} [x = i] = \Pr_{y \sim D_{\mathbb{N}}} [y = i]$$

and

$$\Pr_{x \sim D_{\mathbb{N}}(n)} [x = n] = \Pr_{y \sim D_{\mathbb{N}}} [y \geq n] .$$

That is, we are "piling up" the tail of the $D_{\mathbb{N}}$ distribution on n.

The following theorem shows that we can use $D_{\mathbb{N}}(n)$ to bound the number of comparisons in an ECS algorithm when the equivalence classes are drawn from D. In particular, we focus here on an algorithm by Jayapaul *et al.* [56] for equivalence class sorting. This algorithm involves a round-robin testing regiment, such that each element, x, initiates a comparison with the next element, y, with an unknown relationship to x, until all equivalence classes are known.

Theorem 3.7. *Given a distribution, D, on a set of equivalence classes, then n elements who have corresponding equivalence class independently drawn from D can be equivalence class sorted using a total number of comparisons stochastically dominated by twice the sum of n draws from the distribution $D_{\mathbb{N}}(n)$.*

Proof. Let V_i denote the random variable that is equal to the natural number corresponding to the equivalence class of element i in $D_{\mathbb{N}}(n)$. We denote the number of elements in equivalence class i as Y_i. Let us denote the number of equivalence tests performed by the algorithm by Jayapaul *et al.* [56] using the random variable, R.

By a lemma from [56], for any pair of equivalence classes, i and j, the round-robin ECS algorithm performs at most $2\min(Y_i, Y_j)$ equivalence tests in total. Thus, the total number of equivalence tests in our distribution-based analysis is upper bounded by

$$\begin{aligned}
R &\leq \sum_{i=0}^{\infty}\sum_{j=0}^{i-1} 2\min(Y_i, Y_j) \\
&= 2\sum_{i=0}^{n}\sum_{j=0}^{i-1} \min(Y_i, Y_j) + 2\sum_{i=n+1}^{\infty}\sum_{j=0}^{i-1} \min(Y_i, Y_j) \\
&\leq 2\sum_{i=0}^{n}\sum_{j=0}^{i-1} Y_i + 2\sum_{i=n+1}^{\infty} nY_i \\
&\leq 2\left(\sum_{i=0}^{n} iY_i + \sum_{i=n+1}^{\infty} nY_i\right) = 2\sum_{i=1}^{n} V_i
\end{aligned}$$

The second line in the above simplification is a simple separation of the double summation and the third line follows because $\sum_{j=0}^{i-1}\min(Y_i, Y_j)$ is zero if Y_i is zero and at most n, otherwise. So the total number of comparisons in the algorithm is bounded by twice the sum of n draws from $D_{\mathbb{N}}(n)$. $\square$

Given this theorem, we can apply it to a number of distributions to show that the total number of comparisons performed is linear with high probability.

Theorem 3.8. *If D is a discrete uniform, a geometric, or a Poisson distribution on a set equivalence classes, then it is possible to equivalence class sort using linear total number of comparisons with exponentially high probability.*

Proof. The sum of n draws from $D_{\mathbb{N}}(n)$ is stochastically dominated by the sum of n draws from $D_{\mathbb{N}}$. Let us consider each distribution in turn.

- Uniform: The sum of n draws from a discrete uniform distribution is bounded by n times the maximum value.

- Geometric: Let p be the parameter of a geometric distribution and let $X = \sum_{i=0}^{n-1} X_i$ where the X_i are drawn from $Geom(p)$, which is, of course, related to the Binomial distribution, $Bin(n,p)$, where one flips n coins with probability p and records the number of "heads." Then, by a Chernoff bound for the geometric distribution (e.g., see [67]),

$$\begin{aligned}
\Pr[X - (1/p)n > k] &= \Pr[Bin(k + (1/p)n, p) < n] \\
&\leq e^{-2\frac{(pk+n-n)^2}{k+(1/p)n}} \\
\Pr[X > (2/p)n] &\leq e^{-np}
\end{aligned}$$

- Poisson: Let λ be the parameter of a Poisson distribution and let $Y = \sum_{i=0}^{n-1} Y_i$ where the Y_i are drawn from $Poisson(\lambda)$. Then, by a Chernoff bound for the Poisson distribution (e.g., see [67]),

$$\begin{aligned}
\Pr[Y > (\lambda(e-1)+1)n] &= \Pr[e^Y > e^{(\lambda(e-1)+1)n}] \\
&\leq \frac{(E[e^{Y_i}])^n}{e^{(\lambda(e-1)+1)n}} \\
&= \frac{e^{\lambda(e-1)n}}{e^{(\lambda(e-1)+1)n}} = e^{-n}
\end{aligned}$$

So, in each case with exponentially high probability, the sum of n draws from the distribution is $O(n)$ and the round-robin algorithm does $O(n)$ total equivalence tests. □

We next address the zeta distribution.

Theorem 3.9. *Given a zeta distribution with parameter $s > 2$, n elements who have corresponding equivalence class independently drawn from the zeta distribution can be equivalence class sorted in $O(n)$ work in expectation.*

Proof. When $s > 2$, the mean of the zeta distribution is

$$\frac{\zeta(s-1)}{\zeta(s)},$$

which is a constant. So the sum of n draws from the distribution is expected to be linear. Therefore, the expected total number of comparisons in the round-robin algorithm is linear. $\square$

Unfortunately, for zeta distributions it is not immediately clear if it is possible to improve the above theorem so that total number of comparisons is shown to be linear when $2 \geq s > 1$ or obtain high probability bounds on these bounds. This uncertainty motivates us to look experimentally at how different values of s cause the runtime to behave. Likewise, our high-probability bounds on the total number of comparisons in the round-robin algorithm for the other distributions invites experimental analysis as well.

3.5 Experiments

In this section, we report on experimental validations of the theorems from the previous section and investigations of the behavior of running the round-robin algorithm on the zeta distribution. For the uniform, geometric, and Poisson distributions, we ran ten tests on sizes of $10,000$ to $200,000$ elements incrementing in steps of $10,000$. For the zeta distribution, because setting $s < 2$ seems to lead to a superlinear number of comparisons, we reduced the test sizes by a factor of 10 and ran ten tests each on sizes from $1,000$ to $20,000$ in increments of $1,000$. For each distribution we used the following parameter settings for various experiments:

Uniform:	$k = 10, 25, 100$
Geometric:	$p = \frac{1}{2}, \frac{1}{10}, \frac{1}{50}$
Poisson:	$\lambda = 1, 5, 25$
Zeta:	$s = 1.1, 1.5, 2, 2.5$

The results of these tests are plotted in Figure 3.5. Best fit lines were fitted whenever we have theorems stating that there will be a linear number of comparisons with high probability or in expectation (i.e., everything except for zeta with $s < 2$). We include extra plots of the zeta distribution tests with the $s = 1.1$ data and the $s = 1.1, 1.5$ data removed to better see the other data sets.

We can see from the data that the number of comparisons for the uniform, geometric, and Poisson distributions are so tightly concentrated around the best fit line that only one data point is visible. Contrariwise, the data points for the zeta distributions do not cluster nearly as nicely. Even when we have linear expected comparisons with $s = 2$, the data points vary by as much as 10%.

3.6 Conclusion

In this chapter we have studied the equivalence class sorting problem, from a parallel perspective, giving several new algorithms, as well as new lower bounds and distribution-based analysis. We leave as open problems the following interesting questions:

- Is it possible to find all equivalance classes in the ER version of the ECS problem in $O(k)$ parallel rounds, for $k \geq 3$, where k is the number of equivalence classes? Note that the answer is "yes" for $k = 2$, as it follows from previous results for the parallel fault diagnosis problem [11–13].

- Is it possible to bound the number of comparisons away from $O(n^2)$ for the zeta distribution when $s < 2$ even just in expectation?

- Is it possible to prove a high-probability concentration bound for the zeta distribution, similar to the concentration bounds we proved for other distributions?

