Exploiting Non-Uniform Query Distributions in Data Structuring Problems

by

John Howat

A thesis submitted to
the Faculty of Graduate and Postdoctoral Affairs
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

Carleton University
Ottawa, Ontario

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-93655-9

Our file Notre référence
ISBN: 978-0-494-93655-9

Abstract

This thesis examines data structures with query times that are a function of the distribution of queries made to them. When a query distribution exhibits non-uniformity—as is often the case in many applications—the sequence of queries can often be executed faster. Several such problems are considered.

For the dictionary problem, this thesis presents the first binary search tree to achieve the working-set property for individual queries in the worst case, a data structure that supports searching from arbitrary temporal fingers, and a data structure that supports a stronger version of the unified property.

For the predecessor search problem in bounded universes, this thesis presents a data structure that answers queries in time that is a function of the distance between the query and its answers (which leads to several applications in the areas of nearest neighbour search and range searching), as well as several data structures that answer queries in time that is a function of the entropy of the query distribution and various space requirements.

Acknowledgements

A PhD thesis does not happen in isolation.

I owe my supervisors, Prosenjit Bose and Pat Morin, a large debt of gratitude for their assistance during my studies. Their willingness to work with me and read countless drafts of papers has been incredible. I cannot imagine how many times I've knocked on their doors over the years and asked, "Got a second?"

My thanks are also due to the other members of my thesis committee: Paola Flochinni, Ian Munro, Michiel Smid, and Brett Stevens. Their feedback was invaluable.

I also gratefully acknowledge the generous funding I have received from Carleton University, the School of Computer Science, the Computational Geometry Laboratory, and the Natural Sciences and Engineering Research Council of Canada.

It has been a pleasure to be a part of the Computational Geometry Laboratory at Carleton University for the past five years. This lab would not be possible without Prosenjit Bose, Anil Maheshwari, Pat Morin, and Michiel Smid, as well as our many students and postdoctoral fellows (past and present). Our frequent seminars, open problem sessions, and social activities have all resulted in a happy and productive environment. The laboratory was lucky enough to host a number of visitors dur-

ing my time here. In particular, discussions with Rolf Fagerberg and John Iacono contributed to some parts of this thesis.

Like any good graduate student at Carleton, I spent a completely reasonable amount of time at Mike's Place. My sincere thanks are owed to my fellow graduate students in other departments for joining me there. In particular, I thank Matthew Meier, Tara Ogaick, and Elise Vist for helping me unwind after long days of research.

I am fortunate to have an incredibly supportive family. I can only begin to offer my thanks: my father, Robert Howat, for starting my journey with an old 386; my late mother, Judy Howat, for being my biggest advocate; my sister, Laurie Howat, for being my number one fan; my stepmother, Debbie Howat, for letting me talk through everything; and more extended family than I can enumerate. I could not have asked for more support.

Finally, I thank my wonderful wife, Bianca Howat, for helping me through graduate school in more ways than I suspect either of us realize. I would not have been able to complete this thesis without her love and support (and occasional interest in the Fibonacci numbers).

Contents

List of Figures

Chapter 1

Introduction

In this thesis, we present methods to exploit non-uniform query distributions in data structuring problems. Data structures that exhibit behaviour of this kind are typically termed *distribution-sensitive*. The idea of a query distribution being *non-uniform* is a deliberately vague notion; distribution-sensitive data structures can take advantage of many different types of underlying patterns in a query distribution, and we will elaborate on the specific types of patterns that are generally studied in Chapter 2.

Most distribution-sensitive data structures operate using the same high-level principle. For a given problem, a data structure that does not take advantage of the query distribution has a query time that is usually a function of the size of the data structure, and a (possibly matching) lower bound may also exist. Some quantity related to a property of the query distribution is defined and a new data structure is created such that its query time is a function of this quantity as opposed to the size of the data structure. Typically, this quantity, in the worst case, is no larger than the size of the data structure. As a result, any existing lower bounds are

not violated. However, in many cases, it is possible to beat known lower bounds, at least for some queries (or sequences of queries).

In this chapter, we motivate the use of distribution-sensitive data structures in Section 1.1. We then describe the problems to be addressed in this thesis in Section 1.2. The contributions of this thesis are outlined in Section 1.3. Most of these contributions have been or are in the process of being published; bibliographic details appear in Section 1.4. We conclude the introduction by outlining the organization of the remainder of the thesis in Section 1.5.

1.1 Motivation

Distribution-sensitive data structures are often motivated by practical concerns, since the types of queries handled in real-world applications are generally not uniformly random.

For example, imagine a file system on a server that handles a large number of users. One possible source of non-uniformity is that some files get accessed much more frequently than others (*e.g.*, programs essential to the operating system or commonly-accessed documents, such as a thesis), and an efficient file system may wish to speed up access to these files. The same could be said of users, as well. Users who log-in to the system infrequently (such as guest users) might have their files retrieved more slowly in order to speed up the access of more frequent users.

Another source of non-uniformity could be the relative location of files. During a directory traversal, for example, a user might be likely to access files that are, in some sense, close together (either physically on disk or abstractly in the directory

structure). If directory traversal is a common operation, speeding up such accesses would be beneficial.

Perhaps the clearest source of non-uniformity would be an explicit description of the distribution of queries to files. Each file would have some specified probability of being requested by a user. Naturally, we would expect frequently requested files to be accessed quickly, while files that are seldom requested could be accessed relatively slowly. Having complete information about the distribution of queries is, in general, a somewhat unrealistic assumption. Nevertheless, this information can be approximated empirically.

1.2 Problem Statements

This thesis addresses the following problems.

The Working-Set Property in Binary Search Trees. The working-set property roughly states that queries are fast when their answer has been reported recently. Several data structures have this property (as we shall see in Chapter 2), but those that have it even in the worst case (as opposed to the expected sense or amortized sense) do not belong to the binary search tree model of computation. Our goal in this line of research is to obtain the working-set property in the worst case in the binary search tree model. Since some binary search trees have the working-set property in the amortized sense, one can also view this problem as de-amortizing the working-set property in the binary search tree model.

Searching with Temporal Fingers. While the working-set property states that queries are fast when their answer has been reported recently, the queueish property conversely states that queries are fast when their answer has *not* been reported recently. We propose a generalization of these two properties: queries are fast when they are close to the element accessed a certain number of distinct queries ago (*i.e.*, the temporal finger). In this line of research, we desire data structures with running times that satisfy both the working-set and queueish properties, depending on the choice of temporal finger.

The Strong Unified Property. The unified property states that a query is fast if it is close to a recent query. Here, "close" refers to the rank distance among all elements stored in the data structure. We propose a strengthened version of this propery that allows us to ignore less-recent elements when measuring rank distance. Essentially, the distance we consider is the distance among recently-accessed elements instead of among all elements. In this line of research, our goal is to achieve data structures that satisfy this property.

Distance-Sensitive Predecessor Search in Bounded Universes. Searching for predecessors (or successors) is a fundamental problem in computer science. Given a set of elements drawn from a large, totally-ordered universe, we wish to know the largest element of the set that is less than or equal to (*i.e.*, the predecessor of) a given query drawn from the universe. In the case of bounded universes, there are some (known) bounds on the query time that are based on the size of the universe. This line of research seeks data structures that use the distance (measured as the difference between elements) between the query and the predecessor to bound the

time needed to perform a predecessor search as opposed to the size of the universe.

Biased Predecessor Search in Bounded Universes. This line of research returns to the problem of predecessor search, but instead seeks query times that are related to the query distribution itself: each element of the universe is associated with a probability that it is the query. The data structure is then tasked with performing a predecessor search in time that is a function of the probability of receiving that query.

1.3 Summary of Contributions

The following is a list of the contributions of this thesis. Each result is categorized into one of the five problems described in Section 1.2. Relevant theorems are included.

The Working-Set Property in Binary Search Trees

In Theorem 3.4, we establish the existence of the first binary search tree that has the working-set property on a per-query basis in the worst-case (instead of the amortized sense). This closes the gap between data structures that are binary search trees but only have the working-set property in the amortized sense and data structures that have this version of the working-set property but are not binary search trees.

Searching with Temporal Fingers

In Theorem 4.1, we describe a data structure that supports queries in time that is logarithmic in the distance between the query element and a pre-specified temporal finger, plus a small additive term. Despite this additional query overhead, it should be noted that by choosing the temporal finger that corresponds to the working-set property or the queueish property, we match the bounds of known data structures (*i.e.*, without this overhead). This is the first dictionary data structure to provide this temporal finger property.

The Strong Unified Property

In Theorem 5.1, we establish a data structure that comes to within a small additive term of achieving the strong unified property, a property we introduce in this thesis. The property roughly states that accesses are fast if they are close (among certain recently accessed elements) to a recently accessed element. This is the first data structure to strengthen the unified property in this manner.

Distance-Sensitive Predecessor Search in Bounded Universes

In Theorem 6.8, we describe a data structure for predecessor search in a bounded universe with a query time that is a function of the distance between the query and the element returned by the data structure, where the distance is the difference between the query element and its predecessor. The data structure supports insertions and deletions of elements in the same time bound.

Theorem 6.9 gives a data structure for the dynamic approximate nearest neigh-

bour problem in bounded universes. The query time of this data structure is a function of the Euclidean distance between the query point and the point returned. Insertions and deletions are also supported in the same time bound.

Theorems 6.10, 6.11 and 6.12 describe data structures for range searching in a bounded, two-dimensional universe. Supported query regions include quadrants (*i.e.*, dominance reporting), half-infinite regions (unbounded in one direction), and orthogonal range queries. The query times are a function of the dimensions of the region queried (plus the number of points contained in the region).

Biased Predecessor Search in Bounded Universes

In Theorem 7.2, we give a data structure for the predecessor search problem in a bounded universe whose expected query time is logarithmic in the entropy of the query distribution and with space that is asymptotically less than any fractional power of the universe size, but still super-linear in the number of elements stored.

Theorem 7.4 extends the previous result (at the cost of a small increase in the space) to provide a data structure whose expected query time is a function of an element's weight, where the weight of an element is a non-negative number specified at the time the data structure is constructed.

Theorem 7.5 achieves linear space for the same problem at the cost of a query time that is at most the square root of the entropy.

1.4 Bibliographic Notes

Most of the results in this thesis have been or are in the process of being published.

The results of Chapter 3 appeared in the Proceedings of the 9th Latin American Theoretical Informatics Symposium [14]. My coauthors were Prosenjit Bose, Karim Douïeb, and Vida Dujmović. This work has subsequently been published in *Algorithmica* [16].

The results of Chapter 6 appeared in the Proceedings of the 22nd Canadian Conference on Computational Geometry [15]. My coauthors were Prosenjit Bose, Karim Douïeb, Vida Dujmović, and Pat Morin. This work has subsequently been published in a special issue of *Computational Geometry: Theory and Applications* [17].

The results of Chapter 4 arose from discussions with Prosenjit Bose, and Pat Morin. The results of Chapter 5 arose from discussions with Prosenjit Bose, John Iacono, and Pat Morin. The results of Chapter 7 arose from discussions with Prosenjit Bose, Rolf Fagerberg, and Pat Morin.

1.5 Organization of the Thesis

The remainder of the thesis is organized in the following way.

Chapter 2 presents a review of the relevant literature and background used in the remainder of the thesis. We will describe the history of distribution-sensitive data structures, present properties of distributions that will be studied and survey several data structures. Note that problem-specific background is deferred until the relevant chapter.

Chapter 3 shows how to obtain a data structure that fits into the binary search tree model and has the working-set property even in the worst case. This unifies

previous results that obtain the working-set property in the worst case but are not in the binary search tree model with results that are in the binary search model but only guarantee the working-set property in the amortized sense.

Chapter 4 defines the notion of temporal fingers for the dictionary problem and proceeds to establish a data structure that achieves a query time that is logarithmic in this temporal distance to within a small additive term.

Chapter 5 defines a stronger version of the unified property and presents a data structure that achieves this property to within a small additive term.

Chapter 6 shows how to solve the predecessor search problem in bounded universes in time proportional to the distance between the query and the predecessor of the query.

Chapter 7 shows how to solve the predecessor search problem in bounded universes using a different metric: each element of the universe is assigned a probability and the query time is a function of the probability that element is the query.

The thesis concludes with Chapter 8 which summarizes the results and open problems contained in the thesis.

Chapter 2

Background

In this chapter, we present a summary of the literature relevant to distribution-sensitive data structures. As mentioned in the previous chapter, we defer problem-specific background until the relevant chapter.

We begin with a review of optimum binary search trees in Section 2.1. The notion of a "non-uniform" query distribution is then comprehensively addressed in Section 2.2. Armed with these definitions, we turn our attention to splay trees (which demonstrate several of these properties) in Section 2.3. We conclude our review of the literature with a survey of several other distribution-sensitive data structures in Section 2.4.

2.1 Optimum Binary Search Trees

Perhaps the earliest data structuring result that can be considered distribution-sensitive is that of *optimum binary search trees,* introduced in 1971 by Knuth [48]. An optimum binary search tree is a binary search tree on n keys that has *average*

search cost that is no larger than that of any other binary search tree built on those n keys. Note that finding such a tree is not trivial, and an exhaustive approach is hopeless since it is well-known that there are an exponential number of possible binary search trees on a given number of keys.

To construct such a tree, one defines a query probability for each key.[1] The construction algorithm uses dynamic programming to construct the tree in $O(n^2)$ time. The average search cost for a key drawn from the specified probability distribution is the sum (over all keys) of the probability associated with a key multiplied by the cost to search for that key in the tree (*i.e.*, the depth of the node containing that key).

It is important to note that the optimum binary search tree has the minimum average search cost over all binary search trees built on the given keys: there is no asymptotic notation required. This is, in general, a fairly lofty goal (as evidenced by the large amount of time required to construct the tree), and we typically settle for such results to within constant factors.

One important limitation of optimum binary search trees is that they require *a priori* knowledge of the query distribution, which (in many applications) may not be available.

2.2 Defining Non-Uniformity

Optimum binary search trees exploit the most obvious form of non-uniformity: explicit non-uniformity. In this section, we discuss other patterns and properties of

[1] One may also wish to consider the case when not all queries are stored in the tree (*i.e.*, some queries are unsuccessful). In this case, one also defines the probability that a query lies between two adjacent keys for each such pair.

queries that are commonly observed in data structuring problems. We defer examples of data structures with these properties until Section 2.4.

For simplicity, we describe these properties as they are used for *static dictionary* data structures. Such data structures maintain a static set $S = \{s_1, s_2, \ldots, s_n\}$ of n keys under the *search* operation, where we are given a key and must return the corresponding element in the data structure. The sequence of queries is given online and is denoted $A = \langle a_1, a_2, \ldots, a_m \rangle$, where $a_t \in S$ (for $1 \leq t \leq m$) and m is the length of the query sequence. We also assume a sufficiently long query sequence, so that m is $\Omega(n \log n)$. Throughout this thesis, we use $\log x$ to denote $\max\{1, \log_2 x\}$.

2.2.1 Static and Dynamic Optimality

Consider the *frequency* $f(x)$ of a query $x \in S$, *i.e.*, the number of times that query is made in A: $f(x) = |\{t \mid a_t = x\}|$. The *static optimality property* states that the time to execute A is

$$O\left(m + \sum_{i=1}^{n} f(x_i) \log(m/f(x_i))\right)$$

It may not be clear from where the "optimality" in "static optimality" is derived. To see this, we note that the quantity $f(x_i)/m$ is essentially the probability $p(x_i)$ that x_i is queried. The sum can therefore be re-written as

$$\sum_{i=1}^{n} f(x_i) \log(m/f(x_i)) = \sum_{i=1}^{n} mp(x_i) \log(1/p(x_i)) = m \sum_{i=1}^{n} p(x_i) \log(1/p(x_i)) = mH$$

where H is the empirical entropy of the query distribution.[2] The average query time is therefore $mH/m = H$. Since the entropy of the query distribution is a lower

[2]For this reason, the static optimality property is also known as the *entropy bound*.

bound on the average query time (to within lower-order terms) [59], it follows that a data structure with the static optimality property is optimal in this sense. The optimum binary search trees of Knuth [48] match this average query time. Note that, although the query time is stated in terms of query frequencies, it is not necessarily required that the data structure be given the frequencies. While optimum binary search trees require the frequencies to be specified in advance, not all data structures require this.

The *dynamic optimality property* enforces a much stronger sense of "optimality." Given some specific class of data structure (*e.g.*, binary search trees), let $OPT(A)$ denote the total query time of the fastest possible data structure in that class to execute the sequence A, given that it may perform arbitrary restructuring after each query (*e.g.*, rotations in a binary search tree), which are included in the access time. Another data structure of that class is *dynamically optimal* if it can execute A in time $O(OPT(A))$, assuming A is sufficiently long. Note that the optimum binary search trees of Knuth [48] do not necessarily fall into this category, because they do not take advantage of the possibility of restructuring after a query.

It may be helpful to consider some lower bound on $OPT(A)$. In binary search trees, for example, Wilber [66] presented two lower bounds on the time to execute an access sequence. Such results will not be required for this thesis, however, as we concentrate on upper bounds.

2.2.2 Key-Independent Optimality

Another kind of optimality is that of *key-independent optimality*, which was introduced by Iacono [42]. Let $b : S \to S$ be a random bijection and let $b(A)$ denote the

query sequence $\langle b(a_1), b(a_2), \ldots, b(a_m)\rangle$. Recall that $OPT(A)$ represents the fastest possible time that a data structure in a given class can execute A. Now, consider the quantity $E[OPT(b(A))]$: the expected value of the fastest time a data structure in a given class can execute $b(A)$. We say that a data structure is key-independently optimal if it can execute A in time $O(E[OPT(b(A))])$.

Intuitively, a key-independently optimal data structure executes sequences with random key values as fast as any other data structure in that class.

