

Examining Computational Geometry, Van Emde Boas Trees and Hashing from the Perspective of the Fusion Tree

Dan E. Willard *

Published in Siam Journal on Computing 29 (2000) pp. 1030–1049.

Abstract

This article illustrates several examples of computer science problems whose performance can be improved with the use of either the fusion trees [24, 25] or some related data structures by Albers, Andersson, Ben-Amram, Galil, Hagerup, Mitlenssen, Nilsson, Raman and Thorup. It is likely that many other data structures can also have their performance improved with fusion trees. The examples here are only illustrative.

1. Introduction

The fusion tree [24] and its several variants [1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 25, 35, 38, 39, 41, 42, 43, 44] are data structures that perform sorting operations in better than $O(N \log N)$ worst-case time and execute dynamic search operations in faster than $O(\log N)$ time in a very natural model of complexity. Since many classic algorithms were designed under the assumption that further improvements for Sorting and Searching were impossible, one would intuitively anticipate many of the classic search procedures to permit $O(\log \log N)$ or better time improvements when their use of conventional Dynamic search trees is simply replaced by the faster underlying data structures made theoretically possible by new advances in data structure theory.

This article will illustrate several examples of such improvements. Most of these improvements will use a data structure, called the q-heap, as an intermediate device to speed up the search methodology. Q-heaps were introduced in [25] as a vehicle for solving the Minimum Spanning Tree problem in linear time. However, they also have many other applications. They will be shown in this article to improve many other classic searching problems.

*Computer Science Department, University at Albany. Email=dew@cs.albany.edu. Partially Supported by NSF Grant CCR 9302920. The article [51] was the earlier conference proceedings version of this paper.

The particular examples and new results we establish are listed at the bottom of this paragraph. None of the so-called “*improvements*” in this list are practical improvements because the coefficients associated with their asymptotic changes are undesirably large. However Item (1) below (dynamic universal hashing) is a foundational problem in Computer Science. All the other topics already appear (in some form) in Volumes 1 and 3 of Mehlhorn’s textbook. Thus our improvements are very germane to a 2-semester course on General Algorithms and Computational Geometry. The six topics (below) are only intended as a representative sample to illustrate what can be done theoretically (albeit not practically) when fusion trees and q-heaps are applied to several problems:

1. a variant of dynamic hashing where each lookup, insertion and deletion respects an $O(1)$ time bound with a probability exceeding $1 - o(N^{-(\log N)^K})$, for any arbitrary constant $K > 0$. (This result is a significant improvement over Dietzfelbinger and Meyer auf de Heide [21]’s Universal Hash scheme. It will probably be the most noteworthy result of this paper because Hashing is one of the basic operations studied in Computer Science. The improvement is based on combining the fusion-tree formalism with some of Siegel’s theorems [40] about universal hashing.)
2. an improved version of McCreight’s Priority-Search trees [34] that reduces the worst-case dynamic time performance to $O(\log N / \log \log N)$. (This result immediately implies that one can search a set of N rectangles and output all K intersections among this set in time $O(K + N \log N / \log \log N)$. See Mehlhorn’s textbook [33] for a concise description of McCreight’s Priority-Search trees.)
3. a related $\log \log N$ improvement of Chazelle’s $O(N \log^{d-1} N / \log \log N)$ space data structure [15] that performs the reporting version of d -dimensional orthogonal range queries in time $O(\log^{d-1} N / \log \log N + K)$, where K is the number of elements outputted.
4. for any dimension $d \geq 2$ and any $p \geq d - 1$, it will be possible to construct an $O(N \log^p N)$ space structure that supports an $O\{(\log N / \log \log N)^{d-1}\}$ time for executing the aggregate version of orthogonal range queries. (Section 6 will explain how there are at least certain perspectives where this combination of time and space can be viewed as optimal.)

5. an improved variant of van Emde Boas trees [46, 45, 27] that has a somewhat improved memory space.
6. a $\log \log N$ speedup of the Bentley-Shamos linear space method [10, 14] that calculates a set of N d -dimensional ECDF statistics in time $O(N \log^{d-1} N / \log \log N)$.

Each of the results above will employ the q-heap data structure from [25]. Section 9 will briefly illustrate several further examples of algorithmic problems whose performance can be improved with merely a faster sorting algorithm. In particular, such problems can have their runtimes improved by either the Fredman-Willard sorting procedure or by the further subsequent improvements of this procedure that have been proposed by Andersson, Hagerup, Nilsson, Raman and Thorup [4, 5, 42, 44].

We do not believe that any of the six sample topics mentioned in the preceding paragraph are the main point. Rather the key question is how many other well known results in Computer Science can undergo a similar theoretical (albeit perhaps impractical) asymptotic improvement when some type of application of [24, 25]’s fusion trees, q-heaps and their generalizations are employed? For instance, Thorup recently discovered a worst-case linear algorithm algorithm for solving Dijkstra’s Single Source Shortest Path problem (SSSP) with a method, that uses some of [25]’s constructs in one of its main interim steps. In the present article, we focus mostly on a moderately narrow bandwidth of problems, drawn from Computational Geometry and Information Retrieval theory, simply because such problems reflect the author’s particular expertise and knowledge. However, it will become apparent to the reader who examines our six sample problems, in the context of the expanding literature [1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 24, 25, 35, 38, 39, 41, 42, 43, 44] that there are surely many other problems that can undergo theoretical (although often not practical) improvements when some type of variant of fusion tree is present.

Our goal in this article will thus not be exclusively to address the six sample problems mentioned. Sections 2-4 are written so that they can be understood by a reader who is unacquainted with both fusion trees and computational geometry. Their goal will be to provide such a reader with an intuitive feel for this subject, ending with a very curious example about hashing. Our more specialized discussion appears in Sections 4-9. They focus mostly around Computational Geometry

and Van Emde Boas trees.

2. Literature Survey

We will use largely the same notation and computational model that appeared in the articles [1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 24, 25, 35, 38, 39, 41, 42, 43, 44]. A word is assumed to consist of b bits, and each key shall be assumed to be a fixed point integer fitting into one word. The instruction set available to the computer will be arithmetic, bitwise logical, and comparison operations on b -bit words. The integer N will denote the size of the database we search. It will be assumed that $b \geq \log_2 N$ since otherwise our main memory search structures would not even have a sufficiently large word length to store the addresses of the N objects that are stored in the computer's main memory bank. The runtime of our algorithm will be viewed as quantity $F(b, N)$. It has not been usually done, but most of the algorithms of [1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 24, 25, 35, 38, 39, 41, 42, 43, 44] can be trivially generalized to common floating point representations of real numbers.

The first surprising article about fast dynamic tree operations was Van Emde Boas's data structure [46], which had a $O(\log \log U)$ Worst-Case retrieval, insertion and deletion time for a data structure using $O(U \log \log U)$ space to represent a set of elements from the universe $0, 1, 2, \dots, U$. The space was further compressed by Van Emde Boas to $O(U)$ in [45]. (One can also find a further discussion of the implications of Van Emde Boas trees and their variants in for instance [27, 28, 29, 47, 48].) Since $b = \log U$, one can think of Van Emde Boas's algorithm as having an $O(\log b)$ time. (Unless b is a very large number, this time is better than the conventional $O(\log N)$ Balance Tree time.)

The first contribution of [24]'s fusion trees was that it established a sorting time $F(b, N)$ which respected the worst-case asymptotic bound $O(N \log N / \log \log N)$ for an $O(N)$ data space structure (regardless of b 's and N 's value). This was the first method to obtain an $o(N \log N)$ time for sorting using what Andersson, Hagerup, Nilsson and Raman [5] have called a "conservative method" for measuring computing costs. Fusion trees also provided an $O(\log N / \log \log N)$ worst-case and $O(\sqrt{\log N})$ randomized time cost for performing standard Balance-Tree and Search-and-Update operations in

an $O(N)$ space structure. A cousin of the Fusion Tree, called the q-heap, can bring Dynamic Balance Tree Search, Insert and Delete worst-case times down to an $O(1)$ asymptote provided the relevant set has a $\text{Polylog}(N)$ size and one has access to an $o(N)$ sized Lookup Table. Q-heaps were introduced in [25] as a vehicle for devising a Worst-Case Linear Time algorithm for the Minimum Spanning Tree problem.