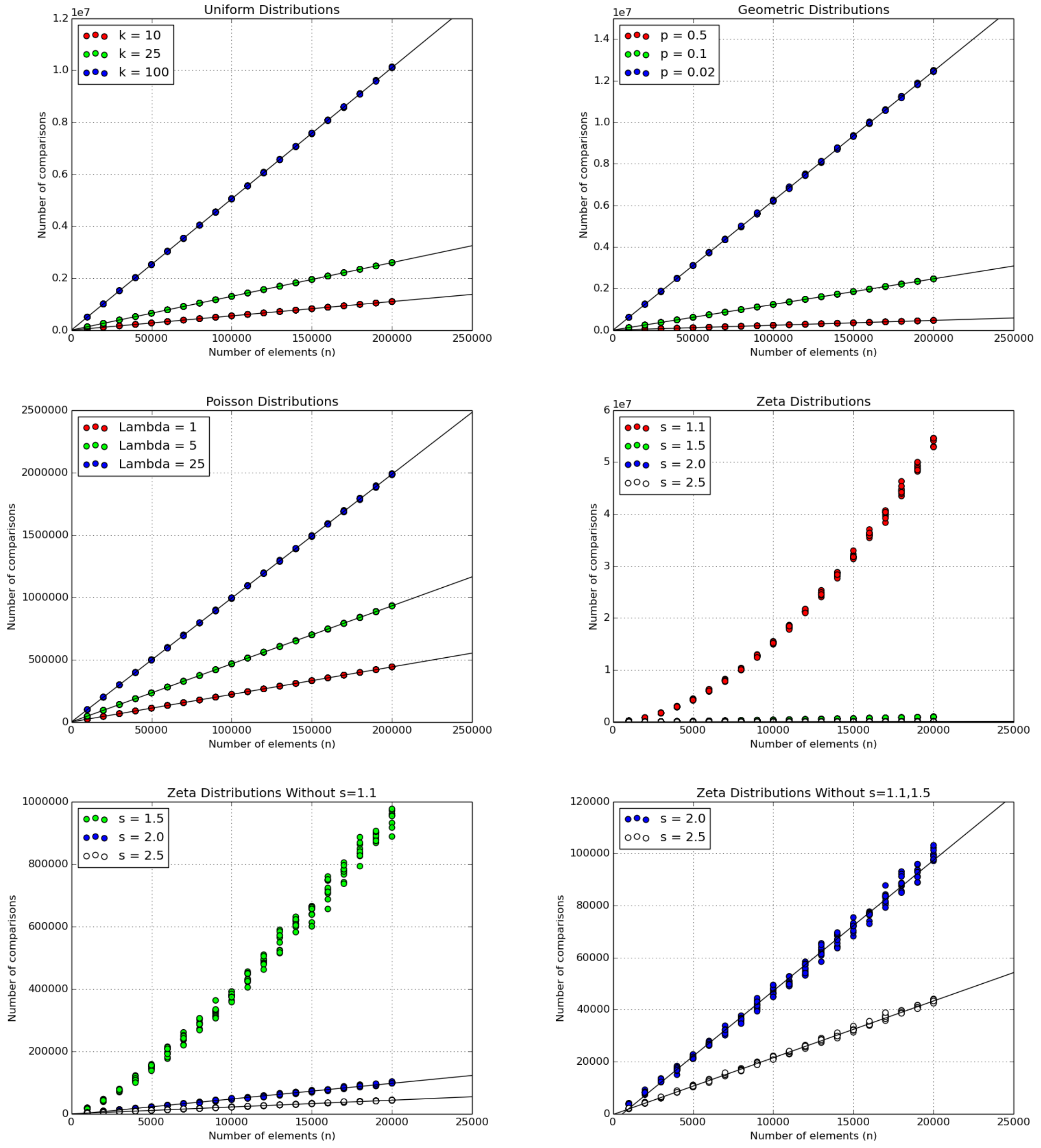


Figure 3.5: The results of the experiments are plotted and best fit lines are placed when we have a linear number of comparisons with high probability or in expectation.

Chapter 4

Optimally Sorting Evolving Data

4.1 Introduction

In the classic version of the sorting problem, we are given a set, S, of n comparable items coming from a fixed total order and asked to compute a permutation that places the items from S into non-decreasing order, and it is well-known that this can be done using $O(n \log n)$ comparisons, which is asymptotically optimal (e.g., see [30, 45, 58]). There are a number of interesting applications where this classic version of the sorting problem doesn't apply, however.

For instance, consider the problem of maintaining a ranking of a set of sports teams based on the results of head-to-head matches. A typical approach to this sorting problem is to assume there is a fixed underlying total order for the teams, but that the outcomes of head-to-head matches (i.e., comparisons) are "noisy" in some way. In this formulation, the ranking problem becomes a one-shot optimization problem of finding the most-likely fixed total order given the outcomes of the matches (e.g., see [20, 40, 47, 50, 65]). In this chapter, we study an alternative, complementary motivating scenario, however, where instead of there being a fixed total order and noisy comparisons we have a scenario where comparisons are accurate but the underlying total order is evolving. This scenario, for instance, captures the real-world phenomenon where sports teams make mid-season changes to their player rosters and/or coaching staffs that result in improved or degraded competitiveness relative to other teams. That is, we are interested in the sorting problem for *evolving data*.

4.1.1 Related Prior Work for Evolving Data

Anagnostopoulos *et al.* [7] introduce the *evolving data* framework, where an input data set is changing while an algorithm is processing it. In this framework, instead of an algorithm taking a single input and producing a single output, an algorithm attempts to maintain an output close to the correct ouptut for the current state of the data, repeatedly updating its best estimate of the correct output over time. For instance, Anagnostopoulos *et al.* [7] mention the motivation of maintaining an Internet ranking website that displays an ordering of entities, such as political candidates, movies, or vacation spots, based on evolving preferences.

Researchers have subsequentially studied other interesting problems in the evolving data framework, including the work of Kanade *et al.* [57] on stable matching with evolving preferences, the work of Huang *et al.* [52] on selecting top-k elements with evolving rankings, the work of Zhang and Li [84] on shortest paths in evolving graphs, the work of Anagnostopoulos *et al.* [8] on st-connectivity and minimum spanning trees in evolving graphs, and the work of Bahmani *et al.* [10] on PageRank in evolving graphs. In each case, the goal is to maintain an output close to the correct one even as the underlying data is changing at a rate commensurate to the speed of the algorithm. By way of analogy, classical algorithms are to evolving-data algorithms as throwing is to juggling.

4.1.2 Problem Formulation for Sorting Evolving Data

With respect to the sorting problem for evolving data, following the formulation of Anagnostopoulos *et al.* [7], we assume that we have a set, S, of n distinct items that are properly ordered according to a total order relation, "$<$". In any given time step, we are allowed to compare any pair of items, x and y, in S according to the "$<$" relation and we learn the correct outcome of this comparison. After we perform such a comparison, α pairs of items that are currently consecutive according to the "$<$" relation are chosen uniformly at random and their relative order is swapped. As in previous work [7], we focus on the case where $\alpha = 1$, but one can also consider versions of the problem where the ratio between comparisons and random consecutive swaps is something other than one-to-one. Still, this simplified version with a one-to-one ratio already raises some interesting questions.

Since it is impossible in this scenario to maintain a list that always reflects a strict ordering according to the "$<$" relation, our goal is to maintain a list with small *Kendall tau* distance, which counts the number of inversions, relative to the correct order.[1] Anagnostopoulos *et al.* [7] show that, for $\alpha = 1$, the Kendall tau distance between the maintained list and the underlying total order is $\Omega(n)$ in both expectation and with high probability. They also show how to maintain this distance to be $O(n \log \log n)$, with high probability, by performing a multiplexed batch of quicksort algorithms on small overlapping intervals of the list.

4.1.3 Our Contributions

The main contribution of the present chapter is to provide algorithms that maintain an asymptotically optimal Kendall tau distance, with high probability, for sorting evolving data. Moreover, we show how to achieve such a result using a simple repeated insertion-sort

[1]Recall that an *inversion* is a pair of items u and v such that u comes before v in a list but $u > v$. An *inversion* in a permutation π is a pair of elements $x \neq y$ with $x < y$ and $\pi(x) > \pi(y)$.

algorithm. This algorithm repeatedly makes in-place insertion-sort passes (e.g., see [30, 45]) over the list, l_t, maintained by our algorithm at each step t. Each such pass moves the item at position j to an earlier position in the list so long as it is bigger than its predecessor in the list. With each comparison done by this repeated insertion-sort algorithm, we assume that a consecutive pair of elements in the underlying ordered list, l'_t, are chosen uniformly at random and swapped. In spite of the uncertainty involved in sorting evolving data in this way, we prove the following theorem, which is the main result of this chapter.

Theorem 4.1. *When running the repeated insertion-sort algorithm, for every step $t = \Omega(n^2)$, the Kendall tau distance between the maintained list, l_t, and the underlying ordered list, l'_t, is $O(n)$ with exponentially high probability.*

That is, after an initialization period of $\Theta(n^2)$ steps, the repeated insertion-sort algorithm converges to a steady state having an asymptotically optimal Kendall tau distance between the maintained list and the underlying total order, with exponentially high probability. We also show how to reduce this initialization period to be $\Theta(n \log n)$ steps, with high probability, by first performing a quicksort algorithm and then following that with the repeated insertion-sort algorithm.

Intuitively, our proof of Theorem 4.1 relies on two ideas: the adaptivity of insertion sort and that, as time progresses, a constant fraction of the random swaps fix inversions. Ignoring the random swaps for now, when there are k inversions, a complete execution of insertion sort performs roughly $k + n$ comparisons and fixes the k inversions (e.g., see [30, 45]). If an ϵ fraction of the random swaps fix inversions, then during insertion sort $\epsilon(k + n)$ inversions are fixed by the random swaps and $(1 - \epsilon)(k + n)$ are introduced. Naively the total change in the number of inversions is then $(1 - 2\epsilon)(k + n) - k$ and when $k > \frac{1-2\epsilon}{2\epsilon}n$, the number of inversions decreases. So the number of inversions will decrease until $k = O(n)$.

This simplistic intuition ignores two competing forces involved in the heavy interplay between the random swaps and insertion sort's runtime, however, in the evolving data model. First, random swaps can cause an insertion-sort pass to end too early, thereby causing insertion sort to fix fewer inversions than normal. Second, as insertion sort progresses, it decreases the chance for a random swap to fix an inversion. Analyzing these two interactions comprises the majority of our proof of Theorem 4.1.

In Section 4.3, we present a complete proof of Theorem 4.1. The most difficult component of Theorem 4.1's proof is Lemma 4.5, which lower bounds the runtime of insertion sort in the evolving data model. The proof of Lemma 4.5 is presented separately in Section 4.4.

4.2 Preliminaries

The sorting algorithm we analyze in this chapter for the evolving data model is the repeated insertion-sort algorithm whose pseudocode is shown in Algorithm 4.

Algorithm 4 Repeated insertion sort pseudocode

```
function REPEATED_INSERTION_SORT($l$)
    while true do
        for $i \leftarrow 1$ to $n - 1$ do
            $j \leftarrow i$
            while $j > 0$ and $l[j] < l[j-1]$ do
                swap $l[j]$ and $l[j-1]$
                $j \leftarrow j - 1$
            end while
        end for
    end while
end function
```

Formally, at time t, we denote the sorting algorithms's list as l_t and we denote the underlying total order as l'_t. Together these two lists define a permutation, σ_t, of the indices, where $\sigma_t(x) = y$ if the element at index x in l_t is at position y in l'_t. We define the *simulated final state at time t* to be the state of l obtained by freezing the current underlying total order, l'_t, (i.e., no more random swaps) and simulating the rest of the current round of insertion sort (we refer to each iteration of the **while-true** loop in Algorithm 4 as a *round*). We then define a *frozen-state* permutation, $\hat{\sigma}_t$, where $\hat{\sigma}_t(x) = y$ if the element at index x in the simulated final state at time t as at index y in l'_t.