2.2.3 The Working-Set Property

In the query sequence $A = \langle a_1, a_2, \ldots, a_m\rangle$, we say that a_t is queried at time t for $1 \leq t \leq m$. We define the *working-set number* of x at time t, denoted $w_t(x)$, to be the number of *distinct* elements queried since the last time x was queried (*i.e.*, the last time x appeared in A). If x has not yet appeared in A (prior to time t), then we set $w_t(x) = n$. To be more precise, we slightly modify the notation of Iacono [42]: let $l_t(x) = \min(\{\infty\} \cup \{t' > 0 \mid a_{t-t'} = x\})$ and define

$$w_t(x) = \begin{cases} n & \text{if } l_t(x) = \infty \\ |\{a_{t-l_t(x)+1}, \ldots, a_t\}| & \text{otherwise} \end{cases}$$

The *working set property* states that the time required to execute A is

$$O\left(m + \sum_{t=1}^{m} \log w_t(a_t)\right)$$

Intuitively, query sequences that repeat particular queries frequently are executed faster in data structures with the working-set property. Sleator and Tarjan

[61] indicate one possible motivation for why this is a desirable property. Suppose that, while the set S is quite large, there exists a subset $S' \subset S$ such that all queries to the data structure belong to S'. In this case, the working-set property allows queries to run in $O(\log |S'|)$ time instead of $O(\log |S|)$ time, since the working-set number of any query at any time is at most $|S'|$. This essentially means that the elements of $S \setminus S'$ can be ignored, since they are never queried. Put another way, it is as if those elements are not present in the data structure.

This property was first discussed in this context by Sleator and Tarjan [61], although the general idea is similar to that of the move-to-front heuristic [60]. Interestingly, Iacono [40] proved that any data with the working-set property is also statically optimal. Therefore, ensuring the working-set property is one way of circumventing the need for *a priori* knowledge of the query distribution. Furthermore, Iacono [42] also showed that the bound provided by the working-set property is asymptotically equivalent to that provided by key-independent optimality. Therefore, any data structure that has the working-set property is also key-independently optimal and any data structure that is key-independently optimal has the working-set property.

2.2.4 The Queueish Property

In some sense, the queueish property is the opposite of the working-set property. The *queue number* of the element $x \in A$ at time t is denoted $q_t(x)$ and is defined to be $n - w_t(x) - 1$, *i.e.*, the number of distinct elements of S that have *not* been queried since the last time x was queried. The *queueish property* states that the time

required to execute A is

$$O\left(m + \sum_{t=1}^{m} \log q_t(a_t)\right)$$

Iacono and Langerman [45] proved that no binary search tree has the queueish property. As a result, this particular property has yet to be explored in significantly more detail, at least for the dictionary problem. A weaker version of this property called the *weakly queueish property* exists in at least one dictionary [45]. The weak version of the property states that a query a_t should execute in time $O(\log q_t(a_t)) + o(\log n)$.

2.2.5 The Static and Dynamic Finger Properties

While the working-set and queueish properties consider the *time* between queries, the static and dynamic finger properties consider the *distance* between queries. Let $d(x, y)$ denote the *rank distance* between x and y, *i.e.*, the number of elements in S between x and y.

We first consider the *static finger property*. Fix some particular element $f \in S$, which we shall call a *finger*. The static finger property states that the time required to execute A is

$$O\left(m + \sum_{t=1}^{m} \log d(f, a_t)\right)$$

The "static" in "static finger" arises from the fact that the finger f is fixed. Iacono [41] showed that any data structure that has the static optimality property also has the static finger property, for any choice of finger f.

The *dynamic finger property* allows the finger to dynamically change over time: the finger is always the previous query. Therefore, the dynamic finger property

states that the time required to execute the sequence A is

$$O\left(m + \sum_{t=2}^{m} \log d(a_{t-1}, a_t)\right)$$

2.2.6 The Unified Property

The *unified property*[3] is a combination of the working-set and dynamic finger properties and was introduced by Bădoiu et al. [20]. The unified bound $u_t(x)$ for a query x at time t is defined as follows

$$u_t(x) = \min_{y \in S}(w_t(y) + d(y, x))$$

Intuitively, $u_t(x)$ is small whenever a recently queried element is close to x. More formally, the unified property states that the time required to execute A is

$$O\left(m + \sum_{t=1}^{m} \log u_t(a_t)\right)$$

It is important to note that having the unified property is not the same as simultaneously having both the working-set and dynamic finger properties. To illustrate this, we consider an example of a query sequence due to Bădoiu et al. [20]. Suppose $S = \{1, 2, \ldots, n\}$ for some even n and define A as follows

$$A = \langle 1, n/2+1, 2, n/2+2, 3, n/2+3, \ldots, n/2, n, 1, n/2+1, \ldots \rangle$$

[3] As noted by Bădoiu et al. [20], a "unified bound" appears in the original paper on splay trees by Sleator and Tarjan [61], which is simply the minimum of the bounds provided by static optimality, the working-set property and the static finger property. The property we discuss here is distinct.

In A, every query (after the first n queries) will have working-set number n, and so the working-set property gives a bound of $O(m \log n)$ for the sequence. Similarly, every query is at distance $n/2$ from the previous query, and so the dynamic finger property gives a bound of $O(m \log n)$ for each sequence as well. However, observe that after the first n queries, query a_{t-2} is at distance 1 from a_t. The unified property therefore gives a bound of only $O(m)$ for the sequence, since both the working-set number and the distance are constant, which is a factor of $O(\log n)$ better than either of the other two bounds.

2.3 Splay Trees

The splay tree of Sleator and Tarjan [61] is among the first distribution-sensitive data structures and is certainly one of the most interesting. The reasons for this are two-fold: first, splay trees have a considerable number of distribution-sensitive properties, including most of those mentioned in Section 2.2. Perhaps more interesting is the fact there is still much to be shown about the behaviour of splay trees.

The splay tree fits into the usual binary search tree model (which we define precisely in Section 3.1.3), and the *splay* operation, which consists of a series of rotations, is used after every query. We omit the details of the splaying operation. Splay trees support queries in $O(\log n)$ *amortized* time; this means that individual queries may take as much as $O(n)$ time, but the sequence as a whole executes in $O(m \log n)$ time.

Splay trees have many of the properties mentioned in Section 2.2. Sleator and

Tarjan [61] showed that splay trees are statically optimal, have the static finger property and have the working-set property. Iacono [42] showed that splay trees are key-independently optimal.

Sleator and Tarjan conjectured that splay trees have the dynamic finger property; this was verified 15 years later with a fairly difficult proof due to Cole et al. [23, 24], although Tarjan showed that the less general sequence where the keys are queried in sequential order can be executed in $O(n)$ time [62]. This bound is known as the *scanning bound* or *sequential access theorem*.

Given that splay trees have both the working-set and dynamic finger properties, Bădoiu et al. [20] conjectured that splay trees also satisfy the unified property; this question is still open.

Perhaps the most important unresolved conjecture was also posed by Sleator and Tarjan in their original paper: are splay trees dynamically optimal? This problem remains open and is generally viewed as the most important question in this line of research. Note that this problem must, by its nature, be studied with respect to a particular class of data structure. For example, the problem is resolved for some classes of skip lists and B-trees [11].

2.4 Some Distribution-Sensitive Data Structures

In this section, we survey a selection of known distribution-sensitive data structures.

The Working-Set Structure. The working-set structure is due to Bădoiu et al. [20] and provides an even stronger version of the working-set property. Rather than executing the sequence in time $O(m + \sum_{t=1}^{m} \log w_t(a_t))$, the working-set structure

guarantees that each query a_t is answered in worst-case time $O(\log w_t(a_t))$ per query in the sequence. This can be viewed as de-amortizing the working-set property provided by, for example, splay trees. This means that any query can be answered in time $O(\log n)$, since $w_t(a_t) \leq n$ for all t; this is a significant improvement over the $O(n)$ guarantee offered by splay trees. The working-set structure is not a binary search tree, however. Instead, it is a collection of binary search trees and queues, each of which increases doubly exponentially in size to ensure that recently queried elements are found quickly. Therefore, the working-set structure is not directly comparable with splay trees.

The Unified Structure. The unified structure was also introduced by Bădoiu et al. [20] and was the first data structure to have the unified property. Like the working-set structure, the unified structure is not a binary search tree and is therefore not directly comparable with splay trees either. However, Derryberry [30] studied binary search trees that support the unified property.

Skip-Splay Trees. The skip-splay tree is due to Derryberry and Sleator [31] and is an attempt to achieve the unified property within the binary search tree model. Skip-splay trees fall slightly short of this goal: they execute the query sequence in time $O(m \log\log n + \sum_{t=1}^{m} \log u_t(a_t))$, and so spend an extra $O(\log\log n)$ amortized time per query.

Nearing Dynamic Optimality. Considering the apparent difficulty of the problem of proving splay trees to be dynamically optimal, attention has turned somewhat to finding other binary search trees that are *competitive* to dynamically optimal data

structures: their query times are a (sub-logarithmic) factor away from dynamic optimality. Several data structures come within a factor of $O(\log \log n)$ of dynamic optimality [13, 28, 65].

Finger Search. In the finger search problem, we are given a pointer to some element of the data structure before we perform a query for another element. The goal is then to execute the query in time that is a function (usually logarithmic) of d, where d is the rank distance from the query to the finger. This is a generalization of the dynamic finger property: to achieve the dynamic finger property, one can simply use a pointer to the previous answer. Several finger search data structures are known for various models (*e.g.*, [4, 19, 32]). Skip lists also allow for finger searches [57]. Kaporis et al. [47] show how to achieve $O(\log \log d)$ queries when the contents of S are chosen according to a certain kind of distribution.

Biased Search Trees. The biased search trees of Bent et al. [10] are similar in spirit to optimum binary search trees. Each element $s_i \in S$ is assigned a positive real weight w_i, and a query for s_i is executed in time $O(\log W/w_i)$, where $W = \sum_{i=1}^{n} w_i$ is the sum of all weights in the data structure. When w_i is viewed as the probability that s_i is queried, then we have $W = 1$ and achieve essentially the same as an optimum binary search tree (to within a constant factor). The key difference is that biased search trees support insertions and deletions of elements into and from S. Splay trees can also be made to support this query time, although only in the amortized sense. Treaps support this query time in the expected sense [58]. Bagchi et al. [6] presented a biased version of skip lists that can also match this query time.

Priority Queues. Johnson [46] described a priority queue that allowed for insertion and deletion in time $O(\log \log D)$, where D is the difference in priorities of the elements before and after the element to be inserted or deleted. In this setting, the allowable values of priorities come from a restricted range of integers. This type of distribution sensitivity is similar in spirit to that of the dynamic finger property, except that the measure of distance is in terms of priority and that the distance is measured to both the previous and next elements. Iacono [40] showed that pairing heaps offer a bound very similar to that of the working-set property, except in the priority queue setting. The minimum element can be extracted from a pairing heap in time $O(\log \min\{n, k\})$, where n is the number of elements in the pairing heap and k is the number of heap operations performed since x was inserted. Similar results were obtained by Elmasry [36].

Queaps and Queueish Dictionaries. Iacono and Langerman [45] presented a heap with the queueish property: the minimum element x can be extracted in time $O(\log k)$, where k is the number of elements that have been in the heap longer than x. A dictionary data structure is also presented that supports a query for x at time t in time $O(\log q_t(x) + \log \log n)$. Iacono and Langerman [45] also showed that no binary search tree can have the queueish or even weakly queueish property by showing an access sequence such that having the queueish property would violate the lower bound of Wilber [66].

The Temporal Precedence Problem. For the temporal precedence problem, a list must be maintained over many insertion operations. Queries consist of two pointers into the list and must quickly determine which of the two items pointed

to was inserted first. Brodal et al. [18] gave a data structure in the pure pointer machine model that is capable of answering queries in time $O(\log\log\delta)$, where δ is the number of insertions that occurred between the insertions of the query elements.

Random Input and Predecessor Search. The predecessor search problem asks for the largest element of S less than or equal to some query element. Belazzougui et al. [9] showed that this can be accomplished quickly when the set S is chosen according to a certain kind of distribution (as in the work of Kaporis et al. [47]). This differs from the usual notion of distribution sensitivity: rather than considering patterns in the query distribution, patterns in the input itself are considered.

Point Searching. For the (planar) point searching problem, the set S consists of points in the plane and a query consists of a point and the data structure must determine if that point is in the set S. If the point is not in the set S, then the data structure must indicate this. Demaine et al. [27] described how to answer point searching queries in a distribution-sensitive manner. The query time is logarithmic in a function related to the number of points in a certain region defined by the current and previous queries. Therefore, this data structure can be considered to have a two-dimensional analogue of the dynamic finger property.

Point Location. For the (planar) point location problem, we are given a series of regions in the plane. Queries consist of a query point and the data structure must determine which region contains the query point. This particular problem has been thoroughly studied and has had several distribution-sensitive data structures

proposed for it. One approach is to obtain an optimal data structure based on the probability distribution of the queries (*e.g.*, [5, 25]). In fact, it is possible to achieve close to this even without knowledge of the distribution [43]. Another approach to the problem is to support an analogue of the dynamic finger property, as proposed by Iacono and Langerman [44]. In this setting, the query time achieved is very similar to that achieved for point searching [27].

Nearest Neighbour Search. For the nearest neighbour problem, we are given a set of points in the plane, and a query asks for the closest point in the set to the query point. In general, it is difficult to find the exact solution quickly using a reasonable amount of space, and so most data structures settle for an approximate solution (*i.e.*, a point that is not much further away than the nearest neighbour). Derryberry et al. [29] proposed a data structure for the (approximate) nearest neighbour problem that executes queries in time that is a (logarithmic) function of the number of points in a certain box containing the query and the answer to the previous query. In fact, this technique works in any constant dimension; the query time has quadratic dependence on the dimension (and, of course, the approximation becomes worse as the dimension increases).

Biased Range Trees. Range searching is a very well-studied problem, but very little has been done in terms of distribution-sensitive data structures for range searching. Recall that for the range searching problem, we are given a set of points in the plane. A query consists of some region (typically of a specific shape) and we must report (or count) the number of points in the region. Dujmović et al. [34] described a data structure that, given a query distribution, can answer quarter-space

(*i.e.*, quadrant) queries in time that is optimal (to within a constant factor). Afshani et al. [2] subsequently extended this result to four-sided queries.

Chapter 3

The Working-Set Property in Binary Search Trees

In this chapter, we show how to construct a binary search tree, which we call a *layered working-set tree,* that satisfies a slightly stronger version of the working-set property. In particular, the time to execute an individual query is logarithmic in the working-set number of the query *in the worst case.* This differs from the working-set property defined in Section 2.2.3, since there we only require this bound for the *amortized* time of an individual query.

3.1 Problem Definition

Let S be a set of keys drawn from a totally ordered universe and denote a sequence of queries into this set by $A = \langle a_1, a_2, \ldots, a_m \rangle$, where $a_t \in S$. We wish to construct a binary search tree such that the query a_t can be executed in (worst case) time $O(\log w_t(a_t))$, where $w_t(a_t)$ is the working-set number of a_t at time t.

Our binary search tree will also support insertion and deletion. Let $S_t \subseteq S$ denote the set of keys stored in the binary search tree at time t. We allow elements of S to be inserted or deleted at time t in time $O(\log |S_t|)$.

3.1.1 Background

As discussed in Sections 2.3 and 2.4, the splay trees of Sleator and Tarjan [61] fall into the binary search tree model (to be defined in Section 3.1.3) and have the working-set property. However, individual queries may take as much as $\Theta(n)$ time to execute. The working-set structure of Bădoiu et al. [20], conversely, guarantees that an individual query can be executed in (worst case) time $O(\log w_t(a_t))$ (which implies that any query can be executed in time $O(\log |S_t|)$, since $w_t(a_t) \leq |S_t|$ for all t), but their structure is not a binary search tree. Therefore, all that remains is to unify these results by providing a binary search tree that guarantees a query can be executed in time logarithmic in the working-set number of the query in the worst case.

3.1.2 The Working-Set Structure

It will be useful to review how the working-set structure achieves its query time. In this section, we summarize the work of Bădoiu et al. [20].

The working-set structure, pictured in Figure 3.1 is composed of k balanced binary search trees (*e.g.*, AVL trees [1]) $T_1, T_2, \ldots, T_k$ along with k doubly-linked lists $Q_1, Q_2, \ldots, Q_k$. For all $1 \leq i \leq k$, the contents of T_i and Q_i are identical, and pointers are maintained between corresponding elements of T_i and Q_i. Each element in the set S_t is contained in exactly one tree and its corresponding list. For

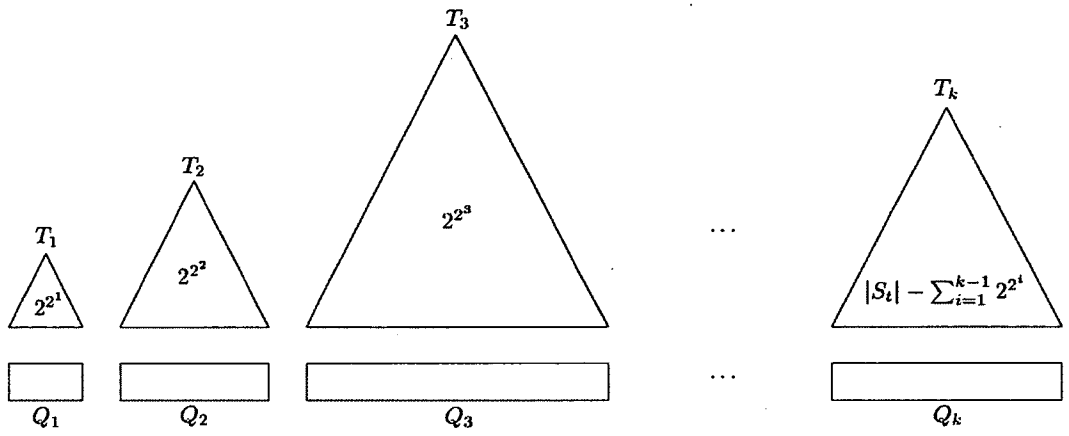


Figure 3.1: The working-set structure. Pointers between corresponding elements in the trees and linked lists have been omitted.

$i < k$, the size of T_i and Q_i is 2^{2^i}; the size of T_k and Q_k is the number of elements remaining, *i.e.*, $|S_t| - \sum_{i=1}^{k-1} 2^{2^i} < 2^{2^k}$. Note that since T_i is a balanced binary search tree, it can answer queries in $O\left(\log 2^{2^i}\right) = O(2^i)$ time. Furthermore, since there are a total of $|S_t|$ elements, we have that k is $\Theta(\log\log|S_t|)$.