One open question raised by Fredman and Willard’s original paper [24] was whether or not one could achieve an $O(\sqrt{\log N})$ worst-case time cost for performing standard Balance-Tree Search-and-Update operations in an $O(N)$ space structure. Such a question was naturally pressing because Fredman and Willard had achieved an $O(\sqrt{\log N})$ worst-case time in the three cases where a static data structure had access to $O(N)$ space, a dynamic data structure was allowed more space, or where the environment was dynamic, the space was $O(N)$ but the time was now *randomized*. The final resolution to this open question should be credited to Arne Andersson. He showed in [4] that both an $O(\sqrt{\log N})$ amortized as well as worst-case (see footnote¹) cost for performing standard Balance-Tree Search-and-Update operations was indeed possible in $O(N)$ space.

Another open question raised by [24]’s discussion on Fusion Trees was whether the cost of sorting could be improved beyond the $O(N \log N / \log \log N)$ worst-case and $O(N\sqrt{\log N})$ randomized times. This problem has been studied extensively by [1, 4, 5, 38, 42, 44], and the best combined results from this evolving literature are roughly that $O(N)$ space supports either a randomized time $O(N \log \log N)$ or a worst-case time $O[N(\log \log N)^2]$ for sorting. In particular, the randomized problem was resolved by Andersson, Hagerup, Nilsson, Raman and Thorup [5, 42], and the best worst-case sorting is due to Thorup [44],

Another recent direction of research has been the study of Dijkstra’s Single Source Shortest Path Problem (SSSP). Thorup showed in [43] that it was possible to devise a fully $O(N)$ worst-case algorithm for constructing the SSSP. Most of Thorup’s algorithm is unrelated to the prior Fredman-Willard fusion tree research, but he does use a q-heap data structure as one important subcomponent

¹The relevant “worst-case” cost algorithm did not actually appear in Andersson’s paper [4], but we think it should be credited fully to Andersson because it is an easy extension of his amortization techniques combined with standard methods for converting an amortized optimizing algorithm into a worst-case control procedure. For instance, one could use techniques roughly similar to either our proof of Lemma 2 (see the Appendix) or [50, 52]’s methodologies to easily transform Andersson’s amortized optimizing algorithm into a worst-case controlling procedure.

of his final data structure.

There are simply too many new ideas in the rapidly expanding literature [1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 35, 38, 39, 41, 42, 43, 44] for us to describe all these results in full detail. Albers and Hagerup [1] have discussed parallel analogs of Fusion Tree Sorting. (An important aspect of their work is that some of its ideas are essential for implementing the *non-parallel sorting* algorithms of [5].) Ben Amran and Galil [11] illustrate a very eloquent hybrid perspective that allows one to visualize how the $N \log N$ lower bounds of Paul-Simon [36] can be compared with the contrasting $o(N \log N)$ Upper Bounds of Fredman-Willard [24] in a more general and unified setting. Many other more recent and expansive lower bounds have been developed by Andersson, Beame, Fische, Mitlerson, Riis and Thorup [6, 9]. Andersson, Mitlerson and Thorup showed in [3, 7] that the Fredman-Willard Fusion Tree results could be extended to a data model using exclusively AC^0 instructions. Raman and Thorup [38, 41] have developed some interesting new priority queue data structures, which ultimately led Thorup to announce a linear algorithm [43] for Dijkstra's SSSP problem and a worst-case $O[N(\log \log N)^2]$ algorithm [44] for sorting.

It should be noted that while many of these articles improve upon the original Fredman-Willard results by obtaining new upper and lower bounds, a large number of them (such as most notably [4, 43]) explicitly mention using particular parts of the original Fredman-Willard data structure as subroutines of their new improved algorithms. In essence what we will do in Sections 4-8 of this paper is also employ such subcomponents of Fusion Trees to examine six other algorithmic paradigms.

It also should be mentioned that it is not entirely true that all aspects of research related to Fusion Trees are fully divorced from practical application. For instance, the discussion of AC^0 circuits by Andersson, Mitlerson and Thorup [3, 7] could result in practical hard-wired algorithms. Also some of the engineering-grade sorting algorithms of Bentley, Bostic, McIlroy and McIlroy [13, 31] are partly related to Fusion Tree research in that one part of their procedures break a larger problem into locally smaller problems with tiny constants. It also should be mentioned that some of the older subcomponent data structures utilized by the original Fusion Trees, such as Perfect Hashing, Van Emde Baas Tree and Fast Tries [21, 23, 40, 45, 46, 47, 48] can be potentially pragmatic when used in a context *perhaps different from* that employed by the fusion tree.

As we noted in Section 1, our goal in the present article will be to consider six sample problems, which illustrate some types of theoretical (albeit mostly impractical) improvements that can result when the Fredman-Willard q-heap data structure is employed. In order to simplify the presentation, we will not require the reader to examine [25] or any other article that was mentioned in the preceding paragraphs of this Literature Survey. All the reader is required to know is that there is a *Black Box Software Package*, called the q-heap, that was proven in [25] to have the following characteristics:

Theorem 1. Suppose S is a subset of cardinality $M < \sqrt[5]{\log N}$ lying in a larger database consisting of N elements. Then there exists a q-Heap data structure for representing S such that the q-Heap uses $O(M)$ space and enables insertions, deletions, member and predecessor queries into the subset S to run in constant worst-case time, provided access is available to a precomputed lookup table of size $o(N)$.

The lookup table for q-heaps can be constructed in $O(N^\epsilon)$ time (for some infinitesimal constant $\epsilon > 0$). The space and preprocessing costs accrued to this lookup table are a minor expense because a large number of q-heaps will share access to one common search table. In other words, a large number of different mini-sets S_1, S_2, \dots, S_j will typically each have different q-heap data structures $Q(S_1), Q(S_2), \dots, Q(S_j)$, but they will share use of the very expensive common look-up table. This notion of sharing the expensive look-up table, jointly developed with Fredman in [25], is the reason the performance of such a large number of different well known data structures from the literature can be improved.

We emphasize that this article has been organized so that the reader does not need to know what algorithm is actually contained in Theorem 1's "Black Box" of software. It will matter not whether the software inside this curious Black Box is manufactured by some corporate giant, such as IBM, MicroSoft, NetScape, or by some financially stumbling Fredman-Willard Computer Company, which is called perhaps "The *NanoSoft* Corporation". Regardless of such details, our improvements over the prior literature's six sample problems will be easily comprehensible to the reader, as a theoretical family of algorithms, without knowing what lies inside the curious Black Box, inserted by the engineers of *NanoSoft*TM.

Indeed if I may drop the mild humor from the preceding paragraph, it makes a very serious point. It is that one does not need to understand the details of [24, 25]’s fusion procedures to grasp the nature of our six sample problems. That is, it is sufficient to view Theorem 1 as a Black Box of software. Our six sample problems are intended to motivate curiosity into [1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 24, 25, 35, 38, 39, 41, 42, 43, 44]’s subject matter by illustrating the breadth of the theoretical improvements which they make feasible.

Despite the perhaps impractical coefficients associated with most fusion-like algorithms, the enticing aspect of this topic is apparent when one realizes *how long* is the actual list of problems which can have some facet of their theoretical capacity undergo an asymptotically infinite (although usually impractical) improvement. The six sample problems examined in Sections 4-9 and Thorup’s recently announced linear solution to Dijkstra’s SSSP problem are only a sample of what can be done.

3. Corollaries to Theorem 1

This section will discuss two simple corollaries to Theorem 1, which will illustrate the main format in which we will apply Theorem 1. First we will introduce one useful lemma:

Lemma 1. For simplicity, let us assume that $B > 16$. Consider a B-tree whose internal nodes have arity between $B/8$ and B , whose root has arity between 2 and B , and whose leaves store the data and each have the same depth. Suppose that searches, insertions and deletions in such a tree of height h will have an $O(h)$ cost when no splits or merges occur, and the costs of splitting a node or merging two nodes is bounded by $O(B)$. Then regardless of the details of the structure of the node v , it is possible to devise an insertion and deletion algorithm for this tree that runs in amortized time $O(h)$, with the $O(h)$ asymptote using a constant that is independent of B . (Our main interest in Lemma 1 will be when Theorem 1’s q-heaps are the main organizing method for v ’s internal structure.)