Let us denote the number of inversions at time t, in σ_t, with I_t. Throughout the chapter, we may choose to drop time subscripts if our meaning is clear. The Kendall tau distance between two permutations π_1 and π_2 is the number of pairs of elements $x \neq y$ such that $\pi_1(x) < \pi_1(y)$ and $\pi_2(x) > \pi_2(y)$. That is, the Kendall tau distance between l_t and l'_t is equal to I_t, the number of inversions in σ_t. Figure 4.1 shows the state of l, l', I, and σ for two steps of an insertion sort (but not in the same round).

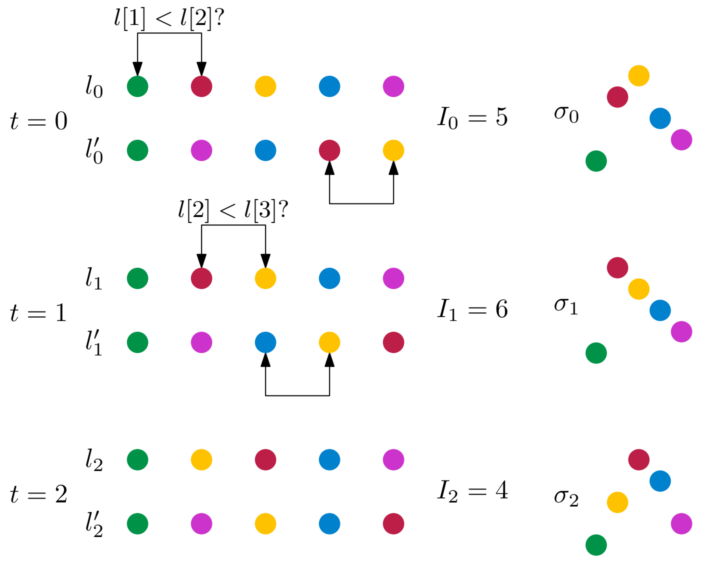


Figure 4.1: Examples of l, l', I, and σ over two steps of an algorithm. In the first step the green and red elements are compared in l and the red and yellow elements are swapped in l'. In the second step the red and yellow elements are compared and swapped in l and the blue and yellow elements are swapped in l'.

As the inner **while**-loop of Algorithm 4 executes, we can view l as being divided into three sets: the set containing just the *active* element, $l[j]$ (which we view as moving to the left, starting from position i, as it is participating in comparisons and swaps), the *semi-sorted* portion, $l[0 : i]$, not including $l[j]$, and the *unsorted* portion, $l[i + 1 : n - 1]$. Note that if no random adjacent swaps were occurring in l' (that is, if we were executing insertion-sort in the classical algorithmic model), then the semi-sorted portion would be in sorted order.

Given a list, L, of m numbers with no two equal numbers, the *Cartesian tree* [79] of L is a binary rooted tree on the numbers where the root is the minimum element $L[k]$, the left subtree of the root is the Cartesian tree of $L[0 : k-1]$, and the right subtree of the root is the Cartesian tree of $L[k+1 : m]$. In our analysis, we will primarily consider the Cartesian tree of the simulated final state at time t where $L[k] = \hat{\sigma}_t(k)$ in the frozen-state permutation $\hat{\sigma}_t$. We also choose to include two additional elements, $L[-1] = -1$ and $L[n] = n$, for boundary cases. Figure 4.2 shows an example Cartesian tree we might consider. The Cartesian trees we consider are only for the sake of analysis. They are not explicitly constructed.

We call the path from the root to the rightmost leaf of the Cartesian tree the (right-to-left) minima path as the elements on this path are the right-to-left minima in the list. The minima path is highlighted in Figure 4.2. For a minimum, $l[k]$, denote with $M(k)$ the index of the element in the left subtree of $l[k]$ that maximizes $\hat{\sigma}(k)$, i.e., the index of the largest element in the left subtree.

We use the phrase *with high probability* to indicate when an event occurs with probability that tends towards 1 as $n \to \infty$. When an event occurs with probability of the form $1 - e^{-poly(n)}$,

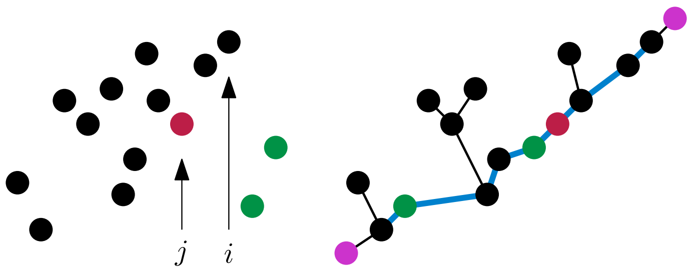


Figure 4.2: On the left we have a representation of σ, a dot for each element x is drawn at the coordinate (a, b) where $x = l[a] = l'[b]$. On the right the elements have been moved to their position in $\hat{\sigma}$ and the corresponding Cartesian tree is superimposed. The active element of insertion sort at the current moment is highlighted in red, the elements that haven't been seen by the algorithm are highlighted in green, the added elements are highlighted in pink, and the minima path is highlighted in blue.

we say it occurs with *exponentially high probability.* During our analysis, we will make use of the following facts.

Lemma 4.1 (Poisson approximation (Corollary 5.9 in [68])). *Let $X_1^{(m)}, \ldots, X_n^{(m)}$ be the number of balls in each bin when m balls are thrown uniformly at random into n bins. Let $Y_1^{(m)}, \ldots, Y_n^{(m)}$ be independent Poisson random variables with $\lambda = m/n$. Then for any event $\varepsilon(x_1, \ldots, x_n)$:*

$$\Pr\left[\varepsilon\left(X_1^{(m)}, \ldots, X_n^{(m)}\right)\right] \leq e\sqrt{m}\Pr\left[\varepsilon\left(Y_1^{(m)}, \ldots, Y_n^{(m)}\right)\right].$$

Lemma 4.2 (Hoeffding's inequality (Theorem 2 in [51])). *If $X_1, \ldots, X_n$ are independent random variables and $a_k \leq X_k \leq b_k$ for $k = 1, \ldots, n$, then for $t > 0$:*

$$\Pr\left[\sum_k X_k - E\left[\sum_k X_k\right] \geq tn\right] \leq e^{-2n^2t^2/\left(\sum_k (b_k - a_k)^2\right)}.$$

4.3 Sorting Evolving Data with Repeated Insertion Sort

Let us begin with some simple bounds with respect to a single round of insertion sort.

Lemma 4.3. *If a round of insertion sort starts at time t_s and finishes at time t_e, then*

1. *$t_e - t_s = F + n - 1$, where F is the number of inversions fixed (at the time of a comparison in the inner* **while***-loop) by this round of insertion sort.*

2. *$t_e - t_s < n^2/2$*

3. *for any $t_s \leq t \leq t_e$, $I_t - I_{t_s} < n$.*

Proof. (1): For each iteration of the outer **for**-loop, each comparison in the inner **while**-loop either fixes an inversion (at the time of that comparison) or fails to fix an inversion and completes the inner **while**-loop. Note that this "failed" comparison may not have compared elements of l, but may have short circuited due to $j \leq 0$. Nevertheless, every comparison that doesn't fail fixes an inversion (at the time of that comparison); hence, each non-failing comparison is counted in F.

(2): In any round, there are at most $n(n-1)/2$ comparisons, by the formulations of the outer **for**-loop and inner **while**-loop.

(3): At time t, the round of insertion sort will have executed $t - t_s$ steps. Of those steps, at least $t - t_s - (n-1)$ comparisons resulted in a swap that removed an inversion and at most $n-1$ comparisons did not result in a change to l. The random swaps occurring during these comparisons introduced at most $t - t_s$ inversions. So $I_t - I_{t_s} \leq t - t_s - \big(t - t_s - (n-1)\big) = n - 1$. □

We next show that, while a round of insertion sort executes, a constant fraction of the random swaps are actually fixing inversions.

Lemma 4.4. *There exists a constant, $0 < \epsilon < 1$, such that, for a round of insertion sort that takes time t^*, at least ϵt^* of the random adjacent swaps in l' decrease I during the round, with exponentially high probability.*

Proof. We call a random adjacent swap that decreases the number of inversions, I, during the insertion-sort round a *good swap.*

Break the time interval for this round of insertion sort into epochs, each of size between $n/32$ and $n/16$ (this is possible because $t^* \geq n-1$, by Lemma 4.3) and let t_k be the start of epoch k. Denote the length of epoch k by $t_k^* = t_{k+1} - t_k$. Given the values of i and j at t_k, only the elements in the ranges $l[j - n/16, j]$ and $l[i - n/16, i + n/16]$ will be involved in insertion sort comparisons during epoch k. This set of potentially compared elements has size at most $3n/16$.

Consider the set of adjacent disjoint 4-tuples in l', $l'[4a], l'[4a+1], l'[4a+2], l'[4a+3]$ for $a = 0, 1, \ldots, n/4$. There are $n/4$ of these tuples and so there are at least $n/4 - 3n/16 = n/16$ tuples whose elements cannot be involved in comparisons during a given epoch. Call such a tuple of elements an *untouchable tuple*.

We now examine just the swaps during one specific epoch. Let X_i be the number of random adjacent swaps that swap $l'[i]$ with $l'[i+1]$ for $i = 0, 1, \ldots, n-1$. Let Y_i be independent identically distributed Poisson random variables with parameter $\lambda = \frac{t_k^*}{n-1}$ for $i = 0, 1, \ldots, n-1$. Note that $1/32 \leq \lambda \leq \frac{n}{16(n-1)} \leq 1/15$ for large enough n. Let $f(z_1, z_2, \ldots, z_n)$ be the function that counts how many $a = 0, 1, \ldots, n/4$ there are such that the tuple $l'[4a], l'[4a+1], l'[4a+2], l'[4a+3]$ is untouchable and $z_{4a} = 0$, $z_{4a+1} = 2$, and $z_{4a+2} = 0$.