The working-set structure works by keeping an element x in a low index tree and linked list when its working-set number is small and allows elements with large working-set numbers to fall back to the high index trees. More precisely, an element x with working-set number $w_t(x)$ at time t will be stored in a tree T_i with $i \leq \lceil \log\log w_t(x) \rceil$. In order to help ensure this, we will use the doubly-linked lists $Q_1, Q_2, \ldots, Q_k$. Every list Q_i orders the elements of its corresponding tree T_i in order of their last access: the youngest (*i.e.*, most recently accessed) element of T_i appears at the beginning of Q_i and the oldest (*i.e.*, least recently accessed) element of T_i appears at the end of Q_i.

The key operation used by the working-set structure is called a *shift*. A shift is performed between two trees T_i and T_j. Intuitively, a shift from T_i to T_j decreases

the size of T_i by one and increases the size of T_j by one. This will allow us to move elements around the data structure. A shift works in the following way. Assume that $i < j$ (the other case is symmetric). Look at the oldest element of Q_i (which is located at the end) and follow the pointer to the corresponding element in T_i. Remove the element from both Q_i and T_i (recall that deletions can be performed from the end of a doubly-linked list in constant time and from a balanced binary search tree of size 2^{2^i} in $O(2^i)$ time). This element is then inserted into T_{i+1} and Q_{i+1} (at the beginning, since this element is now the youngest element of Q_{i+1}). We then continue the shift from T_{i+1} to T_j. This process repeats until we attempt to shift from a tree to itself. Observe that the total time required to perform this shift is $O(2^{j-i}) = O(2^j)$.

We now describe how to perform a search in the working-set structure. During a query for element x, we sequentially query $T_1, T_2, \ldots$ until we either find x or search all trees and fail to find x. If we fail to find x, then we report so in total time $\sum_{i=1}^{k} 2^i = O(2^k) = O(\log |S_t|)$.[1] Otherwise, suppose we find x in T_i. Since T_i has size 2^{2^i}, the time spent to get to T_i is $O(2^i)$. We now remove x from T_i and Q_i and insert it into T_1 and Q_1. At this point, T_1 and Q_1 are one element too large, while T_i and Q_i are one element too small. To fix this, we perform a shift from 1 to i in time $O(2^i)$. The total time spent searching is thus $O(2^i)$.

The key observation is that if $x \in T_i$ for $i > 1$, then x must have been removed from Q_{i-1}. At the point it was removed from Q_{i-1}, x was the oldest element of Q_{i-1}, which contains a total of $2^{2^{i-1}}$ elements. Therefore, $w_t(x) \geq 2^{2^{i-1}}$, which implies that $i \leq 1 + \log\log w_t(x)$. The search time of $O(2^i)$ is thus $O(2^{\log\log w_t(x)}) = O(\log w_t(x))$,

[1] For simplicity, we say that the working-set number of any element not stored in a data structure is equal to the size of the data structure.

as required.

To insert x into the working-set structure, we insert it into T_1 and Q_1, which are now one element too large. To fix this, we shift from 1 to k in total time $O(2^k)$. Since k is $O(\log\log|S_t|)$, the insertion time is $O(\log|S_t|)$. To delete x from the working-set structure, we first search for it in time $O(\log w_t(x)) = O(\log|S_t|)$. Suppose we find x in T_i. We then remove x from T_i and Q_i in time $O(2^i)$ and perform a shift from k to i in time $O(2^k)$. In total, we spend $O(2^k) = O(\log|S_t|)$ time to delete x. Note that during an insertion, we may need to increment k and create a new tree if necessary. Conversely, during a deletion, we may need to decrement k and remove the last tree if it becomes empty.

3.1.3 Model

Until now, we have avoided precisely defining what it means to be a binary search tree. We use the model of Wilber [66]. Each node of the tree stores the key of S that is associated with it, and further maintains pointers to its left and right children, as well as its parent. Recall that the keys in the binary search tree are from a totally ordered universe. Nodes are ordered so that at any given node, all of the keys in the left subtree are less than that stored in the node and all of the keys in the right subtree are greater than that stored in the node.

Each node is permitted to store a constant[2] amount of additional information called *fields*, but no additional pointers may be stored.

To perform a query for a key, the key is *accessed* in the binary search tree. Ini-

[2]By standard convention, "constant" means asymptotically equivalent to the size of a key (which is typically logarithmic in the number of keys). Since each node must store a key, this only increases the storage at a node by a constant factor.

tially, we are given a finger to the root of the tree. An access consists of moving the pointer from a node to one of its adjacent nodes (through the parent pointer or one of the child pointers) until the pointer reaches the key being queried. Any time the pointer points to a node, we are allowed to modify the fields and pointers stored at that node at no cost. The total cost of an access (*i.e.*, the query time) is the total number of nodes pointed to by the pointer.

3.2 Definition of Layered Working-Set Trees

In this section, we define the structural portion of layered working-set trees. A layered working-set tree results from adapting the working-set structure into the binary search tree model.

3.2.1 Tree Decomposition

Let T denote our layered working-set tree. At a high level, one can view a layered working-set tree as layering the trees $T_1, T_2, \ldots, T_k$ of the working-set structure into a single binary search tree, while augmenting each node with sufficient information to determine the oldest element of a tree at a given time.

The first issue to address is precisely how to layer the trees of the working-set structure. To do this, we will label the nodes of T with a number from the set $\{1, 2, \ldots, k\}$ such that no element of T has an ancestor with a label that is strictly greater than its own label. Given a label i from the set $\{1, 2, \ldots, k\}$, we say that the set of all nodes of T with label i form a *layer* L_i. A layer L_i plays exactly the same role in T as the tree T_i in the working-set structure. In particular, layer L_i consists

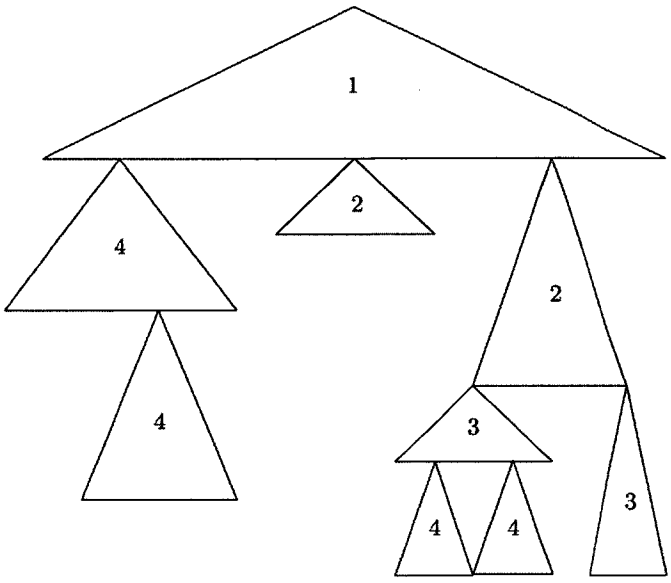


Figure 3.2: The decomposition of a layered working-set tree into layers. All triangles represent layer-subtrees, while all triangles with the same label form a layer.

of 2^{2^i} elements for all $i < k$, and L_k contains the remaining elements. However, L_i differs from T_i in that L_i is not a single tree: instead, L_i consists of a collection of subtrees in T. We refer to a connected component of nodes with equal labels as a *layer-subtree*. Figure 3.2 show an example of such a decomposition.

It is important to note that layer L_i can be connected to any layer $L_{i'}$ with $i' > i$. For example, one of the layer-subtrees below the topmost layer in Figure 3.2 has label 4. It is not always the case that labels increase by one over any root-to-leaf path.

Every node $x \in T$ will maintain a field that records the layer that x is contained in (*i.e.*, the value i such that $x \in L_i$). We shall denote this field's value at a node x by layer[x]. At the root of the tree, we will also maintain the number of layers k and the size $|L_k|$ of the deepest layer L_k.

Layer-subtrees are maintained independently as balanced binary search trees. The key requirement is that given a layer-subtree T', each node $x \in T'$ has depth $O(\log |T'|)$, measured from the root of T' (*i.e.*, the highest node of T'). Now, suppose that the layer-subtree T' is part of layer L_i. It therefore follows that the depth of any node in T' is at most $O(\log |L_i|)$. The independent maintenance of a layer-subtree means that whatever balance criteria are applied to ensure logarithmic depth are applied only within the layer-subtree. For our purposes, we maintain layer-subtrees as red-black trees [7, 38].

We begin with an observation about the depth of a node $x \in L_i$.

Observation 3.1. *The depth of a node $x \in L_i$ (measured in the tree as a whole) is $O(2^i)$.*

Proof. Consider the layers traversed on the path from the root to x. There are at most i such layers: $L_1, L_2, \ldots, L_i$. The path from the root to x traverses at most one layer-subtree in each of these layers. In layer $1 \leq j \leq i$, there are 2^{2^j} elements, and so the layer-subtree we pass through at this layer has size at most 2^{2^j}. Since each layer-subtree is maintained as a balanced binary search tree, each layer-subtree has logarithmic depth. Therefore, the total depth of x is $\sum_{j=1}^{i} O(2^j) = O(2^i)$. □

3.2.2 Encoding the Linked Lists

In the previous section, we have shown how to incorporate all trees of the working-set structure into a single binary search tree. It remains to show how we will encode the linked lists into the binary search tree model. For a summary of how the linked lists are encoded, refer to Figure 3.3.

Let L_i $(1 \leq i \leq k)$ be a layer of the tree. Each node $x \in L_i$ maintains a field that stores the key of the element in that layer inserted immediately before and immediately after it. This information is stored in the fields younger$[x]$ and older$[x]$, respectively. One can read younger$[x]$ as "the element inserted into L_i immediately *before* x," and older$[x]$ as "the element inserted into L_i immediately *after* x." If x is the most recently inserted element in its layer, then we assign younger$[x]$ = nil, and if x is the least recently inserted element in its layer, then we assign older$[x]$ = nil.

Each node $x \in L_i$ also maintains a field nextlayer$[x]$. If x is the most recently accessed element in a layer (*i.e.*, x is the youngest so that younger$[x]$ = nil), then nextlayer$[x]$ contains the key of the most recently accessed (*i.e.*, youngest) element in the next layer L_{i+1}. Conversely, if x is the least recently accessed element in a layer (*i.e.*, x is the oldest so that older$[x]$ = nil), then nextlayer$[x]$ contains the key of the least recently accessed (*i.e.*, oldest) element in the next layer L_{i+1}. If x is neither the youngest nor the oldest element in its layer, we set nextlayer$[x]$ = nil.

To ensure that we stay within the binary search tree model of computation described in Section 3.1.3, the fields of every node contain keys, and *not pointers*. Therefore, the operations we describe in the next section cannot use these fields to move around the tree. The only permissible use of these keys is to direct a subsequent search through the binary tree.

3.3 Operations on Layered Working-Set Trees

In this section, we define how operations are performed on layered working-set trees. The main obstacle in adapting the working-set structure operations into a

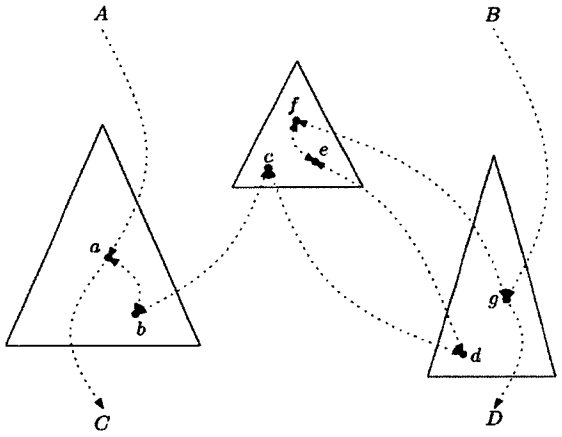


Figure 3.3: Encoding linked lists into a layered working-set tree. In this example, the elements (from most recently inserted (youngest) to least recently inserted (oldest)) are a, b, c, d, e, f, g. For example, $\mathsf{older}[b] = c$ and $\mathsf{younger}[f] = e$. A is the most recently inserted element in the previous layer and B is the least recently inserted element in the previous layer. C is the most recently inserted element in the next layer and D is the least recently inserted element in the next layer. For example, $\mathsf{nextlayer}[A] = a$ and $\mathsf{nextlayer}[g] = D$.

single binary search tree is that layers correspond to single trees in the working-set structure but to a series of trees in layered working-set trees. Layer-subtrees, however, are implemented as red-black trees and so the usual operations on red-black trees must be adapted to work between layers. We divide operations on layered working-set trees into those operating on a single layer (intra-layer operations), those spanning two adjacent layers (inter-layer operations), and finally those operating on the tree as a whole (tree operations).

3.3.1 Intra-Layer Operations

We refer to operations performed on a single layer as intra-layer operations. These operations correspond to those traditionally performed on any balanced binary

search tree. In particular, we need algorithms to maintain balance in layer-subtrees after insertion and deletion operations, as well as algorithms to perform splitting and joining of layer-subtrees to facilitate restructuring of layers.

For clarity, we describe how these operations can be achieved using red-black trees. This is not a requirement, however, and any binary search tree that meets the speed requirements for each operation can be used. It is also an important responsibility of layer-subtrees to ensure that their operations do not leave the layer-subtree in which they are originated; this can be achieved by checking the layer number (in the corresponding field) of a node before visiting it or performing any operation on it.

As indicated above, intra-layer operations rearrange layer-subtrees in some manner. At the boundaries of a layer-subtree (where a node has a different layer number than its child), there may be several layer-subtrees of different layers. Observe that the roots of each of these layer-subtrees can be viewed as the results of unsuccessful searches. In this sense, then, intra-layer operations on layer-subtrees are local: we need not concern ourselves with impacting any layer-subtree below the one we are performing an operation on.

Let T' denote a layer-subtree of layer L_i and let $x \in T'$. We define the following operations.

REBALANCE-INSERT(x) This operation restores logarithmic depth within T' after the node x has been inserted into this layer-subtree. For red-black trees, this operation is precisely the RB-INSERT-FIXUP operation presented by Cormen et al. [26, Section 13.3]. While the version presented there does not handle properly assigning a colour to x, it is straightforward to modify it to do so.

REBALANCE-DELETE(x) This operation is the counterpart to the previous operation and is responsible for ensuring logarithmic depth within T' after a deletion in T'. The node x to be given to this operation is dependent on the underlying structure used to implement layer-subtrees. For red-black trees, this operation is precisely the RB-DELETE-FIXUP operation presented by Cormen et al. [26, Section 13.4]. In this case, the node x is the child of the node spliced out by the deletion algorithm. We will elaborate on this when discussing inter-layer operations in Section 3.3.2.

SPLIT(x) This operation moves x to the root of T', which results in all other elements of that layer-subtree appearing as a descendant of either the left or the right child of x. A balance condition is also ensured, so that both the subtrees of the left and right children that are within the layer-subtree each have logarithmic depth. It may be the case that T', as a whole, is no longer balanced, however. For red-black trees, this operation is described by Tarjan [63, Chapter 4]. Note that, in our case, we do not destroy the original trees, but rather stop when x becomes the root of the layer-subtree.

JOIN(x) This operation is the inverse of the previous operation. Initially, we have a node x in a given layer and we wish to join it with its children (if they are in the same layer). We therefore must rebalance the layer-subtree so that each node has logarithmic depth. For red-black trees, this operation is described by Cormen et al. [26, Problem 13-2].

We summarize the results of these operations in Lemma 3.2.

Lemma 3.2. *For a node $x \in L_i$, the intra-layer operations* REBALANCE-INSERT(x), REBALANCE-DELETE(x), SPLIT(x) *and* JOIN(x) *can be implemented in such a way that they can be executed in time $O(2^i)$.*

Proof. By implementing layer-subtrees as red-black trees, the time bounds follow immediately from the arguments given by Cormen et al. [26] and Tarjan [63]. □

3.3.2 Inter-Layer Operations

In this section, we turn our attention to inter-layer operations. These operations facilitate structural changes between adjacent layers and correspond roughly to those used by the shift operation in the working-set structure. We defer discussion of running times until after the descriptions of the operations.

As in the previous section, we will describe the requirements of the operations independently of the actual layer-subtree implementation. Only the MOVEDOWN(x) operation will require knowledge of how layer-subtrees are implemented; the remaining operations simply use the operations defined in the previous section, allowing for maximum flexibility.

YOUNGESTINLAYER(L_i) This operation returns the key of the youngest (most recently accessed) element in the layer L_i. To accomplish this, we first exhaustively examine all elements in L_1. This exhaustive search is permissible for two reasons: first, this layer has size $O(1)$, and second, all elements in L_1 must be in the same layer-subtree and can thus be enumerated in time linear in the size of the layer. Once we find the element of L_1 that is the youngest (by looking for the element x_1 for which younger$[x_1]$ = nil), we go back to the root and search for nextlayer$[x_1]$,

which will return the key of the youngest element in L_2, say x_2. We then return to the root and search for nextlayer$[x_2]$, and so on. This process repeats until we find the youngest element in L_i, as desired.

OLDESTINLAYER(L_i) This operation returns the key of the oldest (least recently accessed) element in the layer L_i. This is accomplished in a manner identical to that of YOUNGESTINLAYER(L_i), except that the initial search in L_1 is for the oldest element (the element x_1 for which older$[x_1]$ = nil).

MOVEUP(x) This operation will move x from its current layer L_i to the next higher layer L_{i-1}. This corresponds to (one part of) a shift operation in the working-set structure in which we shift from a higher-index tree to a lower-index tree. To accomplish this, we begin by bringing x to the root of its layer-subtree by executing SPLIT(x). We then remove x from L_i and place it into L_{i-1} by setting layer$[x] = i-1$, which creates two new layer-subtrees in L_i: one for each child of x. We must now ensure balance in both the layer-subtree x was removed from in L_i and the layer-subtree x was inserted into in L_{i-1}. Observe that, by the definition of the split operation in the previous section, both of the layer-subtrees that are rooted at the children of x are already balanced. Therefore, we need only ensure balance in the layer-subtree in which x was inserted. This can be accomplished by executing the intra-layer operation REBALANCE-INSERT(x).