Proof Sketch. Consider the natural B-tree insertion/deletion algorithm that merges a non-root internal node v with its sibling w if v ’s arity is less than $B/8$, that splits a node into two equal halves whenever its arity exceeds B , and which makes the child of a tree root into the new root

if the preceding operations caused the old root to have only one child. (Sometimes a merge will immediately trigger a split into two equal sized halves because a node becomes too large after the merge operation.) It is easy to devise an accounting function that shows there will be only an amortized number of $O(1/B)$ splits and merges in a tree of height h . (This is essentially because nodes of height j will have an amortized frequency of $O[(8/B)^{j+1}]$ of splitting and merging.) Hence the splits and merges will have an $O(1)$ amortized cost. This shows that the total cost of insertions and deletions is $O(h)$, since splitting and merging are the only costly operations outside a general $O(h)$ searching cost. Q.E.D.

Lemma 1 also holds if splits and merges have a cost proportional to the number of leaf descendants of a node. (Multi-branching B-trees with such properties were presented in [49].) However, such trees are not relevant to our present discussion.

We will frequently employ versions of Lemma 1's B-tree structure where $B = \sqrt[5]{\log N}$ and where a q-heap is used to format the structure of the individual internal nodes. This tree will have a height, search time, and update time proportional to $1 + \log M / \log \log N$ when it stores M elements. The term q*-heap will refer to such a modified B-tree form of q-heap. From the combination of Lemma 1 and Theorem 1, we thus have the following corollary:

Corollary 1. Assume in a database of N elements, we have available the use of precomputed tables of size $o(N)$. Then for sets of arbitrary cardinality $M \leq N$, it is possible to have available variants of q*-heaps using $O(M)$ space that have a worst-case time $O\{1 + \log M / \log \log N\}$ for doing member, predecessor and rank searches, and which support an amortized time $O\{1 + \log M / \log \log N\}$ for insertions and deletions.

The proof of Corollary 1 is an immediate consequence of Lemma 1 and Theorem 1 because Corollary 1's data structure simply consists of a B-tree storing M records whose internal nodes are q-heaps, and which has a branching factor $B = \sqrt[5]{\log N}$.

Lemma 2. Consider again a B-tree data structure that satisfies the hypothesis of Lemma 1. Then it is possible to develop a more elaborate version of Lemma 1's insertion and deletion algorithms so that the insertion or deletion of a leaf-record has an $O(h)$ worst-case (rather than amortized) cost.

The previous literature has illustrated many examples of amortized optimization algorithms which can also guarantee worst-case time, if their procedures are made somewhat more elaborate. A similar type of proof of Lemma 2 is sketched in an Appendix at the end of this article. The reason we postpone Lemma 2's proof until the Appendix is that its worst-case control procedure has a poor coefficient, and the techniques needed to transform Lemma 1's amortized time-optimizing algorithm into a worst-case controlled procedure is very similar to what has transpired often in the prior literature.

Corollary 2. The data structure in Corollary 1 can be improved so that its predecessor-query, member-query, insertion and deletion operations all have a worst-case complexity $O\{1 + \frac{\log M}{\log \log N}\}$.

Once again, no proof is needed to verify Corollary 2. It is an immediate consequence of Lemma 2 and Theorem 1 because Corollary 2's data structure simply consists of a B-tree storing M records, whose internal nodes are q-heaps, and which has a branching factor $B = \sqrt[5]{\log N}$. We use the term q*-heap to refer to either Corollaries 1's or 2's data structure because they are essentially the same concept, except that one uses an amortized-optimizing algorithm and the other employs a slightly more elaborate worst-case control. We close this section by emphasizing that the q*-heaps of Corollaries 1 and 2 are related to [25]'s q-heaps and AF-heaps. This latter data structure, originating in our joint paper with Fredman, stimulated Corollaries 1 and 2.

Finally, we wish to close this section by pointing out that there are roughly two types of applications of q*-heaps that are explored in this paper and the prior literature. One type of application is based on using the q*-heap to speed up an algorithm by essentially reducing the height of a tree. This method is feasible because the q*-heap can often allow us to traverse in $O(1)$ time tree nodes which have roughly $\text{PolyLog}(N)$ arity. An alternate approach is to forgo making any use of the q*-heap until one has essentially reduced an initial problem of size N to one of roughly size $(\log N)^c$ for some fixed constant $c > 0$, at which time the final processing step can be made to run in $O(1)$ time. In essence, the first method is used by us in Sections 5-7 of this paper, while the second was used by Fredman, Thorup and Willard in [25, 43] and will be again used by us in Section 4 of this paper.

4. Hashing

Most of the sample problems discussed in this article come from Computational Geometry. However, this article will begin by considering universal hashing because our proposed improvement has a short description and contains one pretty idea. It will be unnecessary for the reader to examine the prior literature on hashing before examining this section. Our discussion, although only an abbreviated outline, should be sufficiently self-contained for the reader inexperienced with Universal Hashing to still appreciate the gist.

Dietzfelbinger and Meyer auf de Heide [21] have developed a universal hashing scheme where each insertion, deletion and lookup has a probability exceeding $1 - o(N^{-k})$ of running in constant time (for an arbitrary $k > 0$). This section will illustrate how q*-heaps enable us to develop a quite different alternate structure that provides a better probability $1 - o(N^{-(\log N)^k})$ for these three operations to run in constant time (where $k > 0$ is again an arbitrary constant, and it is assumed again that the universe size $U < \text{Polynomial}(N)$).

The data structure for achieving this functionality rests on two concepts. First, it was noted by Corollary 2 that the q*-heap provides a formalism to perform constant time insertions, deletions and retrievals on any set of $\text{PolyLog}(N)$ cardinality. Now, consider a hash table, with N addressable buckets for representing a time-varying set of cardinality $\leq N$, which has no overflow mechanism and which simply stores in a q*-heap all the records that are mapped into a common bucket address. This hash table will thus assure that a bucket can be searched in $O(1)$ time provided it stores no more than $\text{Polylog}(N)$ elements.

The pleasing point is that the Poisson Probability distribution will assign each bucket a probability less than $o(N^{-(\log N)^{k+1}})$ for containing more than $(\log N)^k + 2$ elements. Thus, there will be a probability greater than $1 - o(N^{-(\log N)^k})$ that every single one of our N buckets will store no more than $(\log N)^k + 2$ elements. Thus for an arbitrary constant k , there is the same probability greater than $1 - o(N^{-(\log N)^k})$ that a constant time bound on searches, insertions and deletions will hold *within all N buckets simultaneously!*

Another pleasing point is that by applying some theorems of Siegel [40], we can strengthen

in a straightforward manner the Poisson probability analysis (above) into a formal theorem about classes of universal hash functions. To do so, we assume that there is a prespecified constant N bounding the maximal size of the time-varying set stored in the hash table and that the universe size satisfies $U \leq \text{Polynomial}(N)$. Then Siegel’s universal hash functions [40] will require only N^c words to form a class satisfying his $(\log N)^k$ -wise independence property, for any constant k . This fact immediately implies that the Poisson distribution is a sufficiently accurate probability predictor for this *universal hash class* to imply the same claimed $1 - o(N^{-\text{PolyLog}(N)})$ probability that each search and update operation will run in constant time. (We omit the further details because they are basically a routine hybridized application of Siegel’s quite sophisticated formalism [40] in the context of the data structure outlined in the prior paragraph of this section. The point is thus that *universal hash classes* are assured by Siegel’s analysis to operate in the same manner as the simpler analysis of randomized Poisson probability distributions from the prior paragraph of our discussion.)

The 3-paragraph passage (above) is obviously more complicated than it appears because it cannot be fully formalized without duplicating both Siegel’s full formalism [40] and [25]’s full proof that a q-heap data structure satisfies Theorem 1’s “Black Box” properties. The reason the preceding discussion is significant is that it uses essentially the same computing model as Dietzfelbinger and Meyer auf de Heide [21] had used for their universal hashing scheme. Our probability of constant time operations is an $1 - o(N^{-\text{PolyLog}(N)})$ magnitude, which is better than [21]’s probability of the form: $1 - o(N^{-k})$.

5. Priority-Search Trees

This section will examine McCreight’s Priority-Search Tree in considerable detail. This search problem will offer an excellent case study illustrating how a data structure can undergo theoretical (although possibly impractical) improvements when q-heaps are used to streamline it. McCreight’s Priority-Search Trees are described by both his journal article [34] and by Mehlhorn’s textbook [33]. We will assume that the reader has examined at least one of these sources when we present our $\log \log N$ improvements. The second half of this section will also explain how to produce similar theoretical (but again not practical) improvements upon one of Chazelle’s orthogonal range query

structures in [15].