By the Poisson approximation, Lemma 4.1, for any $\delta > 0$,

$$\Pr\big[f(X_1, X_2, \ldots, X_{n-1}) \leq \delta n\big] \leq e\sqrt{n/16}\,\Pr\big[f(Y_1, Y_2, \ldots, Y_{n-1}) \leq \delta n\big].$$

As previously stated, there are at least $n/16$ untouchable tuples. Because the Y_i are independent, for an untouchable 4-tuple $l'[4a], l'[4a+1], l'[4a+2], l'[4a+3]$,

$$\begin{aligned}\Pr\big[Y_{4a} = 0, Y_{4a+1} = 2, Y_{4a+2} = 0\big] &= \frac{e^{-3\lambda}\lambda^2}{0!2!0!}\\ &\geq \frac{e^{-3/15}\,(1/32)^2}{2}\\ &\geq \frac{3}{10{,}000}\end{aligned}$$

$f(Y_1, Y_2, \ldots, Y_{n-1})$ is the sum of at least $n/16$ independent indicator random variables that each have probability at least $3/10{,}000$ of being 1. Thus $E[f(Y_1, Y_2, \ldots, Y_{n-1})] \geq \frac{3n}{160{,}000}$. Therefore, by a Chernoff bound from [68]:

$$\Pr\left[f\left(Y_1, Y_2, \ldots, Y_{n-1}\right) \le \left(1 - \frac{1}{2}\right) \frac{3n}{160,000}\right] \le e^{-\Omega(n)}$$
$$\Pr\left[f\left(X_1, X_2, \ldots, X_{n-1}\right) \le \left(1 - \frac{1}{2}\right) \frac{3n}{160,000}\right] \le \frac{e\sqrt{n/16}}{e^{\Omega(n)}} \le e^{-\,poly(n)}$$

Therefore, within each epoch of the insertion sort round there are at least $\frac{3}{320{,}000}n$ untouchable tuples where the middle pair of indices are swapped twice and the other two pairs are not swapped, with exponentially high probability. In each of these tuples one of the two swaps must have been a good swap.

So we can conclude that for each epoch, with exponentially high probability, there are $\frac{3}{320{,}000}n$ good swaps. Because there are at least $\frac{t^*}{n/16}$ epochs, setting $\epsilon = \frac{3}{20{,}000}$ implies there are at least ϵt^* good swaps during the entire insertion sort round, with exponentially high probability. □

We next give a lower bound with respect to a single round of repeated insertion sort.

Lemma 4.5. *If a round of insertion sort starts at time t_s with $I_{t_s} \geq (12c^2 + 2c)n$ and finishes at time t_e, then, with exponentially high probability, $t_e - t_s \geq cn$, i.e., the insertion sort round takes at least cn steps.*

Proof. See Section 4.4. □

4.3.1 Proof of Theorem 4.1

Armed with the above lemmas (albeit postponing the proof of Lemma 4.5), let us prove our main theorem.

Theorem 4.1. *There exists a constant, $0 < \epsilon < 1$, such that, when running the repeated* *insertion-sort algorithm, for every step $t > (1 + 1/\epsilon)n^2$, the Kendall tau distance between the maintained list, l_t, and the underlying ordered list, l'_t, is $O(n)$, with exponentially high probability.*

Proof. By Lemma 4.4, there exists a constant $0 < \epsilon < 1$ such that at least an ϵ fraction of all of the random swaps during a round of insertion sort fix inversions. Consider an epoch of the last $(1 + 1/\epsilon)n^2$ steps of the repeated insertion-sort algorithm, that is, from time $t' = t - (1 + 1/\epsilon)n^2$ to t. During this epoch, some number, $m \geq 1$, of complete rounds of insertion sort are performed from start to end (by Lemma 4.3). Denote with t_k the time at which insertion-sort round k ends (and round $k + 1$ begins), and let t_m denote the end time of the final complete round, during this epoch. By construction, observe that $t' \leq t_0$ and $t_m \leq t$. Furthermore, because the insertion-sort rounds running before t_0 and after t_m take fewer than $n^2/2$ steps (by Lemma 4.3), $t_m - t_0 \geq n^2/\epsilon$.

The remainder of the proof consists of two parts. In the first part, we show that for some complete round of insertion sort ending at time $t_k \leq t$, I_{t_k} is $O(n)$, with exponentially high probability. In the second part, we show that once we achieve I_{t_k} being $O(n)$, for $t_k \leq t$, then I_t is $O(n)$, with exponentially high probability.

For the first part, suppose, for the sake of a contradiction, $I_{t_k} > \left(12(\frac{1}{\epsilon})^2 + \frac{2}{\epsilon}\right)n$, for all $0 \leq k \leq m$. Then, by a union bound over the polynomial number of rounds, Lemma 4.5 applies to every such round of insertion sort. So, with exponentially high probability, each round takes at least n/ϵ steps. Moreover, by Lemma 4.4, with exponential probability, an ϵ fraction of the random swaps from t_m to t_0 will decrease the number of inversions. That is, these random swaps increase the number of inversions by at most

$$(1 - \epsilon)(t_m - t_0) - \epsilon(t_m - t_0) = (1 - 2\epsilon)(t_m - t_0),$$

with exponentially high probability. Furthermore, by Lemma 4.3, at least a $\frac{(1/\epsilon)-1}{1/\epsilon} = 1 - \epsilon$

fraction of the insertion-sort steps fix inversions (at the time of a comparison). Therefore, with exponentially high probability, we have the following:

$$\begin{aligned} I_{t_m} &\leq I_{t_0} - (1-\epsilon)(t_m - t_0) + (1-2\epsilon)(t_m - t_0) \\ &= I_{t_0} - \epsilon(t_m - t_0) \\ &\leq I_{t_0} - n^2. \end{aligned}$$

But, since $I_{t_0} < n^2$, the above bound implies that $I_{t_m} < 0$, which is a contradiction. Therefore, with exponentially high probability, there is a $k \leq m$ such that $I_{t_k} \leq (12(\frac{1}{\epsilon})^2 + \frac{2}{\epsilon})n$.

For the second part, we show that the probability for a round $\ell > k$ to have $I_{t_\ell} > (12(\frac{1}{\epsilon})^2 + \frac{2}{\epsilon} + 1)n$ is exponentially small, by considering two cases (and their implied union-bound argument):

- If $I_{t_{\ell-1}} \leq (12(\frac{1}{\epsilon})^2 + \frac{2}{\epsilon})n$, then Lemma 4.3 implies $I_{t_\ell} \leq (12(\frac{1}{\epsilon})^2 + \frac{2}{\epsilon} + 1)n$.
- If $(12(\frac{1}{\epsilon})^2 + \frac{2}{\epsilon})n \leq I_{t_{\ell-1}} \leq (12(\frac{1}{\epsilon})^2 + \frac{2}{\epsilon} + 1)n$, then, similar to the argument given above, during a round of insertion sort, ℓ, at least a $1-\epsilon$ fraction of the steps fix an inversion. Also at least an ϵ fraction of the random swaps fix inversions. Thus, with exponentially high probability, the total change in inversions is at most $-\epsilon(t_\ell - t_{\ell-1})$ and $I_{t_\ell} < I_{t_{\ell-1}}$.

Therefore, by a union bound over the polynomial number of insertion-sort rounds, the probability that any $I_{t_\ell} > (12(\frac{1}{\epsilon})^2 + \frac{2}{\epsilon} + 1)n$ for $k < \ell \leq m$ is exponentially small. By Lemma 4.3, $I_t \leq I_{t_m} + n$. So, with exponentially high probability, $I_{t_m} \leq (12(\frac{1}{\epsilon})^2 + \frac{2}{\epsilon} + 1)n = O(n)$ and $I_t = O(n)$, completing the proof. □

4.3.2 Improved Convergence Rate

In this subsection, we provide an algorithm that converges to $O(n)$ inversions more quickly. To achieve the steady state of $O(n)$ inversions, repeated insertion sort performs $\Theta(n^2)$ comparisons. But this running time to reach a steady state is a worst-case based on the fact that the running time of insertion sort is $O(n + I)$, where I is the number of initial inversions in the list, and, in the worst case, I is $\Theta(n^2)$. By simply running a round of quicksort on l first, we can achieve a steady state of $O(n)$ inversions after just $\Theta(n \log n)$ comparisons. See Algorithm 5. That is, we have the following.

Algorithm 5 Quicksort followed by repeated insertion sort pseudocode

```
function QUICK_THEN_INSERTION_SORT($l$)
    quicksort($l$)
    while true do
        for $i \leftarrow 1$ to $n - 1$ do
            $j \leftarrow i$
            while $j > 0$ and $l[j] < l[j-1]$ do
                swap $l[j]$ and $l[j-1]$
                $j \leftarrow j - 1$
            end while
        end for
    end while
end function
```

Theorem 4.2. *When running Algorithm 5, for every $t = \Omega(n \log n)$, I_t is $O(n)$ with high probability.*

Proof. By the results of Anagnostopoulos *et al.* [7], the initial round of quicksort takes $\Theta(n \log n)$ comparisons and afterwards the number of inversions (that is, the Kendall tau distance between the maintained list and the true total order) is $O(n \log n)$, with high probability. Using a nearly identical argument to the proof of Theorem 4.1, and the fact that an insertion-sort round takes $O(I + n)$ time to resolve I inversions, the repeated insertion-sort algorithm will, with high probability, achieve $O(n)$ inversions in an additional $O(n \log n)$ steps. From that point on, it will maintain a Kendall tau distance of $O(n)$, with high probability. □

4.4 Proof of Lemma 4.5

Recall Lemma 4.5, which establishes a lower bound for the running time of an insertion-sort round, given a sufficiently large amount of inversions relative to the underlying total order.

Lemma 4.5. *If a round of insertion sort starts at time t_s with $I_{t_s} \geq (12c^2 + 2c)n$ and finishes at time t_e, then, with exponentially high probability, $t_e - t_s \geq cn$, i.e., the insertion sort round takes at least cn steps.*

The main difficulty in proving Lemma 4.5 is understanding how the adjacent random swaps in l' affect the runtime of the current round of insertion sort on l. Let S_t be the number of steps left to perform in the current round of insertion sort if there were no more random adjacent swaps in l'. In essence, S can be thought of as an estimate of the remaining time in the current insertion sort round. If a new round of insertion sort is started at time t_s, then $S_{t_s-1} = 1$ and $I_{t_s} \leq S_{t_s} \leq I_{t_s} + n - 1$. Each step of an insertion sort round decreases S by one and the following random swap may increase or decrease S by some amount. Figure 4.3 illustrates an example where one random adjacent swap in l' decreases S by a non-constant amount (relative to n).