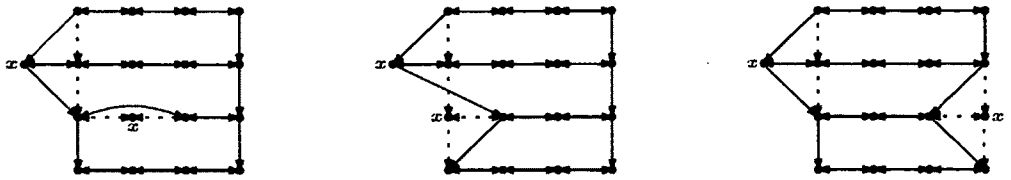
The final step is to update the encoding of the linked lists. To accomplish this, we examine the fields associated with x. If neither older$[x]$ nor younger$[x]$ are nil, then we go back to the root and perform searches for older$[x]$ and younger$[x]$. When we get to those nodes, we set younger[older$[x]$] = younger$[x]$ and older[younger$[x]$] =

older[x], essentially "splicing out" x from the linked list encoding in layer L_i, which means x has been removed from the linked list encoding for L_i. It remains to insert x into the linked list encoding for layer L_{i-1}. To do this, we note that x should now be the youngest element of L_{i-1}, and so we find the youngest element in L_{i-1}, say y, by executing YOUNGESTINLAYER(L_{i-1}). We then search for y and set younger[x] = nil, older[x] = y, and younger[y] = x. Note that, since y was the youngest element in L_{i-1}, we must update nextlayer[x] = nextlayer[y] and set nextlayer[y] = nil. The final step is to update the field stored by the youngest element in L_{i-2}, say z, by executing YOUNGESTINLAYER(L_{i-2}), searching for z, and setting nextlayer[z] = x.

Otherwise, if younger[x] is nil but older[x] is not, then we conclude x was the youngest element of L_i, and so older[x] should be the new youngest element in L_i. We therefore go to the root and search for older[x] and set younger[older[x]] = nil. Since older[x] is now the youngest element in L_i, we also copy nextlayer[x] into nextlayer[older[x]]. At this point, we have removed x from the linked list encoding in layer L_i and must now insert it into the linked list encoding for layer L_{i-1}. This can be achieved exactly as in the previous case.

The final case occurs if older[x] is nil but younger[x] is not. If this happens, then we conclude x was the oldest element of L_i and can proceed in a manner symmetric to the previous case. All three cases are illustrated in Figure 3.4.

MOVEDOWN(x) This operation will move x from its current layer L_i to the next lower layer L_{i+1}. In this sense, it is the opposite of the MOVEUP(x) operation and corresponds to (one part of) a shift operation in the working-set structure in which we shift from a lower-index tree to a higher-index tree. This operation is dependent on how layer-subtrees are implemented; we describe it in terms of red-black trees.



(a) **(b)** **(c)**

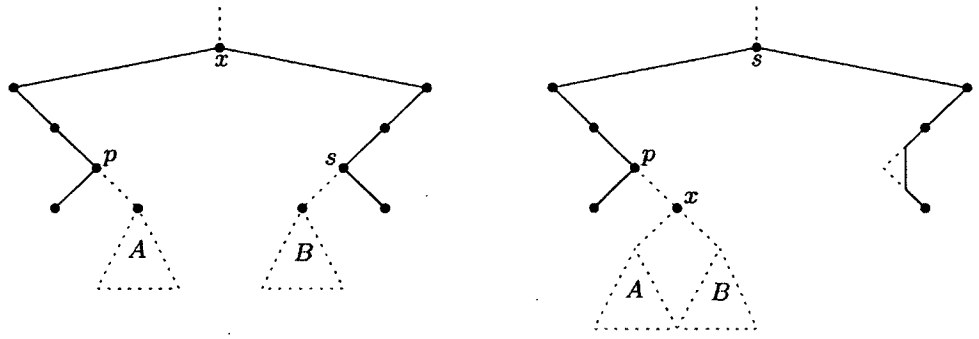
Figure 3.4: The MOVEUP(x) operation in a layered working-set tree. Each row represents a layer. The elements are sorted left to right from youngest to oldest. The layers, from top to bottom, are L_{i-2}, L_{i-1}, L_i, and L_{i+1}. In all cases, the new position of x after the operation is the leftmost (youngest) position in L_{i-1}. The arrows indicate what keys are referred to in the fields of nodes. Dotted arrows indicate the field has been overwritten during the operation. The lower x is the old position of x and the higher x is the new position of x. **(a)** x is neither the youngest nor the oldest element in L_i. **(b)** x is the youngest element of L_i. **(c)** x is the oldest element of L_i.

Let p denote the predecessor of x in L_i, *i.e.*, the largest element of L_i that is less than x. If x is the smallest element of L_i, then set $p = x$. Similarly, let s denote the successor of x in L_i, *i.e.*, the smallest element of L_i that is greater than x. If x is the largest element of L_i, then let $s = x$.

Our first goal is to move x such that it becomes a leaf of its layer-subtree. There are three cases to consider, based on the number of children x has that are in the same layer (and thus the same layer-subtree). If x has no children in its layer-subtree, then it is a leaf of its layer-subtree and we are done.

If x has only one child s', then to make x a leaf of its layer-subtree, we splice out x by attaching its left (respectively right) child s' as a parent of x and attaching x as a right (respectively left) child of p (respectively s).

If x has two children in its layer-subtree, then to make x a leaf of its layer-



(a) (b)

Figure 3.5: The MOVEDOWN(x) operation in a layered working-set tree. In this case, x has two children in its layer-subtree. The dotted lines to nodes and subtrees indicate layer boundaries. **(a)** The initial layer-subtree containing x. **(b)** The layer-subtree after the nodes have been moved and layers changed, but before rebalancing.

subtree, we splice out the node s by making the parent of s point to the right child of s instead of s itself. This is permissible because s is a descendant of x and because s has no left child in L_i, since it is the smallest element greater than x. We then move s to the location of x. Finally, we make x a child of p and make the new children of x the old children of p and s. These operations can all be accomplished by merely changing fields of these nodes. Figure 3.5 explains the process of making x a leaf of its layer-subtree for this case.

Observe that x is now a leaf of its layer-subtree and that the layer-subtree is configured exactly as if we had deleted x using the deletion operation described by Cormen et al. [26, Section 13.4]. Therefore, we can perform REBALANCE-DELETE(s'), where s' is the (only) child of the node we have spliced out during deletion, as required by the description of the operation.

To complete the movement of x to the next layer, we change the layer number of

x accordingly and execute JOIN(x) to create a balanced layer-subtree from x and its children. It is critical to note that if the children of x have larger layer numbers than the new layer number of x, the join operation has no effect and x becomes the lone element in a new layer-subtree; this follows from the fact that the join operation only operates on nodes of the same layer.

We must also update the linked lists encoded at the appropriate layers. This can be accomplished in essentially the same way as was done for the MOVEUP(x) operation.

We summarize the results of these operations in Lemma 3.3.

Lemma 3.3. *For any layer L_i, the intra-layer operations* YOUNGESTINLAYER(L_i) *and* OLDESTINLAYER(L_i) *can be implemented in such a way that they can be executed in time $O(2^i)$. For any node $x \in L_i$, the intra-layer operations* MOVEUP(x) *and* MOVEDOWN(x) *can be implemented in such a way that they can be executed in time $O(2^i)$.*

Proof. The operations YOUNGESTINLAYER(L_i) and OLDESTINLAYER(L_i) proceed to find the youngest (respectively oldest) element in layers $L_1, L_2, \ldots, L_i$. Given the youngest (respectively oldest) element in layer L_j, we can determine the youngest (respectively oldest) element in layer L_{j+1} in constant time since such an element maintains the key of the youngest (respectively oldest) element in the next layer. We then need to traverse from the root to that element. By Lemma 3.1, the total time is thus $\sum_{j=1}^{i} O(2^j) = O(2^i)$.

The operations MOVEDOWN(x) and MOVEUP(x) consist of searching for x, performing a constant number of intra-layer operations and then making a series of queries for the youngest elements in several layers and updating the linked list en-

codings. The search can be done in time $O(2^i)$ by Lemma 3.1 and the intra-layer operations each take time $O(2^i)$ by Lemma 3.2. Finally, the queries for the youngest elements and the time spent updating the linked lists is dominated by the time of the query in the deepest layer, since each has size the square of the previous one. Since $x \in L_i$, the time is $O(2^i)$ by the previous argument. The total time required for MOVEUP(x) and MOVEDOWN(x) is thus $O(2^i)$. □

3.3.3 Tree Operations

Armed with our intra-layer and inter-layer operations, we are now ready to describe how to perform the tree operations SEARCH(x), INSERT(x) and DELETE(x) on the layered working-set tree as a whole. Tree operations are independent of the layer-subtree implementation given suitable implementations of the intra- and inter-layer operations already defined in the previous sections.

SEARCH(x) To perform a search for x, we begin by searching the binary search tree T in the usual way. Once we have found $x \in L_i$, we execute MOVEUP(x) a total of $i - 1$ times in order to bring x into L_1. We then restore the sizes of the layers as was done in the working-set structure. We execute OLDESTINLAYER(L_1) to find the oldest element in L_1, say x_1, and then run MOVEDOWN(x_1). We then perform the same operation in L_2 by running OLDESTINLAYER(L_2) to find the oldest element in L_2, say x_2, and then run MOVEDOWN(x_2). This process of moving elements down layer-by-layer continues until we reach a layer L_j such that $|L_j| < 2^{2^j}$, at which point we stop.[3] Note that efficiency can be improved by remembering the

[3]In an ordinary search, we have $i = j$. However, thinking of the algorithm this way gives us a cleaner way to describe the insertion process later.

oldest elements of previous layers instead of finding the oldest element in each of $L_1, \ldots, L_i$ during the second stage of the algorithm. However, such an improvement does not alter the asymptotic running time.

INSERT(x) To insert x into T, we first examine the index k and size $|L_k|$ of the deepest layer of T (recall that k and $|L_k|$ are stored at the root of T). If $|L_k| = 2^{2^k}$, then we increment k and set $|L_k| = 1$. Otherwise, if $|L_k| < 2^{2^k}$, we simply increment $|L_k|$. We now insert x into T (ignoring layers for now) using the usual algorithm where x is inserted at the point the search path for x terminates. We set layer$[x] = k + 1$ (*i.e.*, a temporary layer larger than any other) and update the linked list encoding in the same manner as described in Section 3.3.2. Finally, we run SEARCH(x) to bring x into L_1. Note that since SEARCH(x) stops moving down elements once the first non-full layer is reached, we do not place another element in layer $k+1$. Thus, this layer is now empty and we update the youngest and oldest elements in layer t to indicate that there is no layer below it.

DELETE(x) To delete x from T, we first examine the number of layers k, which is stored at the root. We then locate $x \in L_i$ and perform MOVEDOWN(x) a total of $k - i + 1$ times. This will cause x to be moved to a new (temporary) layer that is guaranteed to have no other nodes in it. Therefore, x must be a leaf of the tree, and we can simply remove it by setting the corresponding child pointer of its parent to nil. As was the case for insertion, this temporary layer is now empty and can be removed. We must now restore the size of layer i. To do this, we execute YOUNGESTINLAYER(L_k) to determine the youngest element in L_k, say x_k. We then execute MOVEUP(x_k) to increase the size of L_{k-1} by one. We then execute

YOUNGESTINLAYER(L_{k-1}) to determine the youngest element in L_{k-1}, say x_{k-1}, and execute MOVEUP(x_{k-1}) to increase the size of L_{k-2}, and so on, until we reach L_i. At this point, all layers have the correct size. It could now be the case that $|L_k| = 0$. If this happens, we decrement the number of layers k, which is stored at the root, and update the youngest and oldest elements in the new deepest layer to indicate that there is no layer below.

The main contribution of this chapter is

Theorem 3.4. *There exists a binary search tree on n elements that supports a search for key x at time t in worst-case time $O(\log w_t(x))$ (where $w_t(x)$ is the working-set number of x at time t), as well as insertions and deletions in time $O(\log n)$.*

Proof. A search consists of a regular search in a binary tree followed by several layer operations. Suppose $x \in L_i$ at time t. By Lemma 3.1, we can find x in time $O(2^i)$. We then perform MOVEUP(x) in time $O(2^i)$ by Lemma 3.3. We then run a constant number of inter-layer operations for every layer from 1 to i. By Lemma 3.3, this takes total time $\sum_{j=1}^{i} O(2^j) = O(2^i)$. The total time to find $x \in L_i$ is therefore $O(2^i)$. By the same analysis as that of the working-set structure by Bădoiu et al. [20], we have $w_t(x) \geq 2^{2^{i-1}}$, and so $O(2^i) = O(\log w_t(x))$.

An insertion consists of traversing through all layers. By Lemma 3.1, this takes time $\sum_{i=1}^{k} O(2^i) = O(2^k) = O(2^{\log\log|S_t|}) = O(\log|S_t|)$. We then perform a search in time $O(\log n)$ by the previous argument, since the element searched for is in the deepest layer. The total time is therefore $O(\log n)$.

A deletion consists of searching the tree to find $x \in L_i$ and then performing a constant number of inter-layer operations per layer. The initial search takes $O(2^i)$ time by Lemma 3.1 and the inter-layer operations take $\sum_{j=1}^{k} O(2^j) = O(2^k)$ by

Lemma 3.3 for a total time of $O(2^k) = O(\log n)$. □

3.4 Conclusion and Open Problems

In this chapter, we have developed the first binary search tree that answers queries in time logarithmic in the query's working-set number even in the worst case. This unifies the results of Sleator and Tarjan [61], who showed that one binary search tree (namely the splay tree) has this property in the amortized sense with the results of Bădoiu et al. [20], who demonstrated that this property exists in data structures that are not binary search trees. This result can be viewed as de-amortizing the working-set property in the binary search tree model.

There are several possible directions for future research.

1. It seems that by enforcing this good worst-case behaviour for temporally local query distributions, we sacrifice good performance on some other access sequences. Is it the case that a binary search tree that has this stronger version of the working-set property cannot achieve other properties of, *e.g.*, splay trees? For example, what kind of sequential access bound can be achieved in this setting? Recall that the scanning bound refers to the amount of time required to execute all queries in sequential order.

2. Are similar results possible for the dynamic finger property? Is it possible to answer a query in worst-case time $O(\log |a_t - a_{t-1}|)$? As with the working-set property, this is well-known outside of the binary search tree model.

Chapter 4

Searching with Temporal Fingers

In this chapter, we show how to construct a data structure that supports query times that are logarithmic in the distance from the query element to a temporal finger, plus a small additive term.

4.1 Problem Definition

As in Chapter 3, we consider a set S and a sequence $A = \langle a_1, a_2, \ldots, a_m \rangle$, where $a_t \in S$. For the purposes of this chapter, we assume $S = \{1, 2, \ldots, n\}$ (which is simply a reduction to rank-space) and that the set S is static.

Recall that the working-set property roughly states that accesses are fast if they have been made recently. Conversely, the queueish property states that accesses are fast if they have *not* been made recently. We propose a generalization of these two properties, where a *temporal finger* is defined and the access time is a function of the distance in A between the temporal finger and the element to be accessed. Such a property can be viewed as a temporal version of the static finger property.

4.1.1 Defining Temporal Distance

We briefly review some definitions from Section 2.2.3. Recall that our access sequence is $A = \langle a_1, a_2, \ldots, a_m \rangle$. Define

$$l_t(x) = \min\left(\{\infty\} \cup \{t' > 0 \mid a_{t-t'} = x\}\right)$$

One can think of $l_t(x)$ as the most recent time x has been queried in A before time t. We then define

$$w_t(x) = \begin{cases} n & \text{if } l_t(x) = \infty \\ |\{a_{t-l_t(x)+1}, \ldots, a_t\}| & \text{otherwise} \end{cases}$$

Here, $w_t(x)$ is the usual working-set number of element x at time t. We are now ready to define temporal distance. Consider a temporal finger f where $1 \leq f \leq n$. The temporal distance from x to f at time t is defined as

$$\tau_{t,f}(x) = |w_t(x) - f + 1|$$

Observe that if $f = 1$, then $\tau_{t,f}(x) = w_t(x)$. In this case, having query time logarithmic in the temporal distance is equivalent to the working-set property. Conversely, if $f = n$, then $\tau_{t,f}(x) = n - w_t(x) - 1$, which is precisely the queue number $q_t(x)$ defined by Iacono and Langerman [45] and described in Section 2.2.4. Performing a query in time logarithmic in the temporal distance in this case is equivalent to the queueish property. In general, our goal is a query time of $O(\log \tau_{t,f}(x)) + o(\log n)$.

Another way to view temporal distance is the following. At any time t, the query sequence A defines a permutation of the elements that orders them from most recent query to least recent query. The temporal finger f points to the f-th item in this permutation, and the temporal distance $\tau_{i,f}(x)$ is the distance from the f-th item to x in the permutation.

4.1.2 Background

The notion of a dictionary with query times that are sensitive to temporal distance is not new: the working-set structure [20] and the queueish dictionary [45] are two well-known examples. The layered working-set tree of Chapter 3 also falls into this category.

However, the notion of allowing the finger by which temporal distance is measured to be selected in advance is relatively new; prior to this thesis, this problem has only been studied in the context of priority queues. Elmasry et al. [37] developed a priority queue that supports a constant number of temporal fingers. In particular, this shows the existence of a priority queue that supports both the working-set and queueish properties.

4.2 The Data Structure

Our data structure consists of two parts: OLD and YOUNG. A schematic of the data structure is illustrated in Figure 4.1.

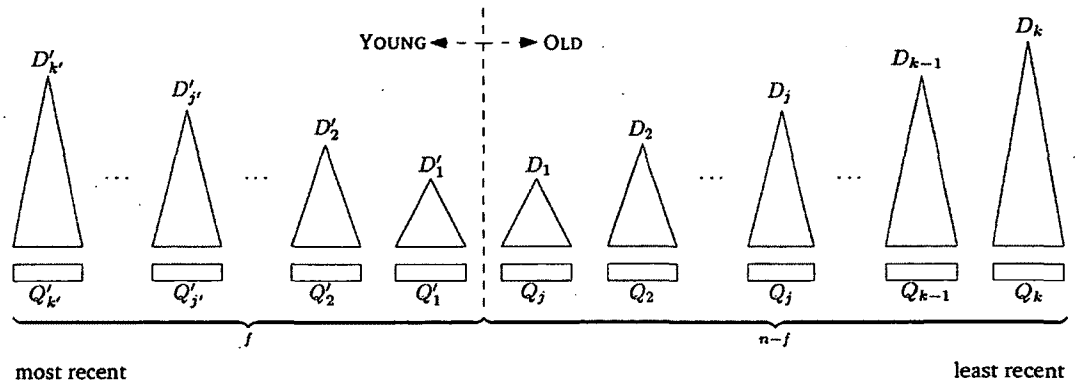


Figure 4.1: A schematic of the data structure for temporal finger searching. Pointers between elements in substructures and the corresponding queue elements are not shown. The substructures in the OLD data structure are drawn as trees, but are actually implemented as sorted arrays.