Given a set S of N points on the xy-plane, define Query (a, b, c) to be a request for the subset of S that satisfies

$$a < x < b \wedge y > c \tag{1}$$

The McCrieght Priority-Search trees can perform this operation in $O(\log N + K)$ time (where K is the size of the output). It supports $\log N$ insert and delete operations and uses $O(N)$ space.

The Fusion Priority-Search tree described here will be a variant of B-tree, with a branching factor $B = O(\sqrt{\log N})$. Each node v will essentially store information about some subset of points whose x-value lies in some line segment $[A_v, B_v)$, whose “range” is implicitly defined by the B-tree’s stored keys. This line segment will be denoted as $\text{Range}(v)$, and $\text{Set}(v)$ will denote the subset of points from S whose x-coordinate lies in this interval. (This notation thus implies that if w_1, w_2, \dots, w_n are the children of v then the union of $\text{Range}(w_1), \text{Range}(w_2), \dots, \text{Range}(w_n)$ will equal $\text{Range}(v)$, and similarly the union of $\text{Set}(w_1), \text{Set}(w_2), \dots, \text{Set}(w_n)$ will equal $\text{Set}(v)$.)

Following McCreight’s example [34], each node v will store the particular ordered pair (x, y) from $\text{SET}(v)$ which has maximal y -value but is not stored in any ancestor of v . This ordered pair is denoted as $\text{MAX}(v)$. The unique aspect of the Fusion Priority-Search tree is that it will contain the following three additional fields:

1. $\text{KEYS}(v)$: This will be a q*-heap describing the keys separating the ranges of v ’s children.
2. $\text{TOPS}(v)$: This will be a second q*-heap that stores the y-coordinates of the MAX elements belonging to v ’s children.
3. $\text{FUSEDTOPS}(v)$: Let $y_1, y_2 \dots y_B$ denote the distinct y-values stored in $\text{TOPS}(v)$ and r_i denote the rank of y_i in this subset. Then $\text{FUSEDTOPS}(v)$ is the ordered tuple $(r_1, r_2 \dots r_B)$. Since $\text{FUSEDTOPS}(v)$ can be encoded in $B \log B \leq O(\sqrt{\log N} \cdot \log \log N)$ bits, it can fit into one word of memory. (Recall that Section 2 indicated we always employ words with a bit length $\geq \log N$.)

Some notation is helpful to describe the search algorithm for the Fusion Priority-Search tree.

Given the ordered triple (a, b, c) associated with the query (1), say a node v is

- i) a *left borderline node* if $\text{Range}(v)$ intersects a but not b .
- ii) a *right borderline node* if $\text{Range}(v)$ intersects b but not a .
- iii) a *double borderline node* if $\text{Range}(v)$ intersects both a and b .
- iv) a *subsumed node* if $\text{Range}(v)$ is contained within the interval (a, b)
- v) a *pivotal node* if v is both subsumed and a child of a double borderline node.

Since a Fusion Priority-Search tree will have a $O(\log N / \log \log N)$ height and a $\sqrt{\log N}$ branching factor, it will have no more than $O(\log N / \log \log N)$ borderline and pivotal nodes. (This is because there can never be more than $\sqrt{\log N}$ pivotal nodes in a tree with $\sqrt{\log N}$ branching factor.) All these pivotal and borderline nodes can be found in $O(\log N / \log \log N)$ time by a trivial application of Corollary 1's "q*-heap" search procedure (see footnote² for the formal details). The first step of our retrieval algorithm will consist of such a search for the borderline and pivotal nodes. The second step of query (1)'s algorithm will examine the $\text{MAX}(v)$ elements of the nodes just visited to check whether they satisfy the query. It will output those elements which do.

Each remaining element in S that satisfies query (1) will be a $\text{MAX}(v)$ element for some subsumed node v . Since we desire to make our algorithm run in time $O(\log N / \log \log N + K)$ (where K is the size of the output), it is necessary to make the last step of our search find these elements in time proportional to $O(\log N / \log \log N + K)$. (This is a quite tight constraint that we are required to satisfy. It will require some care to achieve this objective.)

To obtain this run time, the third step of our search will repeatedly invoke a subroutine called LOOKAHEAD. Upon visiting a node v , this procedure will determine in constant time precisely

²The key idea behind this fast search procedure is that the q*-heap property allows us to trivially organize a node v 's data structure so that only $O(1)$ time is needed to find the children of v that are borderline when v is a borderline node. Therefore by a trivial repeated iteration of this functionality over the $O(\log N / \log \log N)$ different height levels, we can trivially find all the $O(\log N / \log \log N)$ borderline nodes in $O(\log N / \log \log N)$ time. Moreover since the Fusion Priority-Search tree has a $\sqrt{\log N}$ branching factor and since all its pivotal nodes will be the children of that particular double-borderline node which has the greatest depth, the same procedure will obviously only need $\sqrt{\log N}$ time to find the $\sqrt{\log N}$ or fewer pivotal nodes that exist. Hence, this footnote has displayed an $O(\log N / \log \log N)$ time procedure for finding all the pivotal and borderline nodes.

which of v 's children w are *subsumed* nodes that *simultaneously* store an element in their $\text{MAX}(w)$ fields that satisfies the query (1) *before it actually visits these elements!* . (It is *extremely* delicate and tricky to do this search in precisely $O(1)$ time because Query (1) is a query with three inequality constraints rather than the usual two inequalities associated with a conventional 1-dimensional range query condition. Since the tree node v will have $\sqrt{\log N}$ children, the procedure LOOKAHEAD must make strong use of the q^* -heap property to determine *in constant time* which of these $\sqrt{\log N}$ children contain data of interest *before they are even visited.*) The LOOKAHEAD procedure is easiest to describe if we let

- i) J_1 denote an integer indicating how many of v 's children are subsumed,
- ii) J_2 denote an integer indicating how many y -values in $\text{TOPS}(v)$ are greater than c .
- iii) J_3 denote an integer that equals zero if v is a left borderline node, one if it is a right borderline node, and two if it is subsumed.

Each of these three integers can be calculated in constant time, since J_1 can be discovered during a q^* -heap search of the $\text{KEYS}(v)$ field, J_2 discovered by a similar search through $\text{TOPS}(v)$, and J_3 's value will be known by the algorithm as soon as it enters the node v .

Now consider the 4-tuple $J^* = (J_1, J_2, J_3, \text{FUSEDTOPS}(v))$. This tuple can be encoded in $O(\sqrt{\log N} \cdot \log \log N)$ bits. Consequently, we can store all the possible values for this tuple in a precomputed look-up-table of size $o(N)$, where each table entry shall essentially encode a list of which subsumed children w of v have $\text{MAX}(w)$ data elements satisfying query (1). (More precisely, the particular entry in our table that is indexed by the 4-tuple $J^* = (J_1, J_2, J_3, \text{FUSEDTOPS}(v))$ can be thought of as containing a list of integers, I_1, I_2, \dots, I_t , such that the next nodes that should be visited by our top-down search are v 's $I_1 - th, I_2 - th, \dots, I_t - th$ children.) Since the lookup table can be built in $o(N)$ preprocessing time and occupies $o(N)$ space, its presence does not increase the data structure's overall memory space and preprocessing time. But yet it enables LOOKAHEAD to determine in $O(1)$ time which of v 's $O(\sqrt{\log N})$ children are subsumed nodes having MAX -values satisfying the query even before the time these elements are actually visited!

The key point about the preceding paragraph's lookup-table is that its $o(N)$ space and pre-processing costs are insignificant quantities because all the nodes in the Fusion Priority-Search tree will *use one common look-up table*, rendering its cost an acceptably small and trivial burden.

Define $\text{EXPLORE}(p)$ to be a 3-part procedure that first visits the node p , then invokes LOOKAHEAD to discover which children q of p are subsumed nodes with $\text{Max}(q)$ satisfying (1), and finally recursively calls itself to explore these children. The final step of query (1)'s search algorithm will invoke EXPLORE to probe the subtrees descending from the pivotal and the left and right borderline nodes. Each subsumed node whose $\text{MAX}(v)$ element satisfies (1) will lie in one such subtree, and all its ancestors in this subtree will also have their Max elements satisfy (1). These two conditions guarantee that EXPLORE will correctly find all the nodes whose Max elements satisfy (1). Moreover, EXPLORE requires only $O(\log N / \log \log N + K)$ time to process the K subsumed nodes it finds because of LOOKAHEAD 's $O(1)$ search property, combined with the fact that there are only $O(\log N / \log \log N)$ borderline and $O(\sqrt{\log N})$ pivotal nodes generated by the query (1). Hence, we have shown that all three steps of query (1)'s retrieval algorithm run in time $O(\log N / \log \log N + K)$.