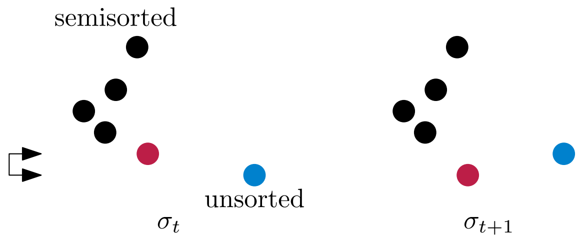


Figure 4.3: An example where swapping the red and blue elements in l' creates multiple blocked inversions between the red element and the black elements.

A random adjacent swap in l' involving two elements in the unsorted portion of l will either increase or decrease S by one depending on if it introduces or removes an inversion. Random adjacent swaps involving elements in the semi-sorted portion have more complex affects on S.

An inversion currently in the list $\big(l[a], l[b]\big)$ will be fixed by insertion sort if $l[a]$ and $l[b]$ will be compared and the two are swapped. Because $a < b$, $l[b]$ must be the active element during this comparison. An inversion $\big(l[a], l[b]\big)$ will not be fixed by insertion sort if $l[b]$ was already inserted into the semi-sorted portion or there is some element $l[c]$ in the semi-sorted portion with $a < c < b$ and $\sigma(c) < \sigma(b)$. We call an inversion with $l[b]$ in the semi-sorted portion a *stuck* inversion and an inversion with a smaller semi-sorted element between the pair a *blocked* inversion. We say an element $l[c]$ in the semi-sorted portion of l *blocks* an inversion $\big(l[a], l[b]\big)$ with $a \leq i$ and $l[b]$ either the active element or in the unsorted portion of l, if $l[c]$ is in the semi-sorted portion of l with $a < c < b$ and $\sigma(c) < \sigma(b)$. Note that there may be multiple elements blocking a particular inversion. Figure 4.4 shows examples of these two types of inversions.

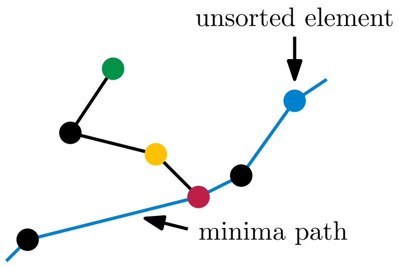


Figure 4.4: In this Cartesian tree, the green-blue pair is a blocked inversion and the green-yellow pair is a stuck inversion. Both pairs of inversions blame the red element.

We denote the number of "bad" inversions at time t that will not be fixed with B_t. That is, B_t is the sum of the blocked and stuck inversions. At the end of an insertion-sort round every inversion present at the start was either fixed by the insertion sort, fixed by a random adjacent swap in l', or is currently stuck. No elements can be blocked at the end of an insertion-sort round, because the semi-sorted portion is the entire list. Stuck inversions are either created by random adjacent swaps in l' or were blocked inversions and insertion sort finished inserting the right element of the pair. Blocked inversions are only introduced by the random adjacent swaps in l'. Thus B_t is unaffected by the steps of insertion sort.

Every inversion present at the start must be fixed by a step of insertion sort, be fixed by a random swap, or it will end up “bad”. Therefore, for any given time, t, by using naive upper bounds based on the facts that every insertion sort step can fix an inversion and every random adjacent swap can remove an inversion, we can immediately derive the following:

Lemma 4.6. *For an insertion sort round that starts at time t_s and ends at time t_e, if $t_s \leq t \leq t_e$, then $S_t \geq I_{t_s} - 2(t - t_s) - B_t$.*

Since, when an insertion sort round finishes, $S_{t_e - 1} = 1$, Lemma 4.6 implies $2(t_e - t_s - 1) + B_{t_e} + 1 \geq I_{t_s}$. If we understand how B changes with each random adjacent swap in l', then we can bound how long insertion sort needs to run for this inequality to be true.

We associate the blocked and stuck inversions with elements that we say are *blamed* for the inversions. A blocked inversion $\big(l[a], l[b]\big)$ blames the element $l[c]$ with $a < c < b$ and minimum $\sigma(c)$. Note than $l[c]$ is on the minima path of the modified Cartesian tree and $l[a]$ is in the left subtree of $l[c]$. A stuck inversion either blames the element on the minima path whose subtree contains both $l[a]$ and $l[b]$ or if they appear in different subtrees, the inversion blames the element $l[c]$ with $a < c < b$ and minimum $\sigma(c)$. Again note that the blamed element is on the minima path and $l[a]$ is in the blamed element's left subtree. The bad inversions in Figure 4.4 blame the red element.

Whether stuck or blocked, every inversion blames an element on the minima path and the left element of the inverted pair appears in that minimum's subtree. If $l[k]$ is on the minima path, $M(k)$ is the index of the element in $l[k]$'s subtree with maximum $\sigma(M(k))$, and an inversion $\big(l[a], l[b]\big)$ has $l[a]$ in $l[k]$'s subtree, then both $l[a]$ and $l[b]$ are in the range $\sigma(k)$ to $\sigma(M(k))$. So we can upper bound B_t by $\sum_{k=0}^{n-1}(\sigma(M(k)) - \sigma(k))^2$, where we extend M to non-minima indices with $M(k) = k$ if k is not the index of a minima in l.

4.4.1 Bounding the Number of Blocked and Stuck Inversions with Counters

For the purposes of bounding B_t, we conceptually associate two counters, $\mathit{Inc}(x)$ and $\mathit{Dec}(x)$, with each element, x. The counters are initialized to zero at the start of an insertion sort round. When an element x is increased by a random swap in l', we increment $\mathit{Inc}(x)$ and when x is decreased by a random swap in l', we increment $\mathit{Dec}(x)$. After the random swap occurs, we may choose to exchange some of the counters between pairs of elements, but we will always maintain the following invariant:

Invariant 1. *For an element, $l[k]$, on the minima path,*

$$\mathit{Inc}\big(l[M(k)]\big) + \mathit{Dec}\big(l[k]\big) \geq \sigma\big(M(k)\big) - \sigma(k).$$

We actually maintain a stronger invariant that implies this invariant. However, Invariant 1 allows us to prove the following Lemma:

Lemma 4.7. *If $\sum_{k=0}^{n-1} \mathit{Inc}\big(l[k]\big)^2 < \kappa$ and $\sum_{k=0}^{n-1} \mathit{Dec}\big(l[k]\big)^2 < \kappa$, then $B_t \leq 4\kappa$.*

Proof.

$$\begin{aligned} B_t &\leq \sum_{k=0}^{n-1} \Big(\sigma\big(M(k)\big) - \sigma(k)\Big)^2 \\ &\leq \sum_{k=0}^{n-1} \Big(\mathit{Inc}\big(M(k)\big) + \mathit{Dec}(k)\Big)^2 \qquad \text{By Invariant 1} \qquad (4.1) \end{aligned}$$

Interpreting Inc and Dec as two n-dimensional vectors, we know their lengths are both less than $\sqrt{\kappa}$. Equation 4.1 is the squared length of the sum of the Dec and Inc vectors with the entries of Inc permuted by the function M. By the triangle inequality, the length of their sum is at most $2\sqrt{\kappa}$ and so the squared length of their sum is at most 4κ. Therefore, $B_t \leq 4\kappa$. □

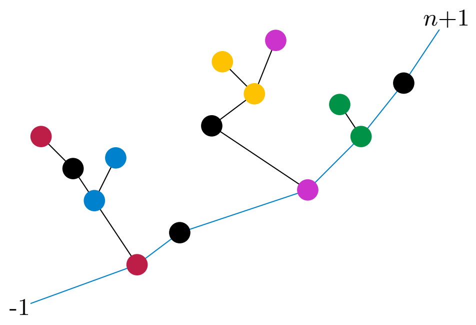


Figure 4.5: Every degree-three vertex is paired up with a leaf in one of it's subtrees. The node -1 is always paired with node $n+1$.

Maintaining Invariant 1 in the face of the random swaps in l' can be difficult, because new minima could be added to the path or old minima could be removed from the path. To handle these challenges, we pair up each element with degree three in the Cartesian tree with a descendant leaf. First, as a special case, the -1 element in the Cartesian tree is paired with the $n+1$ element. To find pairs for the degree-three elements, we consider traversing the tree in depth first order starting at the root. Below a degree-three element in the Cartesian tree there are two subtrees. When a degree-three element is encountered in the traversal, the larger of the maximum leaf element in the left subtree and the maximum leaf element in the right subtree will have already been paired up. So we pair the degree-three element with the unpaired (and smaller) of the two maximum leaves (Figure 4.5). For a degree-three element, $l[a]$, denote the index in l of its pair with $P(a)$. We enforce the following stronger invariant:

Invariant 2. *For every element* $l[a]$ *with degree three in the Cartesian tree,* $\sigma\big(P(a)\big) - \sigma(a) \leq Inc\big(l[P(a)]\big) + Dec\big(l[a]\big)$.

Invariant 2 implies Invariant 1, because each minima along the path is either paired with the maximum leaf element in its left subtree if it has one.

We now consider how to maintain Invariant 2 after each random swap in l'. Suppose $\sigma(a) = k+1$ and $\sigma(b) = k$ are the swapped pair and for now assume neither is the active element. After the swap $\sigma(a) = k$ and $\sigma(b) = k+1$ and the two counters $Dec\big(l[a]\big)$ and

$\mathit{Inc}\big(l[b]\big)$ are incremented. However, the slight upward and downward movement of elements may have changed how element's are paired up either by a structural change in the Cartesian tree or exchanging the relative value of two leaf elements. There are several cases to analyze based on how the random swap affected the modified Cartesian tree.