4.2.1 The OLD Data Structure

The OLD data structure contains the $n - f$ elements that were last accessed more than f queries ago. They are stored in a working-set structure [20].

As discussed in Section 3.1.2, the working-set structure consists of balanced binary search trees (*e.g.*, AVL trees [1]) $D_1, D_2, \ldots, D_k$ of size 2^{2^j} for $j = 1, 2, \ldots, k$. It follows that k is $O(\log\log(n - f))$. Each tree D_j has an accompanying queue Q_j containing the same elements as in the order that they were inserted into D_j. Pointers are maintained between an element in a tree and the corresponding element in the queue. The concatenation of all queues is precisely a list of the $n - f$ elements in the order they were queried, from most recent to least recent.

4.2.2 The YOUNG Data Structure

The YOUNG data structure contains the f elements that were accessed at most f accesses ago. They are stored in a modified queueish dictionary [45].

The queueish dictionary consists of a series of substructures $D'_1, D'_2, \ldots, D'_{k'}$ and queues $Q'_1, Q'_2, \ldots, Q'_{k'}$. As in the OLD data structure, the concatenation of the queues in this data structure orders the elements in increasing order of last access time. However, the queues no longer correspond exactly to the substructures: all of the elements of $Q'_1 \cup Q'_2 \cup \cdots \cup Q'_j$ are stored in D'_j, but D'_j may contain additional elements. Pointers are maintained between each element of D'_j and its corresponding entry in a queue (which may not be Q'_j). The size of Q'_j is between 2^{2^j-1} and 2^{2^j}. $D'_{k'}$ will contain all elements in the structure and thus have size f, and $Q'_{k'}$ has size at least $2^{2^{k'}-1}$. Therefore, $k' = O(\log\log f)$. As suggested by Iacono and Langerman [45], D'_j can be implemented as a sorted array. Note that $D'_{k'}$ will be implemented differently; we address this issue during the analysis.

4.2.3 Performing a Query

To perform the query x at time t, we search in $D_1, D'_1, D_2, D'_2, \ldots$ and so on, until x is found. At this point x is now the most recently accessed (*i.e., youngest*) element in the data structure, so we delete it from the structure we found it in and insert it into YOUNG. There are two possible cases: either x is found in OLD or YOUNG.

Suppose first that x is found in OLD, say $x \in D_j$. In this case, we delete x from D_j, insert x into YOUNG. We then delete the oldest element in YOUNG and insert it into OLD. In doing so, we will need to shift elements in OLD down to restore the size of D_j. This can be done by taking the oldest element out of each subtree and

placing it in the next larger subtree until we reach D_j.

Suppose now that x is found in YOUNG, say $x \in D'_j$. This case is handled exactly as it would be in a regular queueish dictionary: x is removed from Q'_j and inserted at the front of $Q'_{k'}$. In doing so, $|Q'_j|$ may become too small (*i.e.*, less than $2^{2^{j-1}}$). If this occurs, we remove $2^{2^j} - |Q'_j|$ elements from the end of Q'_{j+1}, insert them at the front of Q'_j, and reconstruct D'_j from the elements in $Q'_1, Q'_2, \ldots, Q'_j$. This may result in Q'_{j+1} becoming too small (and so on); these cases are handled identically.

4.2.4 Access Cost

The cost of an access can be separated into two parts: the cost of finding the query element and the cost of adjusting the data structure.

Finding x. To find the element x at time t, we search in $D_1, D'_1, D_2, D'_2, \ldots$ until x is found in D_j or D'_j. The cost to find the element is therefore $\sum_{l=1}^{j} O\left(\log 2^{2^l}\right) = \sum_{l=1}^{j} O(2^l) = O(2^j)$. Again, there are two possible cases: either $x \in D_j$ or $x \in D'_j$.

If $x \in D_j$, then $w_t(x) = f + \Omega\left(2^{2^{j-1}}\right)$. Therefore, $\tau_{t,f}(x) = |w_t(x) - f + 1| = |f + \Omega\left(2^{2^j}\right) - f + 1| = \Omega\left(2^{2^{j-1}}\right)$. Equivalently, $j \leq \log\log \tau_{t,f}(x) + O(1)$. The cost to find the element is therefore $O(2^j) = O(\log \tau_{t,f}(x))$.

If $x \in D'_j$, then $q_t(x) = (n - f) + \Omega\left(2^{2^{j-1}}\right)$, and since $w_t(x) = n - q_t(x) - 1$, we have $w_t(x) = f - \Omega\left(2^{2^{j-1}}\right) - 1$. Therefore, $\tau_{t,f}(x) = |w_t(x) - f + 1| = |f - \Omega\left(2^{2^{j-1}}\right) - 1 - f + 1| = |-\Omega\left(2^{2^{j-1}}\right)| = \Omega\left(2^{2^{j-1}}\right)$. Equivalently, $j \leq \log\log \tau_{t,f}(x) + O(1)$. The cost to find the element is therefore $O(2^j) = O(\log \tau_{t,f}(x))$.

In either case, we have that the portion of the access cost dedicated to finding x is $O(\log \tau_{t,f}(x))$.

Adjusting the Data Structure. The data structure must now be adjusted. If x is found in YOUNG, then YOUNG can be restructured in the usual manner at amortized cost $O(\log\log f)$ by radix sorting the indices, as suggested by Iacono and Langerman [45]. If x is found in OLD, however, an insertion must be performed into $D'_{k'}$. If $D'_{k'}$ is implemented as a binary search tree or sorted array, this will take time $\Theta(\log f)$, which is too slow.

We therefore describe alternative implementations of $D'_{k'}$ to improve our access time.

The first alternative is to use a y-fast trie [67]. In this case, the word size can be considered $\Theta(\log n)$, since we are effectively operating in rank space. Doing so allows us to insert, delete and search in $D'_{k'}$ in amortized time $O(\log\log n)$. This follows from the fact that we know the rank of x because we have already found it in the previous stage.

Recall that the queueish dictionary still requires a way of rebuilding smaller structures from larger structures. If the other substructures are implemented as arrays, all that is required is following pointers from $Q'_{k'}$ to the oldest elements in $D'_{k'}$, placing them in an array, deleting them from $Q'_{k'}$, and proceeding as usual. This results in an amortized query time of

$$O(\log \tau_{t,f}(x)) + O(\log\log n)$$

The second alternative is to instead use the predecessor search structure of Beame and Fich [8]. Doing so allows us to insert, delete and search in $D'_{k'}$ in

time

$$O\left(\min\left\{\frac{(\log\log n)(\log\log f)}{\log\log\log n}, \sqrt{\frac{\log f}{\log\log f}}\right\}\right)$$

This results in an amortized query time of

$$O(\log \tau_{t,f}(x)) + O\left(\min\left\{\frac{(\log\log n)(\log\log f)}{\log\log\log n}, \sqrt{\frac{\log f}{\log\log f}}\right\}\right)$$

At this point we note that, using the first technique, we match the performance of the queueish dictionary described by Iacono and Langerman [45] when $f = n$. Using the second technique, we match the performance of the working-set structure of Bădoiu et al. [20] when $f = 1$. It is also straightforward to determine in advance which technique should be used: if f is $\Omega(\log n)$, then the first technique should be used; otherwise the second technique should be used.

To summarize, we have

Theorem 4.1. *Let $1 \leq f \leq n$. There exists a static dictionary over the set $\{1, 2, \ldots, n\}$ that supports querying element x in amortized time*

$$O(\log \tau_{t,f}(x)) + O\left(\min\left\{\log\log n, \frac{(\log\log n)(\log\log f)}{\log\log\log n}, \sqrt{\frac{\log f}{\log\log f}}\right\}\right)$$

where $\tau_{t,f}(x)$ denotes the temporal distance between the query x and the element f at time t.

4.3 Conclusion and Open Problems

In this chapter, we constructed the first dictionary data structure that supports query times that are sensitive to an arbitrarily placed temporal finger.

There are several possible directions for future research.

1. Can the additive term in Theorem 4.1 be reduced? This would be interesting even for specific (ranges of) values of f. When $f = n$, for example, the best known result is $O(\log \tau_{t,n} n + \log \log n)$ [45]. The case when $f = 1$ is fully solved by Bădoiu et al. [20].

2. Is it possible to support multiple temporal fingers (*e.g.*, $O(1)$ many)? Simply searching the structures in parallel allows us to find the query element in time proportional to the logarithm of the minimum temporal distance, but it is not obvious how to quickly restructure the data structures and promote the query element to the cheapest substructure in parallel for each structure.

3. Is it possible to maintain the temporal finger property while supporting dynamic update operations?

Chapter 5

The Strong Unified Property

In this chapter, we define a stronger version of the unified property. In particular, we count distance only among elements in a certain subset of the dictionary (as opposed to all elements). For many access sequences, this results in smaller query times. We then present a data structure that comes to within a small additive term of the query time stated by this property.

5.1 Problem Definition

We consider a set S and a sequence $A = \langle a_1, a_2, \ldots, a_m \rangle$, where $a_t \in S$. As in Chapter 4, we assume $S = \{1, 2, \ldots, n\}$ (*i.e.*, a reduction to rank-space) and that the set S is static.

We wish to store the elements of S in a data structure so that accessing the elements in the order defined by A is *fast*. Recall that the unified property roughly states that an access is fast if it is close to a recent access. For example, in the access

sequence

$$\left\langle 1, \frac{n}{2}+1, 2, \frac{n}{2}+2, 3, \frac{n}{2}+3, \ldots, \frac{n}{2}, n, 1, \frac{n}{2}+1, \ldots \right\rangle$$

almost every access is at a distance of one from the element accessed two accesses ago. However, consider the following access sequence:

$$\left\langle K, \frac{n}{2}+K, 2K, \frac{n}{2}+2K, 3K, \frac{n}{2}+3K, \ldots \right\rangle$$

where K is a large number. This new access sequence is very similar to the old one, except that distances are no longer preserved. Therefore, the unified property no longer indicates that this sequence should execute quickly. However, it still seems as if this access sequence has some locality: each access is close in terms of rank distance among recently-accessed elements to the query made two accesses ago.

5.1.1 Defining the Strong Unified Property

Recall the definitions used in Section 2.2.3. Define

$$l_t(x) = \min\left(\{\infty\} \cup \{t' > 0 \mid a_{t-t'} = x\}\right)$$

One can think of $l_t(x)$ as most recent time x has been queried in A before time t. We then define

$$w_t(x) = \begin{cases} n & \text{if } l_t(x) = \infty \\ \left|\{a_{t-l_t(x)+1}, \ldots, a_t\}\right| & \text{otherwise} \end{cases}$$

We also define $W_t(s)$ to be the set of elements $x \in S$ such that $w_t(x) \leq s$ and

$$O\left(\min_{y \in W_t(w_t(x)^2)} \log\left(w_t(y) + d\right.\right.$$

$d_T(a, b)$, for a set T and elements $a, b \in T$, to be the rank distance between a and b in the set T.

Intuitively, we would like the strong unified property to state that the time to access an element x at time t to be

$$O\left(\min_{y \in W_t(w_t(x))} \log\left(w_t(y) + d_{W_t(w_t(x))}(x, y)\right)\right)$$
$$O\left(\min_{y \in W_t(w_t(x))} \log\left(w_t(y) + d_{W_t(w_t(x))}(x, y)\right)\right) \qquad (*)$$

Unfortunately, such an access time is not possible. To see this, consider the access sequence

$$\langle 2, 3, 4, 5, 6, \ldots, 1, x, \ldots \rangle$$

where $x \in S$. Suppose the access to x occurs at time t. Note that the access at time $t-1$ is to 1 and that $2, 3, \ldots, x-1$ are not present in $W_t(w_t(x))$. Therefore, the value of y that minimizes (*) is 1, which has constant working-set number and constant rank distance to x. This means that for *any* of the n choices of x, the strong unified property requires the data structure to perform the access in constant time; this is not information-theoretically possible [59].

We therefore modify the strong unified bound so that it is at least information-theoretically plausible:

$$O\left(\min_{y \in W_t(w_t(x)^2)} \log\left(w_t(y) + d_{W_t(w_t(x)^2)}(x, y)\right)\right)$$

Note that this $W_t(w_t(x)^2)$ includes elements that were accessed less recently than x. These additional elements will allow us to support the rest of the access cost while respecting information-theoretic lower bounds. The intuition for expanding the set under consideration in this manner is the fact that the data structure

will consists of substructures that increase doubly-exponentially in size, and so by squaring the working-set number under consideration, we can take advantage of elements in the next substructure.

As an example, consider the following access sequence, where 15 is the element currently being searched for at the end of the sequence:

$$2, 3, 4, 5, 6, 7, 8, \overbrace{9, 10, 11, 12, 13, \underbrace{14, 15, 16, 1}_{W_t(w_t(15))}}^{W_t(w_t(15)^2)}, 15$$

The original definition in (*) uses $d_{W_t(w_t(15))}$, whereas the modified definition uses $d_{W_t(w_t(15)^2)}$. The modified definition allows for $9, 10, 11, 12$ and 13 to contribute to the rank distance which results in an information-theoretically plausible query time, since it no longer results in a situation where any query must be executed in constant time.

5.1.2 Background

The unified property was introduced by Bădoiu et al. [20], who also showed how to construct a data structure that matched this query time. Derryberry [30] further considered the problem when restricted to the binary search tree model and introduced skip-splay and cache-splay trees; the former comes within an additive $O(\log \log n)$ of the unified bound, while the latter achieves it to within a constant factor.

The idea of restricting the rank space used to count distance has not previously been studied.

5.2 Towards the Strong Unified Property

In this section, we describe a data structure that achieves the strong unified property to within a small additive term of

$$O\big((\log d_{W_t(w_t(x))}(x,y))(\log\log w_t(x))\big)$$

5.2.1 The Data Structure

The data structure consists of k finger search trees as well as k accompanying queues. The finger search trees of Brodal et al. [19] support insertions and deletions in $O(1)$ worst-case time (when provided with a pointer to the element to be deleted) and finger searches in $O(\log d)$ worst-case time, where d is the distance between the element being searched for and the supplied pointer into the data structure.

The size of T_j is 2^{2^j}, except for T_k which has size n. It follows that k is $O(\log\log n)$. We will maintain the invariant that $T_j \subset T_{j+1}$ for all $1 \leq j < n$. The queue Q_j contains exactly the same elements as T_j in the order they were inserted into T_j. Pointers are maintained between elements in the queue and corresponding elements in the finger search tree. A schematic is presented in Figure 5.1.

To perform a search, we will perform finger searches in $T_1, T_2, \ldots$ until we find $x \in T_j$. In T_1, we use an arbitrary element as the starting finger for the search. In all other trees, we run two finger searches in parallel, one from the successor of the element found in the previous tree, and one from the predecessor of the element found in the previous tree, stopping as soon as one of these searches terminates. Once we have found $x \in T_j$, we insert x into $T_1, T_2, \ldots, T_{j-1}$ (note that x is not

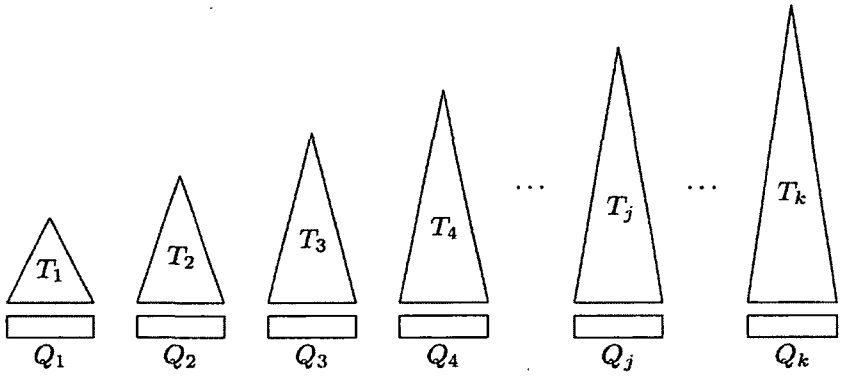


Figure 5.1: A schematic of the data structure for the strong unified property. Pointers between elements in finger search trees and the corresponding queue elements are not shown.

present in any of these trees, since if it were, it would have already been found) and enqueue x in $Q_1, Q_2, \ldots, Q_{j-1}$. At this point, we note that each of $T_1, T_2, \ldots, T_{j-1}$ and $Q_1, Q_2, \ldots, Q_{j-1}$ are too big. We therefore dequeue the oldest element in each of $Q_1, Q_2, \ldots, Q_{j-1}$ and delete the corresponding elements in $T_1, T_2, \ldots, T_{j-1}$.

5.2.2 Analysis

Recall that we are aiming for a running time of

$$O\left(\underset{y\in}{O}\left(\min_{y\in W_t(w_t(x)^2)} \log\left(w_t(y) + d_{W_t(w_t(x)^2)}(x,y)\right)\right)\right)$$

Consider a search for x at time t, and consider the element y that minimizes the above expression. Suppose x first appears in T_j and y first appears in $T_{j'}$. Because x first appears in T_j, we have that $w_t(x) \geq 2^{2^{j-1}}$. Therefore, $j \leq \log\log w_t(x) + 1$. Similar reasoning shows $w_t(y) \geq 2^{2^{j'-1}}$, so that $j' \leq log\log w_t(y) + O(1)$.

If $j \leq j'$ (*i.e.*, x appears no later than y), then the running time follows easily:

x has working-set number $w_t(x) \leq w_t(y)$. The element x can thus be found in time $\sum_{l=1}^{j} 2^l = O(2^j)$, which is $O(2^{j'}) = O(\log w_t(y))$.