Finally, we will show how to execute insertions and deletions in the preceding data structure in amortized time $O(\log N / \log \log N)$. We assume $B = \sqrt{\log N}$ and employ a Fusion Priority-Search tree where each node has arity between $B/8$ and B . We will use Lemma 1's algorithm for rebalancing the B-tree. It is immediate that constant time insertion and deletion operations are feasible in the $\text{KEYS}(v)$ and $\text{TOPS}(v)$ fields because they are q^* -heaps with $O(1)$ height. The similar $O(1)$ algorithm to update $\text{FUSEDTOPS}(v)$ consists of a 2-step procedure that first searches $\text{TOPS}(v)$ to determine in $O(1)$ time the rank of the particular element y_i (that is to be inserted or deleted) and then uses this rank, the integer i and the old value of $\text{FUSEDTOPS}(v)$ to determine $\text{FUSEDTOPS}(v)$'s new value via table look-up. (Once again, we note that the storage and pre-processing costs of the lookup tables are negligible expenses throughout this paper, since they are $o(N)$ costs.)

The natural algorithm for modifying a Fusion Priority-Search tree of height h will perform $O(h)$ search and update operations into the $\text{KEYS}(v)$, $\text{TOPS}(v)$ and $\text{FUSEDTOPS}(v)$ fields (plus some minor book-keeping work) when an insertion or deletion does not trigger a node split or

merge. It will thus consume $O(h)$ time when a split or merge does not occur. Each split and merge will consume $O(B)$ time. This data structure thus satisfies requirements of Lemma 1. Since $h < O(\log N / \log \log N)$, its amortized cost for insertions and deletions is thus $O(\log N / \log \log N)$, by Lemma 1. (Once again by Lemma 2, a strengthened version of this algorithm can also guarantee worst-case insertion and deletion time.)

This paragraph will explain how to use the Fusion Trees to improve one of Chazelle's data structures. His article [15] devised a data structure that occupies $O(N \log^{d-1} N / \log \log N)$ space and allows d -dimensional orthogonal reporting queries to run in time $O(\log^{d-1} N + K)$, where K is the number of elements reported [15]. The new Fusion Priority-Search trees allow us to revise Chazelle's orthogonal range query data structure so that a theoretically faster (but impractical) $O(\log^{d-1} N / \log \log N + K)$ reporting time prevails in the same memory space. For the case of the dimension $d = 2$, the new data structure and algorithm will be the same as Chazelle's except for the following four changes:

1. Wherever Chazelle's data structure had employed McCreight's Priority-Search trees, the new data structure will obviously utilize the new $\log \log N$ faster Fusion Priority-Search trees.
2. The branching factor of [15]'s base tree nodes will be changed from $\log N$ to $\log N / \log \log N$, thereby enabling one to traverse the set of children of a particular node in $\log \log N$ faster time.
3. Wherever an $O(\log N)$ time fractional cascade was previously employed, the new data structure will use a revised fractional cascade that has a fusion tree rather than a binary search tree as its index, thereby speeding up the index search by a $\log \log N$ factor.
4. The $O(\log N)$ time topdown tree walk in [15]'s base tree will be speeded up by a $\log \log N$ factor by having a q*-heap index the $\log N / \log \log N$ children of each node in this tree. (This is possible to do because the base tree has height $O(\log N / \log \log N)$ and every node has an arity $\log N / \log \log N$. Thus, using Corollary 1's q*-heap data structure, each such node can then be traversed in constant time. Therefore, the top-down tree walk therefore will run in time $O(\log N / \log \log N)$.)

It is immediate that this revision of Chazelle’s data structure supports 2-dimensional orthogonal reporting queries in time $O(\log N/\log \log N + K)$. The d -dimensional reporting time of $O(\log^{d-1} N/\log \log N + K)$ in space $O(N \log^{d-1} N / \log \log N)$ follows from [10]’s method of reducing d -dimensional queries to 2-dimensional ones. (All these results are obviously theoretical but not practical modifications of [15]’s procedure.)

The McCreight and Chazelle versions of Priority-Search trees have many uses in computer science [34]. The Fusion Priority-Search trees pertain to all such problems. For example, one can search a set of N rectangles and output all the K intersections among this set in time $O(K + N \log N/\log \log N)$.

6. Aggregation Queries

Let S denote a set of points in d -dimensional space. Each point-record r shall contain a special field denoted as $\text{VALUE}(r)$. Then a d -dimensional aggregation query is defined as a d -tuple (A_1, A_2, \dots, A_d) , representing a request to calculate $\sum \text{VALUE}(r)$ for those elements satisfying

$$\text{KEY} \cdot 1 < A_1 \wedge \text{KEY} \cdot 2 < A_2 \wedge \dots \wedge \text{KEY} \cdot d < A_d \quad (2)$$

This section will explain how to perform query (2) in time $O\{(\log N/\log \log N)^{d-1}\}$ when the data structure uses space $O(N \log^p N)$ with $p > d - 1$. The last paragraph of this section will define a certain sense in which our proposed algorithm can be regarded as optimal.

Our data structure will be a hybridization of fusion trees with the overlapping K -ranges of [12] and fractional cascading [18]. For simplicity, we will restrict our attention to the case of the dimension $d = 2$. The data structure will be a variant of range tree whose “**base**” section T will be a tree with $O(\log N/\log \log N)$ height, $\text{Polylog}(N)$ branching factor and which stores the elements of S at the tree’s leaf level in order of increasing $\text{KEY} \cdot 1$ value. Each internal node v of T will contain B auxiliary fields, denoted as $\text{AUX}(v, 1), \text{AUX}(v, 2) \dots \text{AUX}(v, B)$. The field $\text{AUX}(v, i)$ will describe the leaves descending from v ’s leftmost i sons arranged in order by increasing $\text{KEY} \cdot 2$ value. Let $r_1 r_2 r_3 \dots$ be the elements in $\text{AUX}(v, i)$, so arranged in order of increasing $\text{KEY} \cdot 2$ value. Then $\text{AUX}(v, i)$ will also store the aggregate quantities $\text{SUBTOTAL}(v, i, j) = \sum_{k=1}^j \text{VALUE}(r_k)$, for each $j \leq \text{CARDINALITY}(\text{AUX}(v, i))$.

A 2-dimensional orthogonal wedge query (a,b) is a request to retrieve $\sum \text{VALUE}(r)$ for the subset of elements from S satisfying

$$\text{KEY} \cdot 1 < a \wedge \text{KEY} \cdot 2 < b \tag{3}$$

It was implicit from the prior literature that the data structure from the previous paragraph could answer such a query by retrieving $O(\log N / \log \log N)$ subtotal counters. However, the difficulty was that there was no previously apparent computation method to find these counters in time $O(\log N / \log \log N)$.

This difficulty will be resolved by assigning a q^* -heap to each node v in the base tree. The q^* -heaps will enable a search algorithm to traverse each base tree node in constant time and locate in $O(\log N / \log \log N)$ time the $O(\log N / \log \log N)$ auxiliary fields that will need to be probed. The root's auxiliary field will next be searched in $O(\log N / \log \log N)$ time by using a fusion tree to index it. The remaining AUX-fields can be searched in constant time per field if we employ fractional cascading [18] to interconnect these fields. This algorithm easily generalizes to all dimensions $d \geq 2$, and it provides an $O\left\{\left(\frac{\log N}{\log \log N}\right)^{d-1}\right\}$ search algorithm for doing query (2) in $O(N \cdot \log^p N)$ space, for any $p > d - 1$.

Finally, we wish to define a partial sense in which the algorithms proposed in this section can be regarded as optimal. Let $L_d(N)$ denote the quantity $(\log N / \log \log N)^{d-1}$. Chazelle and Yao have studied time-space tradeoffs for aggregate queries using what is called the semi-group model of computation. Their goal has been to determine how many semi-group addition operations are needed to perform a query similar to (2) or (3). If a d -dimensional data structure occupies $O(N \cdot \log^p N)$ space for any $p > d - 1$, they have determined that the asymptote $L_d(N)$ constitutes both an upper and lower bound for the number of required semi-group addition operations. (Essentially, Yao published this result for the dimension $d = 2$ in 1985, and a more recent 1990 *JACM* article by Chazelle generalized Yao's result for higher dimensions [53, 16].)