First we observe that if the random swap did not affect the pairing of elements, then the incrementing of counters maintains the invariant. For example, if a has a pair $P(a)$, then $\sigma\big(P(a)\big) - \sigma(a)$ is increased by one and if there is an element $l[c]$ with $P(c) = b$, then $\sigma(b) - \sigma(c)$ increased by one. Each of these increases are offset by the incrementing of $\mathit{Dec}\big(l[a]\big)$ and $\mathit{Inc}\big(l[b]\big)$ respectively.

If the random adjacent swap did affect the pairing of elements, then either $l[a]$ and $l[b]$ are adjacent in the tree or $l[a]$ and $l[b]$ are leaf elements with least common ancestor $l[c]$. In this second case, there is an ancestor of $l[c]$ paired with $l[a]$ before the swap which is paired with $l[b]$ after and $l[c]$ is paired with $l[b]$ before the swap and is paired with $l[a]$ after. For both pairing changes, the distance between the paired elements is unchanged, but the *Inc* counter of the leaf element in the pairs may be incorrect. So we exchange $\mathit{Inc}\big(l[a]\big)$ and $\mathit{Inc}\big(l[b]\big)$.

In the case where $l[b]$ and $l[a]$ are adjacent in the tree, before the swap $l[b]$ is the parent of $l[a]$ and afterwards $l[a]$ is the parent of $l[b]$. When this happens, if either $l[a]$ or $l[b]$ are unsorted elements, then both elements must lie on the minima path and the swap simply exchanges their order on the minima path. So while there is a change in the tree structure, there is no change in the pairing of elements.

We can now assume both elements are semi-sorted which leads to some case analysis based on the degrees of $l[a]$ and $l[b]$ which determines how they are paired with other elements. In these cases, the random swap acts almost like a tree rotation.

- If $l[a]$ and $l[b]$ both have degree three, then together there are three subtrees below $l[a]$

and $l[b]$. For the largest elements in these three subtrees, one is paired with $l[a]$, one is paired with $l[b]$, and the third is paired with an ancestor of $l[a]$ and $l[b]$. After the random swap, the ancestor will have the same paired element, but $l[a]$ and $l[b]$ may have had their pairs exchanged. In this case, to maintain our invariant if the pairings changed, we exchange $Dec\big(l[a]\big)$ and $Dec\big(l[b]\big)$.

  This case is shown in Figure 4.6.

- If either $l[a]$ or $l[b]$ has degree three and the other has degree two, then there are two subtrees below $l[a]$ and $l[b]$ in the subtree. Out of the two maximums in the subtrees, one is associated with whichever of $l[a]$ and $l[b]$ has two children and one is associated with an ancestor of $l[a]$ and $l[b]$. Notice that when a swap happens, the degree of $l[a]$ and $l[b]$ will not change if there is a subtree "between them" i.e. there are descendants of $l[a]$ and $l[b]$ with index between a and b (or equivalently $|a - b| \neq 1$).

  When there is no subtree between $l[a]$ and $l[b]$, then the swap exchanges the degrees of the two elements. In this case, to maintain the invariant we also exchange $Dec\big(l[a]\big)$ and $Dec\big(l[b]\big)$.

- If $l[a]$ has degree one and $l[b]$ has degree three, then there is only one subtree below $l[a]$ and $l[b]$. Because $\sigma(a) = \sigma(b) + 1$, that subtree's maximum must be larger than $\sigma(a)$. So $P(b) = a$. After the swap, this pairing relationship is destroyed, because both elements will have degree two. In this case, no additional work is needed to maintain the invariant.

- If $l[a]$ and $l[b]$ both have degree two, then there is only one subtree below $l[a]$ and $l[b]$. Again we condition on whether or not there is a subtree between $l[a]$ and $l[b]$.

  If there is a subtree between them, then the swap simply reorders $l[a]$ and $l[b]$ on the path leading to that subtree causing no change in pairings and maintaining the invariant.

When there is no such subtree, after the swap, one of $l[b]$ will now be a leaf, $l[a]$ will have degree three, and $P(a) = b$. In this case, a new pairing relationship was created between $l[a]$ and $l[b]$. The swap incremented $Dec\big(l[a]\big)$ and $Inc\big(l[b]\big)$ so $\sigma\big(l[a]\big) - \sigma\big(l[b]\big) = 1 < 2 \leq Inc\big(l[b]\big) + Dec\big(l[a]\big)$ and the invariant holds.

- If $l[a]$ has degree one and $l[b]$ has degree two, then there are no subtrees below $l[a]$ and $l[b]$. After the swap, they will switch which element is the leaf. An ancestor was paired with $l[a]$ and is now paired with $l[b]$.

  In this case, to maintain the invariant we exchange $Inc\big(l[a]\big)$ and $Inc\big(l[b]\big)$.

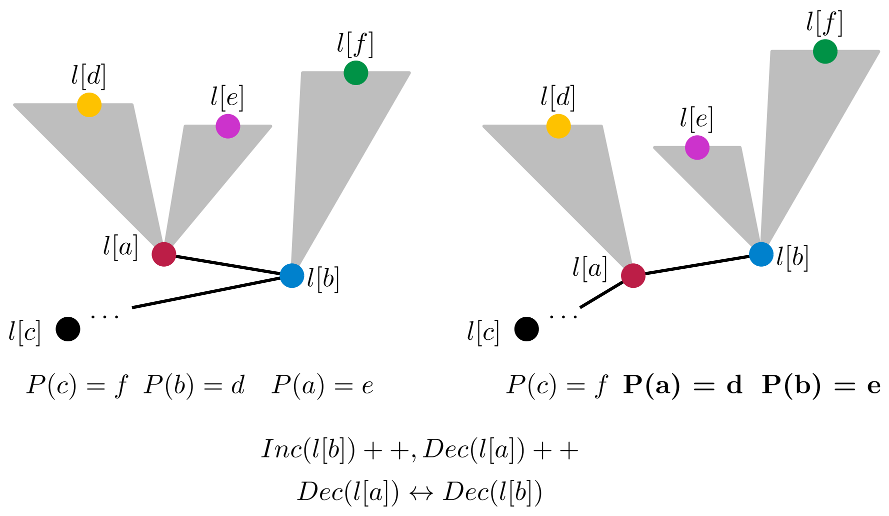

$l[d]$
$l[e]$
$l[f]$
$l[a]$
$l[b]$
$l[c]$
$\cdots$
$P(c) = f \;\; P(b) = d \;\; P(a) = e$
$l[d]$
$l[e]$
$l[f]$
$l[a]$
$l[b]$
$l[c]$
$\cdots$
$P(c) = f \;\; \mathbf{P(a) = d \;\; P(b) = e}$
$Inc(l[b]) + +, Dec(l[a]) + +$
$Dec(l[a]) \leftrightarrow Dec(l[b])$

Figure 4.6: When the red and blue element are randomly swapped in l', they switch paired elements and exchanging their Dec counters maintains the invariant.

When the random adjacent swap in l' involves the active element, the affect on the Cartesian tree can be somewhat more complicated. Issues might arise because $l[j]$ is not yet slotted into its simulated final horizontal position in the Cartesian tree. We need to make sure the horizontal movements of the active element do not invalidate the invariant. Suppose there is a maximal index $k < j$ such that $\sigma(k) < \sigma(j)$, i.e., index $k + 1$ is where the insertion of $l[j]$ will stop. When there is no such k, $l[j]$ will be inserted at the front of the list and so we set k to be -1. If $l[j]$ swaps with an element outside the range $[k, j - 1]$, then no horizontal movement of $l[j]$ will occur and we can handle the case as though $l[j]$ is semi-sorted.

So suppose $l[j]$ is swapped with $l[a]$ with $a \in [k, j-1]$ and $\sigma(a) = \sigma(j) + 1$ before the swap. After the swap, $l[j]$ will be moved immediately to the right of $l[a]$ in the Cartesian tree and is the right child of $l[a]$. Because $\sigma(j)$ is smaller than $\sigma(x)$ for $x \in [k, j-1]$, $l[a]$ must be the right child of $l[j]$ before the swap. So $l[j]$ has degree two and is unpaired before the swap.

- If $l[a]$ had a right child before the swap, then $l[j]$ now subdivides the edge from $l[a]$ to its old right child and has degree two. So the invariant is maintained.
- If $l[a]$ had only a left child before the swap, then $l[a]$ is now paired with $l[j]$, which is a leaf after the swap. The invariant requires $\sigma(j) - \sigma(a) = 1 \le \mathit{Inc}\big(l[j]\big) + \mathit{Dec}\big(l[a]\big)$. This inequality is satisfied, because the swap incremented $\mathit{Inc}\big(l[j]\big)$.
- If $l[a]$ was a leaf paired with $l[c]$ before the swap, then $l[j]$ is now paired with $l[c]$. Exchanging the *Inc* counters for $l[j]$ and $l[a]$ guarantees the invariant is maintained.

Now we consider the final case where $l[j]$ is swapped with $l[a]$ with $a \in [k, j-1]$ and $\sigma(a) + 1 = \sigma(j)$ before the swap. Because $\sigma(a) < \sigma(j)$, $a = k$. Additionally we observe that $l[j]$ is the right child of $l[a]$ in the Cartesian tree before the swap. After the swap, $l[a]$ is the right child of $l[j]$ and $l[j]$ has degree two. So $l[j]$ is unpaired after the swap.

- If $l[j]$ had a right child before the swap, then $l[j]$ now subdivides the edge from $l[a]$ to its old parent and has degree two. So the invariant is maintained.
- If $l[j]$ is a leaf and $l[a]$ has a left child, then $l[a]$ was paired with $l[j]$ before the swap. After the swap, $l[a]$ and $l[j]$ both have degree two with $l[j]$ subdividing the old edge between $l[a]$ and its parent.
- If $l[j]$ is a leaf and $l[a]$ does not have a left child, then there is some ancestor paired with $l[j]$. The pairing will switch to $l[a]$ after the swap. Exchanging the *Inc* counters for $l[j]$ and $l[a]$ maintains the invariant.

The above case analysis describes a counter maintenance strategy satisfying the following lemma.