The more interesting case occurs when $j > j'$ (*i.e.*, x appears after y). Here, y can be found in time $\sum_{l=1}^{j'} 2^l = O(2^{j'}) = O(\log w_t(y))$. By the time the algorithm finishes searching $T_{j'}$, it has a finger for an element y' such that $d_{W_t(w_t(x))}(x, y') \leq d_{W_t(w_t(x))}(x, y)$. Therefore, each of the remaining finger searches is over a rank distance of at most $d_{W_t(w_t(x))}(x, y)$, except for the last search which is over a rank distance of at most $d_{W_t(w_t(x)^2)}(x, y)$. There are thus $O(\log \log w_t(x))$ finger searches that cost $O(\log d_{W_t(w_t(x))}(x, y))$ to be performed, and one that costs $O(\log d_{W_t(w_t(x)^2)}(x, y))$. The total cost of these searches is therefore

$$O\big((\log d_{W_t(w_t(x))}(x, y))(\log \log w_t(x)) + \log d_{W_t(w_t(x)^2)}(x, y)\big)$$

At this point, x has been found and we must now adjust the data structure. First, x must be inserted in $T_1, T_2, \ldots, T_{j-1}$. Because we have a finger for x inside each of these structures, this takes total time $O(\log \log w_t(x))$. Enqueuing x in each of $Q_1, Q_2, \ldots, Q_{j-1}$ also takes $O(j) = O(\log \log w_t(x))$. The subsequent deletions and dequeueings of the oldest elements in $Q_1, Q_2, \ldots, Q_{j-1}$ and $T_1, T_2, \ldots, T_{j-1}$ take a total of $O(j) = O(\log \log w_t(x))$ time as well, since the dequeueing operation takes $O(1)$ time and provides a pointer to the node in the corresponding tree where the deletion must be performed.

We therefore have

Theorem 5.1. *There exists a static dictionary over the set* $\{1, 2, \ldots, n\}$ *that supports*

querying element x in time

$$O\left(\left(\min_{y \in W_t(w_t(x)^2)} \log\left(w_t(y) + d_{W_t(w_t(x)^2)}(x,y)\right)\right) + (\log d_{W_t(w_t(x))}(x,y))(\log\log w_t(x))\right)$$

5.3 Conclusion and Open Problems

In this chapter, we defined a stronger version of the unified property and described a data structure that achieves it to within a small additive term. Instead of computing rank distance over the entire dictionary, we compute rank distance only within a working-set containing an element that is close to a recently-accessed element.

There are several possible directions for future research.

1. One can reduce the distance measure $d_{W_t(w_t(x)^2)}(x,y)$ to $d_{W_t(w_t(x)^{1+\epsilon})}(x,y)$ by changing how the substructures grow. Is it possible to reduce this further, say to $d_{W_t(O(w_t(x)))}(x,y)$?

2. We argued that it is not possible to use $d_{W_t(w_t(x))}(x,y)$ in the worst case. Is this possible to achieve this in the amortized sense?

3. Can the additive term in Theorem 5.1 be reduced? It seems difficult to reduce this term below $\Omega(\log\log w_t(x))$ using an approach similar to the one presented here, since elements must shift through this many substructures.

4. Is it possible to maintain the strong unified property while supporting dynamic update operations?

Chapter 6

Distance-Sensitive Predecessor Search in Bounded Universes

In this chapter, we show how to perform predecessor searches in bounded universes in a distribution-sensitive manner. Given a set of elements drawn from a larger universe of a known size, we wish to know the largest element of the set that is less than or equal to (*i.e.*, the predecessor of) a given query drawn from the universe.

The time for a predecessor query will be a function of the distance between the query and the answer to the query, which unifies results on hashing data structures (*e.g.* [33]) with results from predecessor search data structures (*e.g.* [67]). We also give several useful applications of a distance-sensitive predecessor search data structure, namely the approximate nearest neighbour search problem and the range searching problem.

6.1 Problem Definition

Let $\mathcal{U} = \{0, 1, \ldots, U-1\}$. We address the problem of maintaining a dictionary subject to searches, insertions and deletions on a subset $S \subseteq \mathcal{U}$ of size $|S| = n$ in the word-RAM model. In this model, we allow constant-time access to elements of $\mathcal{U}$, which are repesented by $\Theta(\log U)$ bits, and arithmetic and comparison operations on elements of $\mathcal{U}$ also take constant time.

This problem has been well-studied in several different contexts. If we do not take advantage of the fact that $S \subseteq \mathcal{U}$, then the results of Chapter 3 apply. Using the fact that $S \subseteq \mathcal{U}$, however, we can use dynamic perfect hashing [33] to support all operations in $O(1)$ expected time.

It would therefore seem that the problem is solved. However, hashing does not provide a useful answer after an unsuccessful query. In such a situation, one turns to predecessor queries. In such a query, the data structure must return the element searched for (if it is stored in S) or the largest element that is smaller than the query element (if the query element is not stored in S). Again, if we ignore the fact that $S \subseteq \mathcal{U}$, then the results of Chapter 3 apply. However, using the fact that $S \subseteq \mathcal{U}$ again allows us to achieve a query time of $O(\log\log U)$ using, *e.g.*, van Emde Boas trees [64], x- and y-fast tries [67], or the data structure of Mehlhorn and Näher [52]; this is an improvement when n is $\omega(\log U)$.

Space is an important consideration in such problems. The static version of the problem, for example, can be solved using $O(U)$ space simply by precomputing the answer to every query. However, this amount of space is far too large. Generally, we aim for $O(n)$ or $O(n \log^c U)$ space for some constant c. The same is true for the dynamic version of the problem.

We introduce the idea of *local searching* in a bounded universe in order to unify the results of hashing with those of predecessor search structures. Suppose we have a query $x \in \mathcal{U}$ and let $\text{pred}(x) \in S$ denote its predecessor stored in the data structure (if $x \in S$, then we define $\text{pred}(x) = x$). Equivalently, $\text{pred}(x) = \max\{y | y \leq x\}$. Define $\Delta = |x - \text{pred}(x)|$. We wish to answer predecessor queries (*i.e.*, return $\text{pred}(x)$) in time $O(\log\log\Delta)$. Observe that if it happens that x is in the data structure, then $\Delta = 0$ and so the query will run in $O(1)$ time, which matches the query time obtained by hashing. Conversely, if $x \notin S$, then we still have $\Delta < U$, and so the query runs in $O(\log\log U)$ time, which is no worse than that offered by the usual predecessor search structures. We will also show how updates can be supported with the same time bound.

6.1.1 Background

As mentioned previously, predecessor searches can be performed by van Emde Boas trees [64] or the x- and y-fast tries of Willard [67]. van Emde Boas trees support a query time of $O(\log\log U)$ using space $O(U)$. This large space usage is the primary drawback of van Emde Boas trees. In general, one endeavours to avoid any dependence on U in the space requirements; linear dependence is not acceptable, while polylogarithmic dependence is more reasonable. x- and y-fast tries were proposed by Willard to overcome this deficiency and support the same query time of $O(\log\log U)$ using space $O(n)$. We elaborate on these results in Section 6.1.2 since they play a key role in the results of this chapter.

Perhaps surprisingly, the query time of $O(\log\log U)$ can be further improved somewhat. Beame and Fich [8] gave a data structure that uses space $n^{O(1)}$ and

supports a query time of

$$O\left(\min\left\{\frac{\log\log U}{\log\log\log U}, \sqrt{\frac{\log n}{\log\log n}}\right\}\right)$$

The space can be reduced to $O(n)$ by allowing an additional $O(\log\log n)$ factor in the first half of the previous bound to achieve a bound of

$$O\left(\min\left\{\frac{\log\log U}{\log\log\log U}\log\log n, \sqrt{\frac{\log n}{\log\log n}}\right\}\right)$$

There has been further study into the exact complexity of the predecessor search problem [55, 56], but those results go beyond the scope of the distribution-sensitive data structures to be discussed in this chapter.

Few distribution-sensitive results are known for the predecessor search problem. There are two results that can be viewed as "close" to the result presented here. The first is that of Kaporis et al. [47], who presented a property similar to finger search and supported queries in time $O(\log\log d)$, where d is the distance from the query to a finger. However, that query time only applies to subsets S that are chosen randomly according to a particular kind of distribution. A data structure with a similar restriction was presented by Belazzougui et al. [9] that supports constant query time with high probability. It is crucial to note that these data structures use randomization *in the input* and therefore depart significantly from the usual setting.

The second result similar to the one here is that of Andersson and Thorup [4], who showed that finger search can be supported in time $O\left(\sqrt{\log d/\log\log d}\right)$. If one does not take advantage of the fact that $S \subseteq \mathcal{U}$, then the other finger search results mentioned in Section 2.4 apply as well, of course, but are exponentially

slower than those discussed in this chapter for subsets S of sufficient size. The $O(n)$-space data structure of Nekrich [53] supports $O(1)$-time update operations provided that the updates occur near existing elements, but only achieves a query time of $O(\log \log U)$.

6.1.2 x- and y-fast Tries

We begin with a review of x- and y-fast tries, which were presented by Willard [67] as space-efficient alternatives to van Emde Boas trees [64].

An x-fast trie is a binary tree whose leaves are elements of $\mathcal{U}$ and whose internal nodes represent the binary prefixes of these leaves. Since any element of $\mathcal{U}$ can be represented by $O(\log U)$ bits, the height of an x-fast trie is $O(\log U)$. At any internal node, moving to the left child appends a 0 to the prefix, while moving to the right child appends a 1. The prefix at the root of the trie is empty. Therefore, a node at depth i represents a block of the at most $2^{\log_2 U - i}$ elements of S having the same i highest order bits. Any root-to-leaf path in the trie yields the binary representation of the element at the leaf of that path.

At every level of the tree (where the level of a node is equal to the height of the subtree rooted at that node, so that leaves are at level 0), a hash table is maintained on all nodes (*i.e.*, prefixes of elements) at that level. Each internal node with no left (respectively right) child is augmented with an additional pointer to the smallest (respectively largest) leaf in its subtree, and all leaves maintain pointers to both the previous and next leaves. Nodes with no elements stored at their leaves are removed from the tree. An x-fast trie therefore uses $O(n \log U)$ space, since each of the n elements of S appears in $O(\log U)$ hash tables (one at each level).

A query is performed by executing a binary search on the hash tables to find the deepest node whose prefix is a prefix of the query. If the query x has binary representation $x = x_{\log_2 U} x_{\log_2 U - 1} \cdots x_0$, then the query in the hash table at depth i is for $x_i = x_{\log_2 U} x_{\log_2 U - 1} \cdots x_{\log_2 U - i}$. If this nodes happens to be a leaf, then the search is complete. Otherwise, this node must have exactly one child (since otherwise the found node is not the deepest node whose prefix is a prefix of the query). Since the node has only one child, this node has a pointer to the largest (or smallest) leaf in its subtree. Following this pointer will lead to a node that is distance at most 1 away from the predecessor of the query. Since the leaves form a doubly-linked list, the correct predecessor can thus be found easily. The binary search among the levels takes $O(\log \log U)$ time since there are $O(\log U)$ levels, and the subsequent operations take $O(1)$ time, for a total of $O(\log \log U)$ time.

The main drawback of x-fast tries is the fact that they use $O(n \log U)$ space. The y-fast trie overcomes this obstacle by using indirection to reduce the space usage to $O(n)$. The leaves are divided into $O(n/\log U)$ groups called *buckets*, each of size $O(\log U)$, and each group is placed into a balanced binary search tree. Each binary search tree has a representative which is stored in the trie. A search in the trie for an element will eventually lead to a binary search tree, which can then be searched in time $O(\log \log U)$. Observe that the number of elements stored in the x-fast trie is $O(n/\log U)$, since only one representative from each group is stored. The total space needed is thus $O((n/\log U) \log U) = O(n)$. The y-fast trie also facilitates insertions and deletions easily by simply rebuilding the binary search trees when their sizes have doubled or quartered: this allows for insertion and deletion in expected amortized time $O(\log \log U)$.

6.2 A Preliminary Data Structure

In this section, we describe a preliminary version of the data structure that uses a large amount of space and does not support update operations. We will improve upon this in Section 6.3.

In order to make "local" searches (those searches where Δ is small) fast, the key observation is that instead of performing a binary search on the levels of an x-fast trie, we should instead perform a doubly exponential search on the levels (*i.e.*, searching levels 2^{2^i}), beginning at the leaves. By doing this, we ensure that any element for which $\Delta = 0$ is returned immediately. Furthermore, elements that are close to the query will be contained in the same (fairly small) subtree.

Suppose we have an x-fast trie. During a search, we perform hash table queries for the appropriate prefix of the query at level 0 and then at levels 2^{2^i} for $i = 0, 1, \ldots, O(\log\log\log U)$. If the prefix is found, the query can be answered by performing a search in the x-fast trie starting at that node (*i.e.*, all nodes not in the subtree rooted at that node can be ignored). If, however, the prefix is not found, then we search for the predecessor of that prefix (in the usual binary ordering) at the same level. If this predecessor is found, the query can be answered by following a pointer from that predecessor to the largest leaf in its subtree. If this predecessor is not found, then the exponential search continues.[1] Checking for the presence of this predecessor can be viewed as verifying that the predecessor of the query really is "far away" and so the cost of continuing the exponential search is not too large.

We now show that this query algorithm supports the desired query time.

[1]It is worth noting that we could also check for the successor of the appropriate prefix at that level, since this will also allow us to navigate quickly to a leaf that is close to the predecessor. Such an improvement does not alter the asymptotic query time.

Lemma 6.1. *The modified search algorithm for an x-fast trie described above is correct and can be performed in $O(\log\log\Delta)$ time.*

Proof. If the query is contained in the trie, it must be stored in the hash table containing all the leaves, and thus can be found in $O(1) = O(\log\log\Delta)$ time.

Assume the query is not contained in the trie. We must show that its predecessor is found in time $O(\log\log\Delta)$. Assume that a hash table query (for either the appropriate prefix or its predecessor) is first successful at level 2^{2^i}. Then both the prefix of the query and the predecessor of that prefix were not found at level $2^{2^{i-1}}$. If the prefix of the query and the predecessor had been in the trie, then the subtree rooted there would have size $O\left(2^{2^{2^{i-1}}}\right)$. However, since they are not present, these elements are not present either and we therefore have that $\Delta \geq 2^{2^{2^{i-1}}}$, so that i is $\log\log\log\Delta + O(1)$. The exponential search reaches level 2^{2^i} in $O(i)$ time, and the subsequent search of the trie rooted at the found node will take $O\left(\log\log 2^{2^{2^{i-1}}}\right) = O(2^i)$ time, for a total of $O(2^i) = O(\log\log\Delta)$ time, as required. □

Observe that this data structure is essentially the same as an x-fast trie; we are merely changing the search algorithm. It follows that the space used by this data structure is $O(n\log U)$.

Theorem 6.2. *Let $\mathcal{U} = \{0, 1, \ldots, U-1\}$. There exists a data structure that supports predecessor searches over a subset of $\mathcal{U}$ in time $O(\log\log\Delta)$ using $O(n\log U)$ space.*

As was the case with x-fast tries, the major drawback of the result of Theorem 6.2 is that its space requirements are a function of $\mathcal{U}$. Although only logarithmic, this extra factor is still fairly large. In the next section, we show how to

drastically reduce this factor, while at the same time making the data structure dynamic.

By replacing "predecessor" with "successor" throughout the description of the data structure, the same time bound can be achieved where Δ is defined to be the difference between the element being searched for and its *successor* in the data structure. Furthermore, if both structures are searched simultaneously until one returns an answer, it is possible to answer such queries in $O(\log\log\Delta)$ time where Δ is the minimum of the differences between the query and its predecessor and successor in the structure.

6.3 Reducing Space and Supporting Updates

In this section, we will improve the result of Theorem 6.2 by reducing the space requirements and by supporting insertions and deletions.

One straightforward solution would be to modify the data structure from Theorem 6.2: to insert an element, we add the appropriate prefixes to all of the hash tables, and to delete an element we delete the appropriate prefixes from the hash tables (with some additional bookkeeping to make sure that no other element stored has a prefix that we delete). Unfortunately, this technique is not fast enough for our purposes: there are $O(\log U)$ hash tables in an x-fast trie (one for each level) and so the update time would be $O(\log\log\Delta + \log U)$. This can be partially overcome by storing only the hash tables at level 0 and 2^{2^i} for $i = 0, 1, \ldots, O(\log\log\log U)$. Of course, the search algorithm would still have to be adapted so that not every hash table need be maintained. Furthermore, this still only improves the update time to

$O(\log\log\Delta + \log\log\log U)$.

The key problem here is that too many hash tables must be updated during an insertion or deletion. We would therefore like to have each hash table maintain the appropriate prefixes without needing explicit updating after every insertion and deletion. To do so, we again use indirection.

We maintain a skip list[2] [57] that stores all elements currently in the trie. Pointers are maintained between the elements of the trie and their corresponding elements in the skip list. Furthermore, each element of the hash tables maintains a y-fast trie whose universe is the maximal subset of $\mathcal{U}$ that could be stored as leaves of that node; equivalently, the universe is the set of all elements of $\mathcal{U}$ whose prefix is the element. Therefore, an element of a hash table at level 2^{2^i} has a y-fast trie over a universe of size $2^{2^{2^i}}$. Recall that y-fast tries maintain buckets that have size that is logarithmic in the size of the universe. This means that the y-fast trie pointed to by an element of a hash table at level 2^{2^i} has a y-fast trie with bucket size 2^{2^i}. The key to reducing the query time is to maintain these buckets indirectly. Any pointer in the "top" portion of a y-fast trie that points to an element in a bucket instead points to a representative of that bucket that is stored in the skip list.

The remaining trick is to carefully select a representative of the bucket in order to ensure that buckets do not need to be frequently rebuilt during insertions and deletions. We shall select representatives based on their height in the skip list. For reasons that will become clear shortly, we select the probability of promotion in the skip list to be $1/3$. Now, consider a y-fast trie pointed to by a hash table element at level 2^{2^i}. The representatives of the buckets for that y-fast trie will be the elements

[2]Skip lists belong in the pointer machine model, and can therefore by implemented in our model as well.

having that prefix who have height at least 2^i in the skip list. The expected size of a bucket is thus the expected number of elements between two elements of the skip list at height 2^i. Since the probability of promotion is $1/3$, the expected size of a bucket is thus $1/(1/3)^{2^i} = 3^{2^i}$.

Lemma 6.3. *The expected space used by this data structure is* $O(n \log\log\log U)$.