Our computational model for doing the queries (2) or (3) is similar to that used by Chazelle and Yao, *except that we wish to be more conservative* by charging one unit cost for *both* each semi-group addition operation *as well as* for each other type of standard machine-language computer operation

that is needed to locate these semi-group aggregate counters. From the earlier work of Chazelle and Yao, we know that this model of computation certainly cannot calculate aggregates in a time better than their lower bound asymptotes of the form $L_d(N)$. The pleasing aspect of the algorithm outlined in this section is that the upper bound on its runtime will match the Chazelle and Yao lower bounds up to a constant factor when we assume the computer's machine language commands include the standard machine-language operations recognized by the literature on Fusion Trees.

7. Compressed van Emde Boas Trees

Van Emde Boas trees provide a facility to dynamically perform predecessor queries among a set S of N keys chosen from the universe $0, 1, 2, \dots, U - 1$ in worst-case time $O(\log \log U)$. The early variants of this data structure [46] had used $O(U)$ space, and Johnson [27] showed how the same result could also hold in $O(N \cdot U^\epsilon)$ space. The y-fast tries [47] with their modifications implied by Dietzfelbinger et. al. [19] establishes a $\log \log U$ time in $O(N)$ space, but they establish this speedup in an amortized expected rather than a strict worst-case model of time. This section will show that a $\log \log U$ combination of worst-case retrieval, insertion and deletion times are possible in $O(N)$ space provided that $N \geq U / (\log U)^{c \cdot \log \log U} = U \cdot 2^{-c(\log \log U)^2}$, for any fixed constant c . Without the use of the q-heap, the same $O(\log \log U)$ time in $O(N)$ space would only be possible when $N \geq U / \text{PolyLog}(U)$.

Let us begin by explaining how Van Emde Boas would derive the result in the sentence above. There are two published variants of van Emde Boas trees [46, 45]. The first variant [46] provides dynamic worst-case times $O(\log \log U)$ in space $O(U \log \log U)$. For some fixed constant K , the second variant of van Emde Boas trees [45] divides the initial set S into U/K subsets. Each subset S_i of the larger set S is associated with an integer $i \leq U/K$, and S_i describes the subset of S whose key values lie between $(i-1)K$ and $iK-1$. Let Z denote the set of integers i whose S_i sets are non-empty. Van Emde Boas's second data structure [45] was a 2-part structure whose upper fragment uses his earlier data structure to index the set Z and whose lower fragment is a forest of conventional balanced trees, where there is one tree representing each set S_i . Van Emde Boas [45] noted this structure would have $O(\log \log U + \log K)$ worst-case search, insertion and deletion times, and it would use

space $O[N + (U \log \log U)/K]$ (where N is the cardinality of the set S). Such constraints allow one to obtain $O[\log \log U]$ worst-case times for the basic retrieval and update operations over an $O(N)$ space structure when N satisfies $N \geq U/\text{PolyLog}(U)$ (assuming we set $K = U/N$).

Q*-heaps allow us to improve the result above. In particular the above retrieval and update times can be reduced to $O(\log \log U + \log K / \log \log N)$ when Corollary 2's q*-heaps replace the Balance trees in the preceding data structure. Then the revised Van Emde Boas tree will allow an $O[\log \log U]$ worst-case time for the basic retrieval and update operations over an $O(N)$ space structure when N satisfies the much weaker constraint $N \geq U / (\log U)^{c \cdot \log \log U} = U \cdot 2^{-c(\log \log U)^2}$ for any arbitrarily chosen constant c (assuming we again set $K = U/N$).

8. The ECDF Calculation

The section will describe a new algorithm to calculate a set of N d -dimensional ECDF statistics [10, 14] in linear space and time $O(N \log^{d-1} N / \log \log N)$. This result is a $\log \log N$ improvement over the Bentley-Shamos ECDF algorithm [10, 14]. The last paragraph of this brief section will explain why we conjecture our algorithm is optimal at least for the dimension $d = 2$.

Since our solution to [10, 14]'s ECDF problem will employ a data structure that naturally combines fusion trees with an earlier data structure devised by Dietz [20], we will provide only an abbreviated description of the new ECDF algorithm. Dietz developed an $O(U)$ space data structure for representing any subset of the integers $0,1,2,\dots,U$ which supports $\log U / \log \log U$ time for dynamic rank queries.

Consider a set S of N points in the plane. Let $x(p)$ and $y(p)$ denote the x and y coordinates of a point $p \in S$. Let $r(p)$ denote the rank of $x(p)$ among the set of x -coordinates. The first two steps of the new 2-dimensional ECDF algorithm will sort the set S twice (by x and y coordinate) and calculate the value of $r(p)$ for each $p \in S$. The third step will employ Dietz's data structure, where we now set the universe size $U = N$. It will take the sorted list that enumerates the elements of S by increasing y -value and insert the $r(p)$ attributes of these elements *one-by-one in order by their increasing y -value* into a Dietz data structure (whose search keys are its r -elements). Just

before the insertion of $r(p)$, the algorithm will calculate the quantity $e(p)$ that indicates $r(p)$'s rank *relative to the previously inserted* r -elements. The three steps of this algorithm run in $O(N)$ space and $O(N \log N / \log \log N)$ time. The derived quantities $e(p)$ correspond to [10, 14]'s 2-dimensional ECDF statistics.

For the case of the higher dimensions $d > 2$, we will essentially use [10]'s multidimensional divide-and-conquer method, adapted to Dietz's data structure and fusion trees, in a manner analogous to the preceding algorithm. That is, we will use [10]'s reduction method to reduce a d -dimensional search to a 2-dimensional ECDF search, and then apply the faster 2-dimensional ECDF algorithm outlined in the preceding paragraph.

It is unknown whether the preceding algorithm is *any sense* optimal for the dimension $d \geq 3$. Our algorithmic upper bound *almost matches* one of Chazelle's lower bound [17] when the dimension $d = 2$. The preceding sentence used the phrase "almost matches" because [17]'s computational model for batch-oriented lower bounds is similar but not identical to our model of computation for upper bounds. An interesting open question is whether or not Chazelle's Lower Bound Model could be extended to show that the 2-dimensional ECDF algorithm in this section is optimal.

9. Further Results and Open Questions

There are many examples of algorithms which have linear costs except for their use of sorting and searching. Fusion trees and/or the yet faster and more recent sorting and searching algorithms of especially Andersson, Hagerup, Nilsson, Raman and Thorup [4, 5, 42, 44] immediately imply speedups for such tasks. Some examples of these speedups include the construction of a convex hull, the testing for the equality, disjointness and containment relationships between two sets, the determination of the chromatic number of a permutation graph, etc. In particular, [5, 42]'s algorithm provides an $O(N \log \log N)$ randomized sorting time and [44]'s algorithm achieves a strict $O[N(\log \log N)^2]$ worst-case sorting time applicable for each of the four paradigm applications mentioned in this paragraph.

The main point is not the particular examples explored in the paragraph (above), or indeed

any of the examples discussed in this article. Rather it is that there are so many other similar examples and problems that can somehow be theoretically improved with one of the methodologies of [1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 24, 25, 35, 38, 39, 41, 42, 43, 44]. It should be stressed that we have focused mostly on a moderately narrow bandwidth of problems, drawn from Computational Geometry and Information Retrieval theory because such problems reflect the author's particular expertise and knowledge. If one merely examines Thorup's linear algorithm for resolving Dijkstra's famous Single Source Shortest Path problem [43], it is apparent that there will be many other problems which can be theoretically improved with the use of q-heaps and their many modifications.

One interesting open question is whether or not it is feasible to construct a Vornoi Diagram in faster than $N \log N$ time. Since a convex hull can be constructed in faster time using especially one of the new faster variants of fusion tree sorting algorithm [4, 5, 42, 44], it is natural to inquire whether the same may be somehow true for the Vornoi Diagram construction. Another interesting question is whether or not the original q-heaps of [25] can somehow have their $O(1)$ performance time characteristics extended beyond the mini-sets of $\text{PolyLog}(N)$ size, described by Section 3's Corollaries 1 and 2.