Lemma 4.8. *There is a counter maintenance strategy that maintains Invariant 2 such that after each random adjacent swap in l', the corresponding counters are incremented and then some counters are exchanged between pairs of elements.*

4.4.2 Bounding the Counters with Balls and Bins

We model the *Inc* and *Dec* counters each with a balls and bins process and analyze the sum of squares of balls in each bin. Each element in l is associated with one of n bins. When an element's *Inc* counter is increased, throw a ball into the corresponding bin. If a pair of *Inc* counters are exchanged, exchange the set of balls in the two corresponding bins. The *Dec* counters can be modeled similarly.

This process is almost identical to throwing balls into n bins uniformly at random. Note that the exchanging of balls in pairs of bins takes place after a ball has been placed in a chosen bin, effectively permuting two bin labels in between steps. If every bin was equally likely to be hit at each time step, then permuting the bin labels in this way would not change the final sum of squares and the exchanging of counters could be ignored entirely. Unfortunately the bin for the element at $l[n-1]$ in the case of *Inc* counters or $l[0]$ in the case of *Dec* counters cannot be hit, i.e., there is a forbidden bin controlled by the counter swapping strategy. However, even when in each round the forbidden bin is adversarially chosen, the sum of squares of the number of balls in each bin will be stochastically dominated by a strategy of always forbidding the bin with the lowest number of balls. Therefore, the sum of squares of m balls being thrown uniformly at random into $n-1$ bins stochastically dominates the sum of squares of the *Inc* (or *Dec*) counters after m steps.

Theorem 4.3. *If cn balls are each thrown uniformly at random into n bins with $c > e$, then* *the sum over the bins of the square of the number of balls in each bin is at most* $3c^2n$ *with exponentially high probability.*

Proof. Let $X_1, \ldots, X_n$ be random variables where X_k is the number of balls in bin k and let $Y_1, \ldots, Y_n$ be independent Poisson random variables with $\lambda = c$.

By the Poisson approximation, Lemma 4.1,

$$\Pr\left[\sum_k X_k^2 \geq 3c^2n\right] \leq e\sqrt{cn}\Pr\left[\sum_k Y_k^2 \geq 3c^2n\right].$$

Let Z_k be the event that $Y_k \geq ecn^{1/6}$ and Z be the event that at least one Z_k occurs.

$$\begin{aligned}
&\Pr[Z] \leq n\Pr[Z_1] \quad \text{by a union bound.}\\
&\Pr[Z_1] = e^{-c}\sum_{k=ecn^{1/6}}^{\infty}\frac{c^k}{k!} \leq e^{-c}\sum_{k=ecn^{1/6}}^{\infty}\frac{c^k}{e\left(\frac{k}{e}\right)^k}\\
&\quad = e^{-c-1}\sum_{k=ecn^{1/6}}^{\infty}\left(\frac{ec}{k}\right)^k \leq e^{-c-1}\sum_{k=ecn^{1/6}}^{\infty}\left(\frac{1}{n^{1/6}}\right)^k\\
&\quad = e^{-c-1}(n^{1/6})^{-ecn^{1/6}}\sum_{k=0}^{\infty}\frac{1}{n^{1/6}}^k \leq e^{-c}n^{-\frac{ec}{6}n^{1/6}}.\\
&\Rightarrow \Pr[Z] \leq \frac{n}{e^c n^{\frac{ec}{6}n^{1/6}}} \leq e^{-\Omega(n^{1/6})}.
\end{aligned}$$

Letting $Y = \sum_k Y_k^2$:

$$E[Y|\neg Z] \leq E[Y] = nE[Y_1^2] = n\left(c + c^2\right) \leq 2c^2n.$$

Given $\neg Z$, $(Y_k)^2 \in [0, ecn^{1/3}]$. So we can apply Hoeffding's inequality, Lemma 4.2, to get:

$$\Pr\left[Y - E\left[Y|\neg Z\right] \geq tn|\neg Z\right] \leq e^{-2t^2n^2/\left(n\left(ecn^{1/3}\right)^2\right)}.$$

Setting $t = c^2$, we have:

$$\begin{aligned}\Pr\left[Y - E\left[Y|\neg Z\right] \geq c^2n|\neg Z\right] &\leq e^{\left(-2c^4n^2\right)/\left(n\left(ecn^{1/3}\right)^2\right)} \\ &\leq e^{-2n^{1/3}}.\end{aligned}$$

Because $E\left[Y|\neg Z\right] \leq 2c^2n$, we have $\Pr[Y \geq 3c^2n|\neg Z] \leq e^{-\Omega(n^{1/3})}$.

$$\begin{aligned}\Pr\left[Y \geq 3c^2n\right] &= \Pr\left[Y \leq 3c^2n \text{ and } Z\right] + \Pr\left[Y \leq c^2n \text{ and } \neg Z\right] \\ &\leq \Pr[Z] + \Pr\left[Y \leq 3c^2n|\neg Z\right] \\ &\leq e^{-\Omega(n^{1/6})} + \Pr[Y - E[Y|\neg Z] \geq c^2n|\neg Z] \\ &\leq e^{-\Omega(n^{1/6})} + e^{-\Omega(n^{1/3})} \leq 2e^{-\Omega(n^{1/6})}.\end{aligned}$$

Thus, we can conclude $\Pr[\sum_k X_k^2 \geq 3c^2n] \leq \frac{2e\sqrt{cn}}{e^{\Omega(n^{1/6})}} \leq e^{-\,poly(n)}$. □

Recall that by Lemma 4.6, if an insertion-sort round ends at time t, then $I_{t_s} \leq 2(t-t_s)+B_t+1$. Theorem 4.3 and a simple union bound tell us that if $t \leq t_s + cn$, then $\sum_{k=0}^{n-1} Inc\big(l[k]\big)^2 \leq 3c^2(n-1)$ and $\sum_{k=0}^{n-1} Dec\big(l[k]\big)^2 \leq 3c^2(n-1)$ with exponentially high probability. So by Lemma 4.7, $B_t \leq 12c^2n$.

Recall that when the insertion sort round finishes, $2(t_e - t_s - 1) + B_{t_e} + 1 \geq I_{t_s}$. If fewer than cn steps have been performed, the left hand side of this inequality is less than $(12c^2 + 2c)n$ with exponentially high probability. Therefore, if we started with $(12c^2 + 2c)n$ inversions, the current round of insertion sort must perform at least cn steps with exponentially high probability; otherwise, there are unfixed but still "good" inversions. This completes the proof of Lemma 4.5.

4.5 Conclusion

We have shown that, although it is much simpler than quicksort and only fixes at most one inversion in each step, repeated insertion sort leads to the asymptotically optimal number of inversions in the evolving data model. We have also shown that we can get to this steady state after an initial phase of $O(n \log n)$ steps, which is also asymptotically optimal.

For future work, it would be interesting to explore whether our results can be composed with other problems involving algorithms for evolving data, where sorting is a subcomponent. For additional future work, it might also be interesting to study other sorting algorithms, either empirically or analytically, to see if they achieve similar number of inversions in the steady state in the evolving data model. Finally, it would also be interesting to explore whether one can derive a much better ϵ value than we derived in the proof of Lemma 4.4.

Bibliography

[1] R. C. Agarwal. A super scalar sort algorithm for risc processors. *SIGMOD Rec.*, 25(2):240–246, June 1996.

[2] M. Ajtai, J. Komlós, and E. Szemerédi. An 0(n log n) sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 1–9, New York, NY, USA, 1983. ACM.

[3] L. Alonso and E. M. Reingold. Analysis of Boyer and Moore's MJRTY algorithm. *Information Processing Letters*, 113(13):495–497, 2013.

[4] L. Alonso, E. M. Reingold, and R. Schott. Determining the majority. *Information Processing Letters*, 47(5):253 – 255, 1993.

[5] A. Amir, M. Farach, R. M. Idury, J. A. Lapoutre, and A. A. Schaffer. Improved dynamic dictionary matching. *Information and Computation*, 119(2):258–282, 1995.

[6] A. Amir, G. Franceschini, R. Grossi, T. Kopelowitz, M. Lewenstein, and N. Lewenstein. Managing unbounded-length keys in comparison-driven data structures with applications to online indexing. *SIAM J. Comput.*, 43(4):1396–1416, 2014.

[7] A. Anagnostopoulos, R. Kumar, M. Mahdian, and E. Upfal. Sorting and selection on dynamic data. *Theoretical Computer Science*, 412(24):2564–2576, 2011. Special issue on selected papers from 36th International Colloquium on Automata, Languages and Programming (ICALP 2009).

[8] A. Anagnostopoulos, R. Kumar, M. Mahdian, E. Upfal, and F. Vandin. Algorithms on evolving graphs. In *3rd ACM Innovations in Theoretical Computer Science Conference (ITCS)*, pages 149–160, 2012.

[9] L. Babai, P. Erdös, and S. M. Selkow. Random graph isomorphism. *SIAM Journal on Computing*, 9(3):628–635, 1980.

[10] B. Bahmani, R. Kumar, M. Mahdian, and E. Upfal. Pagerank on an evolving graph. In *18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 24–32, 2012.

[11] R. Beigel, W. Hurwood, and N. Kahale. Fault diagnosis in a flash. In *36th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 571–580, 1995.

[12] R. Beigel, S. R. Kosaraju, and G. F. Sullivan. Locating faults in a constant number of parallel testing rounds. In *1st ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 189–198, 1989.

[13] R. Beigel, G. Margulis, and D. A. Spielman. Fault diagnosis in a small constant number of parallel testing rounds. In *5th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 21–29, 1993.

[14] A. Ben-Aroya and S. Toledo. Competitive analysis of flash-memory algorithms. In Y. Azar and T. Erlebach, editors, *European Symp. on Algorithms (ESA)*, volume 4168 of *LNCS*, pages 100–111. Springer, 2006.

[15] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In R. H. Möhring and R. Raman, editors, *Euro. Symp. on Algorithms (ESA)*, volume 2461 of *LNCS*, pages 152–164. Springer, 2002.

[16] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. *SIAM Journal on Computing*, 35(2):341–358, 2005.