Proof. Since a prefix corresponding to each of the n elements in the dictionary is stored in $O(\log\log\log U)$ hash tables and y-fast tries, each of which uses space that is linear in the number of elements stored in them, the space required for the trie portion of the data structure is $O(n \log\log\log U)$. The expected space used by the skip list is $O(n)$. Therefore, the total expected space is $O(n \log\log\log U)$. □

There remains one slight technical issue that must be dealt with before discussing how to perform operations on this data structure. By allowing insertions and deletions, it is possible to force many hash table updates in the following way. Consider a (large) empty interval of $\mathcal{U}$ that is adjacent to an element stored in the data structure. If the boundary of a tall subtree happens to line up with the end of this interval and the largest element of this interval is repeatedly inserted and deleted, then $\Delta = O(1)$ for all operations, but all hash tables must be updated (since there are no other possible representatives in this interval). Updates will therefore take $\Omega(\log\log\log U)$ time. To remedy this, we randomly shift the universe before building the data structure.[3] Intuitively, performing this random shift means that the probability of being forced to make many hash table updates is small. More formally:

[3]Of course, even after randomly shifting the universe, this bad situation can still arise. However, we make the standard assumption that the adversary lacks access to the size of the shift.

Lemma 6.4. *After performing a random shift by r ($0 \leq r < U$) of $\mathcal{U}$, the expected height of the lowest common ancestor of two leaves x and y is $O(\log |x - y|)$.*

Proof. Let $\Delta = |x - y|$ and assume, without loss of generality, that $x < y$. We begin by computing the probability that the lowest common ancestor of x and y has height at least h. There are at least 2^h elements of $\mathcal{U}$ inside a subtree with height at least h. The probability that the lowest common ancestor of x and y has height at least h is therefore the probability that $r + x$ and $r + x + \Delta$ are in different subtrees of height at least h. This happens precisely when $x + r$ is contained in the last Δ positions of its subtree, and so a probability of $\Delta/2^h$ follows.

We now compute the expected height of the lowest common ancestor x and y:

$$\sum_{h=1}^{\infty} h\frac{\Delta}{2^h} = \sum_{h=1}^{\log \Delta} h\frac{\Delta}{2^h} + \sum_{h=\log \Delta+1}^{\infty} h\frac{\Delta}{2^h} \leq \sum_{h=1}^{\log \Delta} 1 + \sum_{h=\log \Delta+1}^{\infty} h\frac{\Delta}{2^h} = O(\log \Delta)$$

Since $\Delta = |x - y|$, the lemma follows. □

Note that Lemma 6.4 is not needed to prove the running time of the search operation.

Search. To execute a search, we proceed as we did in the previous section and perform hash table queries for the appropriate prefix of the query at levels 2^{2^i} for $i = 0, 1, \ldots, O(\log \log \log U)$. If the prefix is not found at a level, we search for the predecessor of that prefix (in the usual binary ordering) at the same level. If the prefix or such a predecessor is found, then the query can be answered by following a pointer to the maximum representative in the associated y-fast trie and then performing a finger search in the skip list using a pointer to this representative. If the

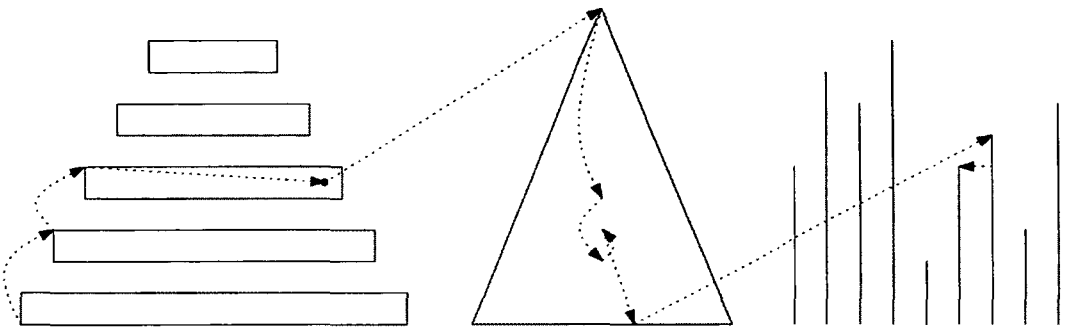


Figure 6.1: Searching for the predecessor of a query. The search path begins at the left, where levels 2^{2^i} (for $i = 0, 1, \ldots, O(\log\log\log U)$) of the modified x-fast trie are searched. When an appropriate element is found (either the appropriate prefix or its predecessor), the search moves to the corresponding y-fast trie, which is then searched in the usual way. This produces a representative of the bucket that contains the predecessor. Once this representative is found, the search path moves into the skip list and performs a finger search using the representative element as a finger.

predecessor of the prefix was not found, then the exponential search is continued. The search algorithm is summarized in Figure 6.1.

We now show that this query algorithm supports the desired query time.

Lemma 6.5. *The search algorithm described above can be performed in* $O(\log\log\Delta)$ *expected time.*

Proof. As in the proof of Lemma 6.1, if the query is contained in the trie, it must be stored at the hash table containing all leaves and is thus found in $O(1) = O(\log\log\Delta)$ time. Otherwise, assume the query is not contained in the trie and a hash table query (either for the appropriate prefix or its predecessor) is first successful at level 2^{2^i}. Then $\Delta \geq 2^{2^{2^i-1}}$, and so i is $\log\log\log\Delta + O(1)$.

If the successful hash table query was for the appropriate prefix, then the query is answered by querying the associated y-fast trie. Recall that a y-fast trie associated

with a node at level 2^{2^i} is over a universe of size $2^{2^{2^i}}$ and therefore can be queried in time $O\left(\log\log 2^{2^{2^i}}\right) = O(2^i)$. This yields the representative which is used as a finger in the skip-list. Since the representative and the desired predecessor are in the same bucket, the expected distance between them is 3^{2^i}. The finger search in the skip list thus takes $O\left(\log 3^{2^i}\right) = O(2^i)$ expected time. If the successful hash table query was for the predecessor of the appropriate prefix, then essentially the same argument shows that the query time is again $O(2^i)$.

In either case, the total expected query time is $O(2^i) = O(\log\log\Delta)$. □

Insertion. Inserting an element into the data structure can be accomplished by first searching for the element to be inserted in order to produce a pointer to that element's predecessor. This pointer is then used to insert the element into the skip list. We then insert the appropriate prefixes of the element into the hash tables and the corresponding y-fast tries. The y-fast tries may need to be restructured as the buckets increase in size, as described in Section 6.1.2. The key observation is that we need only alter a hash table when a representative changes.

Lemma 6.6. *The insertion algorithm described above can be performed in* $O(\log\log\Delta)$ *expected amortized time.*

Proof. During an insertion, we insert the element into the skip list given a pointer to its predecessor. The probability that the new element reaches level 2^i (and thus becomes a representative) is $(1/3)^{2^i}$. If this happens, the prefixes of the new element must be added to every hash table of the corresponding y-fast trie. Since this y-fast trie has height 2^{2^i} and each hash table insertion takes $O(1)$ expected time, the time to modify the y-fast trie is $O\left(2^{2^i}\right)$. This could happen at every level 2^{2^i}, however,

since the y-fast tries are nested. By Lemma 6.4, we expect to go no higher than height $O(\log \Delta)$. The total expected amortized time to restructure the y-fast tries is therefore

$$\sum_{i=0}^{\log \Delta} \left(\frac{1}{3}\right)^{2^i} O\left(2^{2^i}\right) \leq O\left(\sum_{i=0}^{\infty} \left(\frac{2}{3}\right)^{2^i}\right) \leq O\left(\sum_{i=0}^{\infty} \left(\frac{2}{3}\right)^{i}\right) = O(1)$$

Since the initial search takes expected time $O(\log \log \Delta)$ by Lemma 6.5, the insertion takes $O(\log \log \Delta)$ expected amortized time in total. □

Deletion. Deletion works similarly to insertion. We begin by locating the element to be deleted by performing a search to produce a pointer to it. The pointer is then used to delete the element from the skip list. Finally, the appropriate prefixes must be deleted from the hash tables (if they are not a prefix of another element in the structure, of course) and the element must be removed from the corresponding y-fast tries. The analysis applied to insertion applies equally well to deletion.

Lemma 6.7. *The deletion algorithm described above can be performed in* $O(\log \log \Delta)$ *expected amortized time.*

We therefore obtain

Theorem 6.8. *Let* $\mathcal{U} = \{0, 1, \ldots, U-1\}$. *There exists a data structure that supports predecessor searches over a subset of* $\mathcal{U}$ *in expected time* $O(\log \log \Delta)$ *and insertions and deletions in expected amortized time* $O(\log \log \Delta)$ *using* $O(n \log \log \log U)$ *expected space.*

Proof. Combining Lemma 6.5, Lemma 6.6 and Lemma 6.7 proves the time bounds on each operation. The expected space bound is proved in Lemma 6.3. □

6.4 Applications

In this section, we turn our attention to some applications of the data structure established by Theorem 6.8. We study the problems of approximate nearest neighbour queries and of range searching. As before, our results are restricted to bounded universes.

6.4.1 Approximate Nearest Neighbour Queries

Suppose we are given a point set $S \subseteq \mathcal{U}^d$ in a constant dimension $d \geq 2$. Given a query point in $\mathcal{U}^d$, we wish to determine the point in S that is closest to the query point. We refer to such a point as the nearest neighbour of the query. In general, this problem does not admit particularly fast solutions. Therefore, we often settle for a point that approximates the nearest neighbour. In particular, if the point in S nearest to the query point is at distance δ, we wish to return a point that is at distance at most $(1+\epsilon)\delta$ from the query point for some $\epsilon > 0$: such a point is a $(1+\epsilon)$-approximation of the nearest neighbour. If the query point happens to be in S, then we must return that point. Our goal query time is $O(\log\log\Delta)$, where Δ is the Euclidean distance from the query point to the point returned. We would also like to support insertions and deletions in a similar time bound.

This problem is reasonably well-studied. Amir et al. [3] show how to answer queries and perform updates in expected time $O(\log\log U)$ with exponential dependence on d. For the static version of the problem, they also present a deterministic data structure that has only linear dependence on d. In unbounded universes, Derryberry et al. [29] achieves a query time that is logarithmic in the number of points

in a box (whose size depends on d) containing the query point and the answer. There have been no results combining the bounded universe assumption with sensitivity to distance.

The result presented here is based on an observation due to Chan [21]. By placing $d+1$ shifted versions of the points of S onto a space-filling curve,[4] queries can be answered by answering the query on each of these curves (lists) and taking the closest point to be the desired approximation. This yields a query time of $O\big((1/\epsilon)^d \log n\big)$ and $O(\log n)$ update time for sets of size n by placing $O\big((1/\epsilon)^d\big)$ additional query points around the real query point.

To apply our results from Section 6.3, observe that searching with the shifted versions of the lists is a one-dimensional predecessor search problem and can thus be solved in expected time $O(\log\log\Delta)$ by Theorem 6.8. This almost completely solves the problem, except that we cannot bound the search time in *every* one-dimensional structure as a function of Δ, rather only the data structure which finds the element that is ultimately returned.

In order to fix this, we simply randomly shift the entire point set using the technique described by Har-Peled [39, Chapter 11] before building any of the data structures. The result of such a shift is that the expected time spent searching in any one-dimensional structure is $O(\log\log\Delta)$. By combining this with the technique of Chan [21], we achieve a query time of $O\big((1/\epsilon)^d \log\log\Delta\big)$. Insertions and deletions can be performed simply by performing the operation in each of the $O(d)$ one-dimensional structures.

[4] Chan [21] uses the z-order, which is obtained by computing the *shuffle* of a point. If the i-th coordinate of a point p is p_i with binary representation $p_{i,\log U}\ldots p_{i,0}$, then the shuffle of p is defined to be the binary number $p_{1,\log U}\cdots p_{d,\log U}\cdots p_{1,0}\cdots p_{d,0}$.

Theorem 6.9. *Let $\mathcal{U} = \{0, 1, \ldots, U-1\}$ and $d \geq 2$ be a constant. For any $\epsilon > 0$, there exists a data structure that supports $(1+\epsilon)$-approximate nearest neighbour queries over a subset of $\mathcal{U}^d$ in $O\big((1/\epsilon)^d \log\log\Delta\big)$ expected time. Insertions and deletions can be supported in expected amortized time $O(\log\log\Delta)$. The expected space required for this data structure is $O(n \log\log\log U)$.*

6.4.2 Range Searching

In this section, we consider answering range (reporting) queries on the grid $\mathcal{U}^2$. We wish to preprocess a set of points $S \subseteq \mathcal{U}^2$ into a data structure so that given a query region, we can enumerate all points in S that are contained in the query region.

This problem was studied in this setting by Overmars [54], where several different types of query regions are considered. Chan et al. [22] also address this problem and obtain data structures for (among other problems) the related problem of orthogonal *range emptiness* queries: one data structure achieves query time $O(\log\log n)$ and space $O(n \log\log n)$ for points given in rank space, while another data structure achieves space $O(n)$ and answers queries in $O(\log^\epsilon n)$ time.

Dominance Queries

A dominance query asks to enumerate the points that dominate[5] the query point. The query region can then be viewed as the intersection of the halfplanes above and to the right of the query point. To solve the dominance reporting problem, Overmars [54] maintains an x-fast trie with points ordered by x-coordinate.[6] At

[5] A point p dominates a point q if all coordinates of p are at least as large as those of q.

[6] By an observation of Overmars [54], we may assume that no points share any x- or y-coordinates.

each internal node of the trie, the point with the largest y-coordinate in that subtree is stored. Additionally, each leaf of the trie stores a list containing pointers to nodes that are right children of nodes on the path from the root to that leaf, sorted by y-coordinate, as well as a priority search tree that contains all points stored by the nodes on the path from the root to that leaf. Recall that priority search trees [49] answer half-infinite range queries in $O(\log n + k)$ time, where n is the number of points stored in the tree and k is the number of points in the query range, using linear space. Since each root-to-leaf path has length $O(\log U)$, the total space for this structure is $O(n \log U)$, although by bucketing into priority search trees, this can be reduced to $O(n)$.

To answer a dominance query, a search in the trie with the x-coordinate of the query point is performed. As usual, the search ends at some leaf of the trie. All points that dominate the query point must lie either on the path from the root to this leaf, or to the right of this path. The points that lie on the path can be found by querying the associated priority search tree in time $O(\log\log U + k_1)$, where k_1 is the number of points stored on nodes of the root-to-leaf path that dominate the query point. The remaining points can be found by examining the associated list of right children in decreasing order of y-coordinate until they no longer dominate the query point. By traversing these subtrees in a way similar to that of priority search trees, only subtrees that contain dominating points will be explored. This therefore takes $O(1 + k_2)$ time, where k_2 is the number of points to the right of the root-to-leaf path that dominate the query point. Since $k = k_1 + k_2$, a query time of $O(\log\log U + k)$ follows.

We can use the data structure of Theorem 6.8 in place of the x-fast trie. Ad-

ditionally, we store at each leaf another such structure on the points on the path from the root to the corresponding leaf of the x-fast trie that are to the right of the point stored at the leaf. We also store these points in a list ordered by decreasing y-coordinate. As before, we also store a list (ordered again by decreasing y-coordinate) of pointers to nodes of the x-fast trie that are right children of nodes on the root-to-leaf path. Now, consider a query point $q = (a, b) \in \mathcal{U}^2$. Let Δ_h denote the horizontal distance from the query point to the end of the grid and let Δ_v denote the vertical distance from the query point to the end of the grid, *i.e.*, $\Delta_h = U - a$ and $\Delta_v = U - b$. To answer the query, we first perform a successor[7] query for a in the main structure; this takes time $O(\log\log\Delta_h)$. Note that if no successor is found, then no points dominate the query point. We can then find the points on the path by performing a successor search for b in the auxiliary structure stored at the leaf we found in time $O(\log\log\Delta_v)$. As before, if no successor is found, then no points dominate the query point. At this point, we can find the rest of the points that dominate the query point that are on the path by walking along the ordered list of points on that path and stopping when we drop below b. All that remains is to find the points that dominate the query point that are on the right of the search path; this is done exactly as before. The total query time is thus $O(\log\log\Delta_h + \log\log\Delta_v + k) = O(\log\log(\Delta_h + \Delta_v) + k)$. The space used is $O(n \log U \log\log\log U)$: the main data structure uses $O(n\log\log\log U)$ space as before, the auxiliary lists each use $O(\log U)$ space and thus $O(n \log U)$ space in total, and the auxiliary structure uses space $O(\log U \log\log\log U)$, since each root-to-leaf path has $O(\log U)$ points. Since there are n root-to-leaf paths, the total space used

[7]Recall that successor queries can be supported in the same time bound as predecessor queries.

is $O(n \log U \log\log\log U)$. Note that bucketing (as in Section 6.1.2) does not seem to reduce the space without adversely affecting the query time; we elaborate on this point in Section 6.5.

Theorem 6.10. *Let $\mathcal{U} = \{0, 1, \ldots, U-1\}$ and consider a subset $S \subseteq \mathcal{U}^2$. There exists a data structure that reports the points $(x, y) \in S$ that dominate a query point $(a, b) \in \mathcal{U}^2$ in expected time $O(\log\log(\Delta_h + \Delta_v) + k)$ where $\Delta_h = U - a$ and $\Delta_v = U - b$ and k is the number of points dominated by the query point. The data structure uses expected space $O(n \log U \log\log\log U)$.*

Half-infinite Queries

For the case of half-infinite query regions, we consider queries that consist of two points $(a, c) \in \mathcal{U}^2$ and $(b, c) \in \mathcal{U}^2$ and wish to report all points $(x, y) \in S$ such that $a \leq x \leq b$ and $y \geq c$. Overmars [54] shows how to handle half-infinite query regions using a similar technique. Assume queries are unbounded in the positive y direction but bounded on the three remaining sides. As before, priority search trees are stored at each leaf that contain the points stored on each root-to-leaf path. Instead of a single sorted list at each leaf containing the right children of the nodes on that path, there are $O(\log U)$ such lists, where the i-th list contains the points below depth i (therefore, the first such list is exactly the list stored for the dominance reporting data structure discussed previously). Each list is sorted by y-coordinate. Symmetrically, each path contains similar lists except for the left children of nodes on the path. Since each of the $O(\log U)$ lists has size $O(\log U)$, the total space for this data structure is $O(n \log^2 U)$, which can again be reduced to $O(n)$ using buckets. Queries are performed by searching for both the left and right

sides of the query region to find where the search paths diverge in $O(\log\log U)$ time. The points in the query range are either on the search paths or in a right child of the left path or a left child of the right path. Points on the search paths can be found using the priority search trees stored in the leaves as before, and points between the paths can be found by looking at the lists in the leaves corresponding to the depth of the node where the search paths diverge. A query time of $O(\log\log U + k)$ follows.