It is best to close this article by reviewing what we have accomplished, as well as what has certainly *not been done*. Section 1 had deliberately used the phrase "*so-called improvements*" to stress the fact that our theoretical asymptotic improvements are not accompanied with coefficients of practically small size. On the other hand for the sheer delight of the intellectual pleasure of exploration, it is curious that so many different problems in theoretical computer science can undergo asymptotic improvements after the application of fusion trees and q-heaps. (There are obviously many more of these one can find if one investigates the subject further.) Moreover although not practical, our even-theoretical discussion of Universal Hashing is thought-provoking because Dynamic Universal Hashing is a foundational problem.

Appendix: The Proof of Lemma 2

This Appendix will sketch a very tightly abbreviated proof of Lemma 2. Below is the formal statement of the proposition:

Lemma 2 For simplicity, let us assume that $B > 16$. Consider a B-tree whose internal nodes have arity between $B/8$ and B , whose root has arity between 2 and B , and whose leaves store the data and all have the same depth. Suppose that top-down searches in such a tree of height h will have an $O(h)$ cost, and the costs for adding or removing a single pointer from an internal node v are $O(1)$. Then regardless of the details of the structure of the node v , it is possible to devise an insertion and deletion algorithm for this tree that runs in a *worst-case time* $O(h)$, where the *O-notation's constant is independent of B*.

Proof Sketch. The previous literature has illustrated many examples of amortized optimization algorithms which can also guarantee worst-case time, if their procedures are made somewhat more elaborate and complicated. We will use a similar approach here to transform Lemma 1's amortize-optimizing procedure into an algorithm that also controls its worst-case running time. Our algorithm will be in many respects analogous to [52]. The discussion in this Appendix will therefore be brief. It may be helpful if the reader examines [52] at some juncture, if he wishes to see how one should fill in the precise details for the ideas that are intuitively sketched below:

Our B-tree will be quite conventional, in that all the data will be stored at the leaf level of the B-trees and all leaves will have the same depth. Say an internal node v is “**safe**” if its arity lies between $B/4$ and $3B/4$, and it is “**legal**” if its arity lies between $B/8$ and B . (Let us remember that $B > 16$, and that the tree root is allowed to “legally” contain between 2 and B children.) Our insertion/deletion algorithms will then guarantee that each node has legal arity at all times, and it will attempt to make the arity safe as often as possible.

The following notation will help us describe Lemma 2's insertion and deletion algorithms.

- a. An insertion (or deletion) operation will be said to **involve** a node v if it either inserts or deletes a leaf-record L that is either a descendent of v or a descendent of one of the two “adjacent”

siblings of v (that lie to its immediate left or right).

- b. An **evolutionary merge process** will be a procedure that gradually merges two adjacent B-tree sibling nodes, v and w , into one new node x . We will not allow the merge process to merge the two nodes, v and w , during one single unified action because such an operation would require $O(B)$ time and possibly cause our algorithm to lose its desired worst-case running time $O(h)$. Rather for some fixed pre-specified constant K , each **action** of our merge process will take K new pointers from v 's and w 's set of children and put copies of these pointers into the new node x that is gradually being constructed. Once the command to “merge” v and w is initiated, one such “action” will be performed by our insertion deletion algorithm during each insertion and deletion command that “involves” either v or w . Once the new node x is fully constructed, our merge algorithm will deallocate the nodes v and w and make x a new child of the former parent of v and w .
- c. An **evolutionary split process** will be the exact reverse of an evolutionary merge process. It will be a procedure that gradually splits one node x into two equal-sized nodes v and w by doing $O(K)$ units of work for each insertion-deletion command that “involves” x . That is, each operation will take K pointers from x and put copies of them into the new evolving nodes v and w .

Our algorithm for inserting and deleting new leaf-records into the B-tree will employ the above three constructs. It will be called $\text{Alg}(K)$. It will initiate an evolutionary split process whenever a node's arity becomes unsafely large by exceeding the arity $3B/4$. If a deletion causes a node v 's arity to fall below $B/4$ then $\text{Alg}(K)$ will do roughly the reverse action of the preceding sentence by essentially activating an evolutionary merge process (see footnote³ for the exact details). Also, all

³Deletions are conceptually similar to insertions, but their formal algorithm is painfully more complicated because there are several different sub-cases. The problem is that one might wish a tiny node v to merge with its sibling w but the latter cannot be done immediately because either w is currently too large or w is in the midst of another evolutionary split or merge that was previously invoked. If w is too large (say its arity is of size larger than $5B/8$) then v will initiate an evolutionary split to break w into two equal-sized halves (before v merges with the one of the newly produced halves of w). Similarly, if w is previously in the midst of another evolutionary split or merge when v wishes to merge with it, then v 's merge must wait until the latter is completed. (The reason our definition of “involvement” made mention of the “adjacent siblings” of v is that a process where an unsafe node v cannot repair itself until its sibling is fixed will work correctly only if we make certain that every insertion or deletion among the leaves descending from v causes K units of evolutionary repair to be done for **both** v and its two adjacent siblings.)

conventional B-tree algorithms, including $\text{Alg}(K)$, must be prepared for the possibility that the root r might possibly contain only one child s at the end of some deletion (or insertion) command. In this case, $\text{Alg}(K)$ will simply make s the new root of the B-tree and deallocate the node r (similar to conventional B-tree algorithms).

For the sake of brevity, the preceding paragraph's description of $\text{Alg}(K)$ was kept very short. The two natural questions a reader will ask about $\text{Alg}(K)$ are

1. What is its runtime?
2. Is $\text{Alg}(K)$ correct in the sense that it assures the B-tree is always legally balanced ?

The punch-line will be that it will be trivial to show that $\text{Alg}(K)$ always satisfies its claimed $O(h)$ worst-case time bound in a tree of height h , but its correctness will depend on one further delicate observation.

In particular, the answer to Question 1 is easy because each insertion or deletion of a leaf-record L in a tree of height h will “involve” no more than $3h$ nodes in the B-tree. (This is because only the ancestors y of L and y 's two adjacent siblings will be “involved”). Thus the evolutionary split processes of $\text{Alg}(K)$ will execute no more than $3h$ “actions”, each of which requires approximately K units of work. Hence, it will consume no more than worst-case time $O(h)$, where the coefficient hidden within the O -notation is proportional to $3K$.

Now let us turn to Question 2. Its full answer is complex because $\text{Alg}(K)$ will certainly not have the power to assure a B-tree is legally balanced if the pre-specified constant K has too small an initial value. However, it is basically trivial to verify that if the pre-specified constant K is sufficiently large (i.e. say $K > 100$) then the evolutionary processes used by $\text{Alg}(K)$ will be quick enough to repair a temporarily “unsafe” node v and its adjacent siblings (when necessary) to assure that the tree's balance never becomes so much worse as to be “illegal”. The latter is all we need to prove Lemma 2.

We will not delve into further details or describe the tedious formal encoding of the procedure $\text{Alg}(K)$ because many other papers have appeared in the prior literature about how an amortized-

optimizing algorithm can be transformed into a worst-case controlling procedure when one manipulates the algorithm's constant coefficient factor K with prudence (see for example [52, 50]). Our goal in this abbreviated appendix was only to sketch the intuition behind Lemma 2 very briefly. The reader can find much more sophisticated and interesting examples of amortized-to-worst-case transformations in [52].

Acknowledgement: I would like to thank both referees for their helpful comments.

References

- [1] S. Albers and T. Hagerup, Improved parallel integer sorting with concurrent writing, *Proceedings of the 3rd ACM-SIAM Symposium on Discrete Algorithms* (1992) pp. 463-472, journal version to appear in *Information and Computation*.
- [2] S. Alstrup, D. Harel, P. Lauridsen and M. Thorup, Dominators in Linear Time, submitted for publication.
- [3] A. Andersson, Sublogarithmic searching without multiplication, *Proceedings of the 36-th IEEE Symp. on Foundations of Computer Science* (1995) pp. 655-663, journal version to appear in *Journal of Computer and System Sciences*.
- [4] A. Andersson, Faster Deterministic Sorting and Searching in Linear Space, *Proceedings of the 37-th IEEE Symposium on Foundations of Computer Science* (1996), pp. 135-141.
- [5] A. Andersson, T. Hagerup, S. Nilsson and R. Raman, Sorting in Linear Time?, in *Proceedings of the 27-th ACM Symp. on Theory of Computing*, (1995) pp. 427-436.
- [6] A. Andersson, P. Mitlarsen, S. Riis and M. Thorup, Static Dictionaries on AC^0 RAMs: Query Time $\Theta(\sqrt{\text{Log}n / \text{LogLog}n})$ is Necessary and Sufficient, in *Proceedings of the 37-th IEEE Symposium on Foundations of Computer Science* (1996), pp. 441-450
- [7] A. Andersson, P. Mitlarsen and M. Thorup, Fusion Trees Can Be Implemented with AC^0 Instructions Only *Brics TR 96-30*, Aarhus (1996).
- [8] A. Andersson and K. Swanson, On the Difficulty of Range Searching, in *Proceedings of the 1995 Workshop on Algorithms and Data Structures* Volume 955 of Springer-Verlag LNCS, pp. 473-481.
- [9] P. Beame and F. Fich, On Sorted Lists: A Near-Optimal Lower Bound, (Technical Report issued 23 April 1997, available as preprint on web-page of University of Washington Computer Science Department).
- [10] J. Bentley, Multidimensional divide-and-conquer, *Comm. of ACM* 23 (1980), pp. 214-228.