[17] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. *Journal of Algorithms*, 3(2):115–136, 2004.

[18] M. A. Bender, J. T. Fineman, S. Gilbert, T. Kopelowitz, and P. Montes. File maintenance: When in doubt, change the layout! In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, (SODA)*, pages 1503–1522, 2017.

[19] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti. Introduction to flash memory. *Proceedings of the IEEE*, 91(4):489–502, 2003.

[20] M. Braverman and E. Mossel. Noisy sorting without resampling. In *19th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 268–276, 2008.

[21] D. Breslauer and G. F. Italiano. Near real-time suffix tree construction via the fringe marked ancestor problem. *J. Discrete Algorithms*, 18:32–48, 2013.

[22] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. In *Proc. 13th Annual Symposium on Discrete Algorithms (SODA)*, pages 39–48, 2002.

[23] J. Bulánek, M. Koucký, and M. E. Saks. Tight lower bounds for the online labeling problem. *SIAM J. Comput.*, 44(6):1765–1797, 2015.

[24] C. Castelluccia, S. Jarecki, and G. Tsudik. Secret handshakes from CA-oblivious encryption. In P. J. Lee, editor, *Advances in Cryptology - ASIACRYPT*, pages 293–307. Springer, 2004.

[25] S. Chen, P. B. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In *5th Conf. on Innovative Data Systems Research (CIDR)*, pages 21–31, 2011.

[26] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

[27] R. Cole and R. Hariharan. Dynamic LCA queries on trees. *SIAM J. Comput.*, 34(4):894–923, 2005.

[28] D. J. Cook and L. B. Holder. *Mining Graph Data*. John Wiley & Sons, 2006.

[29] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[30] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[31] W. E. Devanny, M. T. Goodrich, and K. Jetviroj. Parallel equivalence class sorting: Algorithms, lower bounds, and distribution-based analysis. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 265–274, New York, NY, USA, 2016. ACM.

[32] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *19th ACM Symp. on Theory of Computing (STOC)*, pages 365–372, 1987.

[33] P. F. Dietz. Fully persistent arrays (extended array). In *Algorithms and Data Structures, Workshop WADS '89, Ottawa, Canada, August 17-19, 1989, Proceedings*, pages 67–74, 1989.

[34] P. F. Dietz and R. Raman. Persistence, amortization and randomization. In *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 78–88, 1991.

[35] D. Dobkin and J. I. Munro. Determining the mode. *Theoretical Computer Science*, 12(3):255–263, 1980.

[36] Y. Emek and A. Korman. New bounds for the controller problem. *Distributed Computing*, 24(3-4):177–186, 2011.

[37] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsificationa technique for speeding up dynamic graph algorithms. *Journal of the ACM (JACM)*, 44(5):669–696, 1997.

[38] D. Eppstein, M. T. Goodrich, M. Mitzenmacher, and P. Pszona. Wear minimization for cuckoo hashing: How not to throw a lot of eggs into one basket. In J. Gudmundsson and J. Katajainen, editors, *13th Int. Symp. on Experimental Algorithms (SEA)*, volume 8504 of *LNCS*, pages 162–173, Cham, 2014. Springer.

[39] D. Eppstein, M. T. Goodrich, and J. Z. Sun. Skip quadtrees: Dynamic data structures for multidimensional point sets. *International Journal of Computational Geometry & Applications*, 18(01n02):131–160, 2008.

[40] U. Feige, P. Raghavan, D. Peleg, and E. Upfal. Computing with noisy information. *SIAM Journal on Computing*, 23(5):1001–1018, 1994.

[41] L. R. Ford and S. M. Johnson. A tournament problem. *The American Mathematical Monthly*, 66(5):387–389, 1959.

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

[43] M. T. Goodrich. Pipelined algorithms to detect cheating in long-term grid computations. *Theoretical Computer Science*, 408(2–3):199–207, 2008.

[44] M. T. Goodrich and R. Tamassia. *Algorithm Design and Applications.* Wiley Publishing, 1st edition, 2014.

[45] M. T. Goodrich and R. Tamassia. *Algorithm Design and Applications.* Wiley Publishing, 1st edition, 2014.

[46] R. Graham. An efficient algorith for determining the convex hull of a finite planar set. *Information Processing Letters*, 1(4):132 – 133, 1972.

[47] B. Groz and T. Milo. Skyline queries with noisy comparisons. In *34th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS)*, pages 185–198, 2015.

[48] Y. Han and M. Thorup. Integer sorting in o (n/spl radic/(log log n)) expected time and linear space. In *Foundations of Computer Science, 2002. Proceedings. The 43rd Annual IEEE Symposium on*, pages 135–144. IEEE, 2002.

[49] D. S. Hirschberg. Fast parallel sorting algorithms. *Commun. ACM*, 21(8):657–661, Aug. 1978.

[50] D. S. Hochbaum. Ranking sports teams and the inverse equal paths problem. In P. Spirakis, M. Mavronicolas, and S. Kontogiannis, editors, *2nd Int. Workshop on Internet and Network Economics (WINE)*, volume 4286 of *Lecture Notes in Computer Science*, pages 307–318, Berlin, Heidelberg, 2006. Springer.

[51] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

[52] Q. Huang, X. Liu, X. Sun, and J. Zhang. Partial sorting problem on evolving data. *Algorithmica*, pages 1–24, 2017.

[53] S. Irani, M. Naor, and R. Rubinfeld. On the time and space complexity of computation using write-once memory or is pen really much worse than pencil? *Mathematical Systems Theory*, 25(2):141–159, 1992.

[54] A. Itai, A. G. Konheim, and M. Rodeh. *A sparse table implementation of priority queues.* Springer, 1981.

[55] S. Jarecki and X. Liu. Unlinkable secret handshakes and key-private group key management schemes. In J. Katz and M. Yung, editors, *5th Int. Conf. on Applied Cryptography and Network Security (ACNS)*, pages 270–287. Springer, 2007.

[56] V. Jayapaul, J. I. Munro, V. Raman, and S. R. Satti. Sorting and selection with equality comparisons. In F. Dehne, J.-R. Sack, and U. Stege, editors, *14th Int. Symp. on Algorithms and Data Structures (WADS)*, pages 434–445. Springer, 2015.

[57] V. Kanade, N. Leonardos, and F. Magniez. Stable Matching with Evolving Preferences. In K. Jansen, C. Mathieu, J. D. P. Rolim, and C. Umans, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM)*, volume 60 of *LIPIcs*, pages 36:1–36:13, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[58] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Pearson Education, 2nd edition, 1998.

[59] D. E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[60] T. Kopelowitz. On-line indexing for general alphabets via predecessor queries on subsets of an ordered list. In *IEEE Symp. on Found. of Comp. Sci. (FOCS)*, pages 283–292, 2012.

[61] T. Kopelowitz, G. Kucherov, Y. Nekrich, and T. A. Starikovskaya. Cross-document pattern matching. *J. Discrete Algorithms*, 24:40–47, 2014.

[62] L. Kozma. Useful inequalities cheat sheet, 2016. `http://www.lkozma.net/inequalities_cheat_sheet/`.

[63] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[64] S. Lohr. Frances e. holberton, 84, early computer programmer. *New York Times*, 2011.

[65] K. Makarychev, Y. Makarychev, and A. Vijayaraghavan. Sorting noisy data with partial information. In *4th ACM Conference on Innovations in Theoretical Computer Science (ITCS)*, pages 515–528, 2013.

[66] W. Meyer. Equitable coloring. *The American Mathematical Monthly*, 80(8):920–922, 1973.

[67] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[68] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, New York, NY, USA, 2005.

[69] H. Orman. The Morris worm: A fifteen-year perspective. *IEEE Security & Privacy*, 1(5):35–43, 2003.

[70] S. Parthasarathy, S. Tatikonda, and D. Ucar. A survey of graph mining techniques for biological datasets. In C. C. Aggarwal and H. Wang, editors, *Managing and Mining Graph Data*, pages 547–580. Springer, 2010.

[71] P. Pavan, R. Bez, P. Olivo, and E. Zanoni. Flash memory cells—an overview. *Proceedings of the IEEE*, 85(8):1248–1271, 1997.

[72] A. Pelc and E. Upfal. Reliable fault diagnosis with few tests. *Combinatorics, Probability and Computing*, 7:323–333, 1998.

[73] F. P. Preparata, G. Metze, and R. T. Chien. On the connection assignment problem of diagnosable systems. *IEEE Trans. on Electronic Computers*, EC-16(6):848–854, 1967.

[74] M. E. Saks and M. Werman. On computing majority by comparisons. *Combinatorica*, 11(4):383–387, 1991.

[75] Y. Shiloach and U. Vishkin. Finding the maximum, merging, and sorting in a parallel computation model. *Journal of Algorithms*, 2(1):88 – 102, 1981.

[76] A. Sorniotti and R. Molva. A provably secure secret handshake with dynamic controlled matching. *Computers & Security*, 29(5):619–627, 2010.

[77] R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

[78] L. G. Valiant. Parallelism in comparison problems. *SIAM Journal on Computing*, 4(3):348–355, 1975.

[79] J. Vuillemin. A unifying look at data structures. *Commun. ACM*, 23(4):229–239, 1980.

[80] D. Willard. Maintaining dense sequential files in a dynamic environment (extended abstract). In *Proc. 14th Annual Symposium on Theory of Computing (STOC)*, pages 114–121, 1982.

[81] D. E. Willard. Good worst-case algorithms for inserting and deleting records in dense sequential files. In *Proc. International Conference on Management of Data (SIGMOD)*, pages 251–260, 1986.

[82] H.-S. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, 2010.

[83] S. Xu and M. Yung. K-anonymous secret handshakes with reusable credentials. In *11th ACM Conf. on Computer and Communications Security (CCS)*, pages 158–167, 2004.

[84] J. Zhang and Q. Li. Shortest paths on evolving graphs. In H. Nguyen and V. Snasel, editors, *5th Int. Conf. on Computational Social Networks (CSoNet)*, volume 9795 of *Lecture Notes in Computer Science*, pages 1–13, Berlin, Heidelberg, 2016. Springer.