- [11] A. Ben-Amram and Z. Galil, When can we sort in $o(N \log N)$ time? *Proceedings of the 34-th IEEE Symp. on Foundations of Computer Science* (1993) pp. 538-546.
- [12] J. Bentley and H. Mauer, Efficient worst-case data structures for range searching, *Acta Informatica* 13 (1980), pp. 155-168.
- [13] J. Bentley and M. McIlroy. Engineering a sort function. *Software-Practice and Experience* 23 (1993) pp. 1249-1265,
- [14] J. Bentley and M. Shamos, A problem in multi-variate statistics: algorithm, data structure and applications, *15th Allerton Conf. on Comm., Contr., and Comp.* (1977), pp. 193-201.
- [15] B. Chazelle, Filtering Search, A New Approach to Query Processing, *Siam Journal on Computing* 15 (1986) pp. 703-724.
- [16] B. Chazelle, Lower Bounds for Orthogonal Range Searching II - The Arithmetic Model *Journal of ACM* 37 (1990), pp. 439-463
- [17] B. Chazelle, Lower Bounds for Off-Line Range Searching *Proceedings of the 27-th ACM Symposium on Theory of Computing* (1995), pp. 733-740.
- [18] B. Chazelle and L. Guibas, Fractional Cascading: A Data Structuring Technique, *Algorithmica* 1 (1986), pp. 133-162.
- [19] M. Dietzfelbinger, A. Karlin, K. Melhorn, F. Meyer auf der Heide, H. Rohnert and R. Tarjan, Dynamic perfect hashing: Upper and lower bounds, *Proc. of the 29th IEEE Symp. on Foundations of Computer Science* , 1988, pp. 524-531; also: Tech. Report No. 282, Fachbereich Informatik, Universitat Dortmund, 1988.
- [20] P. Dietz, Optimal Algorithms for List Indexing and Subset Rank, *Workshop on Algorithms and Data Structures 1989*, published in volume 382 of Springer-Verlag LNCS pp 39-46.
- [21] M. Dietzfelbinger, F. Meyer auf der Heide, A new universal class of hash functions and dynamic hashing in real time *ICALP*, 1990, pp. 6-19.
- [22] H. Edelsbrunner, *Algorithms in Computational Geometry*, Springer-Verlag 1987.
- [23] M. Fredman, J. Komlos and E. Szemerdi, Storing a Sparse Table with $O(1)$ Worst-Case Access Time, *Journal of ACM* 31 (1984) pp. 539-544.
- [24] M. Fredman and D. Willard, Surpassing the Information Theoretic Barrier with Fusion Trees, *Journal of Computer and System Sciences* 47 (1993) pp. 424-436.
- [25] M. Fredman and D. Willard, Trans-dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths, *Journal of Computer and System Sciences* 48 (1994) pp. 533-551.
- [26] G. Gonnet, *Handbook of Algorithms and Data Structures*, Addison-Wesley, 1983

- [27] D. Johnson, A priority queue in which initialization and queue operations take $O(\log \log D)$ time, *Math. System Theory* 15 (1982), pp. 295-309.
- [28] R. Karlsson and J. I. Munro, Proximity on a Grid, *Second Annual Symposium (1985) on Theoretical Aspects of Computer Science* (Springer Verlag LNCS 182), pp. 187-196.
- [29] R. Karlsson, J. I. Munro and E. Robertson, The Nearest Neighbor Problem on Bounded Domains *The 12-th ICALP Symposium*, (1985), pp. 318-327.
- [30] D. Kirkpatrick and S. Reich, Upper Bounds for Sorting Integers on Random Access Machines, *Theoretical Computer Science* 28 (1984) pp. 263-276.
- [31] P. McIlroy, K. Bostic, and M. McIlroy. Engineering radix sort. *Computing Systems* 6 (1993) pp. 5-27,
- [32] K. Mehlhorn, Data Structures and Algorithms 1: Sorting and Searching, 1984, Springer-Verlag, Berlin.
- [33] K. Mehlhorn, Data Structures and Algorithms 3: MultiDimensional Searching and Computational Geometry, 1984, Springer-Verlag, Berlin.
- [34] E. McCreight, Priority Search Trees, *SIAM Journal on Computing*, 14 (1985), pp. 257-276.
- [35] P. Mitlerrsen, Lower Bounds on Static Dictionaries with Bit Operations But No Multiplication, in Proceedings of ICALP-1996, pp 213-225.
- [36] W. Paul and J. Simon, Decision Trees and Random Access Machines, *Proceedings of the Symposium on Logic and Algorithmic*, 1980, pp. 331-340.
- [37] F. Preperata and M. Shamos, *An Introduction to Computational Geometry*, Springer-Verlag, 1985.
- [38] R. Raman, Priority queues: small monotone and trans-dichotomous, *Proceedings of ESA'96* Springer Verlag LNCS 1136 (1996), pp. 121-137.
- [39] R. Raman, A Summary of Shortest Path Results. Kings College (London) TR 96-13 (1996).
- [40] A. Siegel, On universal classes of fast high performance hash function, their time-space tradeoff, and their applications, *Proceedings of 30-th Annual Symposium on Foundations of Computer Science*, 1989, pp. 20-25.
- [41] M. Thorup, On RAM Priority Queues, in the *Proceedings of the 7-th ACM-SIAM Symposium on Discrete Algorithms* (1996) pp. 59-67.
- [42] M. Thorup, Randomized sorting in $o(N \log \log N)$ time and linear space using addition, shift and bitwise operations, in the *Proceedings of the 8-th ACM-SIAM Symposium on Discrete Algorithms* (1997) pp. 352-359.

- [43] M. Thorup, Undirected single source shortest path in linear time, *Proceedings of 38-th Annual Symposium on Foundations of Computer Science*, (1997) pp. 12-21.
- [44] M. Thorup, Faster Deterministic Sorting and Priority Queues in Linear Space, *Proceedings of 9-th Annual ACM-SIAM Symposium on Discrete Algorithms*, (1998) pp. 550-556.
- [45] P. van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space, *Inform. Process. Lett.* 6 (1977), pp. 80-82.
- [46] P. van Emde Boas, R. Kaas and E. Zijlstra, Design and implementation of an efficient priority queue, *Math. Systems Theory* 10 (1977), pp. 99-127.
- [47] D. Willard, Log-logarithmic worst case range queries are possible in space $O(N)$, *Inform. Process. Lett.* 17 (1983), pp. 81-89.
- [48] D. Willard, New trie data structures which support very fast search operations, *J. Comput. System Sci.* 28 (1984), pp. 379-394.
- [49] D. Willard, Reduced Memory Space for Multi-Dimensional Search Trees, *2-nd Symposium on Theoretical Aspects of Computer Science* (published in Springer-Verlag LNCS 182), 1985, pp. 363-374. (The multi-branching B-tree appears in the last section of this paper.)
- [50] D. Willard, A Density Control Algorithm for Doing Insertions and Deletions in a Sequentially Ordered File in Good-Worst Case Time, *Information and Computation* 97 (1992) pp. 150-204.
- [51] D. Willard, Applications of the fusion tree method to computational geometry and searching, *Proceedings of the 3rd ACM-SIAM Symposium on Discrete Algorithms* (1992) pp. 386-395.
- [52] D. Willard and G. Lueker, Adding range restriction capability to dynamic data structures, *Journal of ACM* 32 (1985) pp. 597-619.
- [53] A. Yao, On the complexity of maintaining partial sums, *SIAM Journal on Computing*, 14(1985), pp. 277-289.