As before, we construct the usual x-fast trie as well a main structure described by Theorem 6.8. At each leaf, we store two auxiliary structures: one on the points on the root-to-leaf path that are to the left of the point stored at the leaf, and one on the points on the root-to-leaf path that are to the right of the point stored at the leaf. Both auxiliary structures are ordered by y-coordinate. Auxiliary lists are also stored at each leaf as described for the original structure of Overmars [54]. Queries are answered by performing a successor query for a in the main structure and a predecessor query for b. We must modify the structure of Theorem 6.8 to "give up" the search after a fixed distance; this can be accomplished by simply stopping the exponential search when the distance covered is larger than the specified distance. In our case, the specified distance is $\Delta_h = b - a$. The successor and predecessor queries can thus be performed in expected time $O(\log\log\Delta_h)$. We then execute successor queries for c in both auxiliary structures in time $O(\log\log\Delta_v)$, where $\Delta_v = U - c$, and walk along the sorted lists to find points on the search path that are in the query region. The points to the left and right of the search path can be found using the same technique as that of Overmars [54]. The total query time is thus $O(\log\log\Delta_h + \log\log\Delta_v + k) = O(\log\log(\Delta_h + \Delta_v) + k)$. The space required

for this data structure is $O(n \log^2 U)$, since each auxiliary list has size $O(\log U)$ and each of the n leaves contains $O(\log U)$ such lists.

Theorem 6.11. *Let $\mathcal{U} = \{0, 1, \ldots, U-1\}$ and consider a subset $S \subseteq \mathcal{U}^2$. There exists a data structure that reports the points $(x, y) \in S$ that are contained in a half-infinite region defined by the points $(a, c) \in \mathcal{U}^2$ and $(b, c) \in \mathcal{U}^2$ such that $a \leq x \leq b$ and $y \geq c$ in expected time $O(\log\log(\Delta_h + \Delta_v) + k)$ where $\Delta_h = b - a$, $\Delta_v = U - c$ and k is the number of points reported. The data structure uses expected space $O(n \log^2 U)$.*

Note that the result of Theorem 6.11 can be easily adapted to the case of half-infinite regions that are unbounded in the negative y, positive x and negative x directions.

Four-Sided Queries

The final type of query we will consider is a four-sided (*i.e.*, rectangular) query. A query consists of two points $(a, c) \in \mathcal{U}^2$ and $(b, d) \in \mathcal{U}^2$ and we wish to report all points $(x, y) \in S$ such that $a \leq x \leq b$ and $c \leq y \leq d$ (*i.e.*, (a, c) and (b, d) define the corners of a rectangle). For such queries, the techniques of Overmars [54] described above can be combined with a technique of Edelsbrunner [35]. All points are stored in an x-fast trie, ordered by y-coordinate. Every internal node that is a left child of its parent stores a structure for answering half-infinite range queries among the points stored in that subtree, as described above. Every internal node that is a right child of its parent stores a structure for answering half-infinite range queries that are unbounded in the *negative* y direction (a simple modification of the previous data structure) among points stored in that subtree. Range queries are answered by searching for the y-coordinates of the query and determining where the search

path splits, which can be determined in $O(\log\log U)$ time. The auxiliary structures of the two children of that node are then queried to answer the original query in $O(\log\log U + k)$ time. Observe that each point is stored in exactly one auxiliary structure at each level; the total space is thus $O(n \log U)$.

This case is handled using an adaptation of the technique of Edelsbrunner [35] described by Overmars [54]. The structure works exactly like that of Overmars [54] except we replace the structures for half-infinite range queries with the data structure described by Theorem 6.11. The query time is therefore $O(\log\log(\Delta_h + \Delta_v) + k)$, where $\Delta_h = b - a$ and $\Delta_v = d - c$. Observe that every point is stored in $O(\log U)$ auxiliary structures, each of which use space $O(n \log^2 U)$ by Theorem 6.11. The total space is therefore $O(n \log^3 U)$.

Theorem 6.12. *Let $\mathcal{U} = \{0, 1, \ldots, U-1\}$ and consider a subset $S \subseteq \mathcal{U}^2$. There exists a data structure that reports the points $(x, y) \in S$ that are contained in a rectangular region defined by the points $(a, c) \in \mathcal{U}^2$ and $(b, d) \in \mathcal{U}^2$ such that $a \leq x \leq b$ and $c \leq y \leq d$ in expected time $O(\log\log(\Delta_h + \Delta_v) + k)$ where $\Delta_h = b - a$, $\Delta_v = d - c$ and k is the number of points reported. The data structure uses expected space $O(n \log^3 U)$.*

Reducing Space Usage

The results of Theorems 6.10, 6.11 and 6.12 can be improved by observing that most of the query can be resolved in rank space rather than the entire universe $\mathcal{U}$. In each case, the data structure is built on the set of points translated to rank space (which has size n), which produces data structures of expected size $O(n \log n \log\log\log n)$, $O(n \log^2 n)$ and $O(n \log^3 n)$, respectively.

Before a query can be executed, the query must first be translated into rank

space. This can be achieved using the data structure of Theorem 6.8, with the same modification to stop the searching algorithm after exceeding the dimensions of the query box as was done for Theorem 6.11. In each case, this data structure uses expected space $O(n \log\log\log U)$. We obtain the following corollaries:

Corollary 6.13. *Let $\mathcal{U} = \{0, 1, \ldots, U-1\}$. There exists a data structure that reports the points of a subset of $\mathcal{U}^2$ that dominate a query point $(a, b) \in \mathcal{U}^2$ in expected time $O(\log\log(h+v)+k)$, where $h = U - a$, $v = U - b$ and k is the number of points dominated by the query point. The data structure uses $O(n(\log n \log\log n + \log\log\log U))$ expected space.*

Corollary 6.14. *Let $\mathcal{U} = \{0, 1, \ldots, U-1\}$. There exists a data structure that reports the points (x, y) of a subset of $\mathcal{U}^2$ that are contained in a half-infinite range defined by the points (a, c) and (b, c) such that $a \leq x \leq b$ and $y \geq c$ in expected time $O(\log\log(h+v)+k)$, where $h = b - a$, $v = U - c$ and k is the number of points reported. The data structure uses $O\big(n(\log^2 n + \log\log\log U)\big)$ expected space.*

Corollary 6.15. *Let $\mathcal{U} = \{0, 1, \ldots, U-1\}$. There exists a data structure that reports the points (x, y) of a subset of $\mathcal{U}^2$ that are contained in a rectangular range defined by the points (a, c) and (b, d) such that $a \leq x \leq b$ and $c \leq y \leq d$ in expected time $O(\log\log(h+v)+k)$, where $h = b - a$, $v = d - c$ and k is the number of points reported. The data structure uses $O\big(n(\log^3 n + \log\log\log U)\big)$ expected space.*

6.5 Conclusion and Open Problems

In this chapter, we unified results on hashing with those of predecessor search by describing a data structure that supports predecessor queries in time that is a function

of the distance between the query element and the element returned. For membership queries, we match the results of hashing [33], while for predecessor queries we do no worse than the results of Willard [67]. Several applications were also considered, including dynamic approximate nearest neighbour search and range searching.

There are several possible directions for future research.

1. Is it possible to keep the query times stated in Theorem 6.8 using space $O(n)$? Any progress in this area would immediately improve Theorems 6.10, 6.11 and 6.12.

2. We rely on randomization (skip lists) and amortization (restructuring y-fast tries for the dynamic case). Is it possible to partially de-randomize or de-amortize the result of Theorem 6.8? One might consider replacing the skip list by the optimal finger search structure of Brodal et al. [19], but this seems to require handling restructuring the y-fast tries in a more complicated manner. Note that Theorem 6.8 implies $O(1)$ time for exact searches and therefore the lower bounds for hashing apply [33]: as a consequence, $O(1)$ *worst-case* time is not achievable for searches, insertions and deletions.

3. Is it possible to improve the space usage for the dynamic approximate nearest neighbour problem proved in Theorem 6.9 while maintaining the query time? Any approach that uses the structure described by Theorem 6.8 will use space $\Omega(n \log\log\log U)$.

4. Is it possible to improve the space usage for the range searching problem proved in Theorems 6.10, 6.11 and 6.12 while maintaining the query times?

Any approach that uses the structure described by Theorem 6.8 will use space $\Omega(n \log\log\log U)$.

Chapter 7

Biased Predecessor Search in Bounded Universes

In this chapter, we describe another method to perform predecessor searches in bounded universes in a distribution-sensitive manner. Recall that for the predecessor problem in a bounded universe, we are given a set of elements drawn from a larger universe of known size and wish to know the largest element that is less than or equal to (*i.e.*, the predecessor of) a given query drawn from the universe.

The time for a predecessor query will be a function of the probability distribution of the queries. Intuitively, queries from distributions that are highly non-uniform can be executed fast because the less likely queries can typically be ignored. Conversely, queries from uniform distributions correspond to the more usual analysis of data structures when we do not have *a priori* knowledge of the query distribution.

7.1 Problem Definition

As in Chapter 6, we are given a universe $\mathcal{U} = \{0, 1, \ldots, U-1\}$ of size U and wish to perform predecessor searches over a static subset $S = \{s_1, s_2, \ldots, s_n\} \subseteq \mathcal{U}$. Furthermore, we are given a probability distribution $D = \{p_0, p_1, \ldots, p_{U-1}\}$ such that the probability of receiving $i \in \mathcal{U}$ as a query is p_i. Since D is a probability distribution, we have that $\sum_{i=0}^{U-1} p_i = 1$.

Our goal is to preprocess $\mathcal{U}, S$ and D into a data structure so that the time to execute a query is related to D. The motivation for such results is the following. Let $H = \sum_{i=0}^{U-1} p_i \log(1/p_i)$ be the entropy of the distribution D. Recall that the entropy of a U-element distribution is between 0 and $\log U$. Therefore, a query time of $O(\log H)$ is at most $O(\log \log U)$, which matches the performance of, *e.g.*, van Emde Boas trees [64] and x- and y-fast tries [67]. However, for distributions with low entropy (*i.e.*, highly non-uniform distributions), we can still obtain constant query times. We call data structures with query times that are related to D *biased*.

We present several data structures in this chapter that can be classified based on their space requirements. We give one data structure that has query time $O(\log H)$ but uses space that is a function of U, as well as a data structure that uses only linear space but has query time $O\left(\sqrt{H}\right)$.

7.1.1 Background

The related literature for the predecessor search problem is the same as that discussed in Section 6.1.1, and so we concentrate primarily on biased data structures in this section.

As discussed in Section 2.1, the optimum binary search trees of Knuth [48] are the most efficient binary search tree possible for a given query distribution. However, computing this binary search tree is fairly time consuming. One therefore turns their attention to finding a binary search tree that answers queries in (expected) time $O(H)$; Mehlhorn [50] described such a data structure and subsequently improved this bound to $H + O(1)$ [51]. Even if *a priori* knowledge of the distribution is not available, it is still possible to achieve $O(H)$ expected query time [20, 61].

A similar result is that of biased search trees [10]. A biased search tree can answer a query in time $O(\log 1/p_i)$ (and therefore the average query time is $O(H)$). Perhaps the most natural way to frame the line of research in this chapter is by analogy: the results here are to biased search trees as van Emde Boas trees are to binary search trees.

7.2 Supporting Logarithmic Query Time

In this section, we describe how to achieve query time $O(\log H)$ using space that is a function of both of n and U, rather than only n.

7.2.1 Using $O(n + U^\epsilon)$ Space

Let $\epsilon > 0$. We will place all elements $i \in \mathcal{U}$ with probability $p_i \geq (1/U)^\epsilon$, along with their predecessor (which never changes since S is static) into a hash table T. All elements of S are also placed into a y-fast trie over the universe $\mathcal{U}$. Since there are at most U^ϵ elements with probability greater than $(1/U)^\epsilon$, it is clear that the hash

table requires $O(U^\epsilon)$ space. Since the y-fast trie requires $O(n)$ space, we have that the total space used by this structure is $O(n + U^\epsilon)$. To execute a search, we check the hash table first. If the query (and thus the answer) is not stored there, then a search is performed in the y-fast trie to answer the query.

The expected query time is thus

$$\begin{aligned}
& \sum_{i\in T} p_i O(1) + \sum_{i\in\mathcal{U}\setminus T} p_i O(\log\log U) \\
= \;& O(1) + \sum_{i\in\mathcal{U}\setminus T} p_i O(\log\log U) \\
= \;& O(1) + \sum_{i\in\mathcal{U}\setminus T} p_i O\left(\log\log (U^\epsilon)^{1/\epsilon}\right) \\
= \;& O(1) + \sum_{i\in\mathcal{U}\setminus T} p_i O(\log(1/\epsilon)\log U^\epsilon) \\
= \;& O(1) + \sum_{i\in\mathcal{U}\setminus T} p_i O(\log(1/\epsilon)) + \sum_{i\in\mathcal{U}\setminus T} p_i O(\log\log U^\epsilon) \\
= \;& O(1) + O(\log(1/\epsilon)) + \sum_{i\in\mathcal{U}\setminus T} p_i O\left(\log\log \frac{1}{1/U^\epsilon}\right) \\
\leq \;& O(1) + O(\log(1/\epsilon)) + \sum_{i\in\mathcal{U}\setminus T} p_i O(\log\log(1/p_i))
\end{aligned}$$

The last step here follows from the fact that, if $i \in \mathcal{U} \setminus T$, then $p_i \leq (1/U)^\epsilon$, and so $1/(1/U)^\epsilon \leq 1/p_i$. Recall Jensen's inequality, which states that for concave functions f, $E[f(X)] \leq f(E[X])$. Since the logarithm is a concave function, we therefore have

$$\sum_{i\in\mathcal{U}\setminus T} p_i O(\log\log(1/p_i)) \leq \log \sum_{i\in\mathcal{U}\setminus T} p_i O(\log(1/p_i)) \leq O(\log H)$$

therefore, the expected query time is $O(\log(1/\epsilon)) + O(\log H) = O(\log(H/\epsilon))$.

Theorem 7.1. *Suppose we are given a probability distribution with entropy H over the possible queries in a universe of size U. There exists a data structure that performs predecessor searches over a subset of $\mathcal{U}$ in expected time $O(\log(H/\epsilon))$ using $O(n + U^\epsilon)$ space for any $\epsilon > 0$.*

7.2.2 Using $O\left(n + 2^{\log^{1/c} U}\right)$ Space

At this point, we note that $O(U^\epsilon)$ space is fairly large. Our goal now is to reduce this space as much as possible. One observation is that we can more carefully select the threshold for "large probabilities" that we place in the hash table T. Instead of $(1/U)^\epsilon$, we can use $(1/2)^{\log^{1/c} U}$ for some $c > 1$. The space used by the hash table is thus $O\left(2^{\log^{1/c} U}\right)$, which is $o(U^\epsilon)$ for any $\epsilon > 0$. The analysis of the expected query times carries through as follows

$$\begin{aligned}
\sum_{i \in T} p_i\, O(1) + \sum_{i \in \mathcal{U} \setminus T} p_i O(\log\log U) &= O(1) + \sum_{i \in \mathcal{U} \setminus T} p_i O(\log\log U) \\
&= O(1) + \sum_{i \in \mathcal{U} \setminus T} p_i O\left(\log\log (U^{1/c})^c\right) \\
&= O(1) + \sum_{i \in \mathcal{U} \setminus T} p_i c O\left(\log\log U^{1/c}\right) \\
&= O(1) + c \sum_{i \in \mathcal{U} \setminus T} p_i O\left(\log\log U^{1/c}\right) \\
&= O(1) + c \sum_{i \in \mathcal{U} \setminus T} p_i O\left(\log\log 2^{\log U^{1/c}}\right) \\
&\leq O(1) + c \sum_{i \in \mathcal{U} \setminus T} p_i O(\log\log(1/p_i)) \\
&\leq O(c \log H)
\end{aligned}$$

Theorem 7.2. *Suppose we are given a probability distribution with entropy H over the possible queries in a universe of size U. There exists a data structure that performs predecessor searches over a subset of $\mathcal{U}$ in expected time $O(c \log H)$ using $O\left(n + 2^{\log^{1/c} U}\right)$ space for any $c > 1$.*

7.2.3 Alternate Solution

The same query time as in Theorem 7.2 can be achieved through other means, albeit with slightly higher space requirements. We present this alternate solution for two reasons. First, this solution allows for a smoother tradeoff as H increases: solutions so far all have the property that query times are $O(1)$ or $O(\log \log U)$, whereas the solution presented here has query time that is a function of the probability of the query element; we elaborate on this point after presenting the data structure. Second, this solution may provide a different approach to solving the problem at hand, possibly by use of the output entropy, which will be described in Section 7.3.

By Theorem 6.8, there exists a predecessor search data structure that has size $O(n \log \log \log U)$ that achieves $O(\log \log \Delta)$ expected query time, where Δ is the distance between the query element and the element returned. We add "fake" elements to S (which have their predecessors precomputed) to ensure that the distance between i and its predecessor is at most $\Delta_i \leq (1/p_i)^{\log^{c-1}(1/p_i)}$ for some $c \geq 2$. To perform a search, we query the data structure. If we happen to be returned a "fake" element, then we use its precomputed predecessor to answer the query. We then

we obtain the following expected running time

$$\begin{aligned}
\sum_{i\in\mathcal{U}} p_i O(\log\log\Delta_i) &\leq \sum_{i\in\mathcal{U}} p_i O\Big(\log\log(1/p_i)^{\log^{c-1}(1/p_i)}\Big) \\
&= \sum_{i\in\mathcal{U}} p_i O\big(\log\left(\log^{c-1}(1/p_i)\log(1/p_i)\right)\big) \\
&= \sum_{i\in\mathcal{U}} p_i O(\log\log^c(1/p_i)) \\
&= \sum_{i\in\mathcal{U}} p_i c O(\log\log(1/p_i)) \\
&= c\sum_{i\in\mathcal{U}} p_i O(\log\log(1/p_i)) \\
&\leq O(c\log H)
\end{aligned}$$

It remains to determine the amount of space required for this structure. To do this, we need to count the number of "fake" elements added to the data structure. Let us consider how such elements were added. For each element $i \in \mathcal{U}$, in order of the highest probability to lowest, we check to see if the predecessor of i is within distance $(1/p_i)^{\log^{c-1} 1/p_i}$. Observe that the the answer to this question is always "yes" if $(1/p_i)^{\log^{c-1} 1/p_i} \geq U$. We have

$$\begin{aligned}
& & (1/p_i)^{\log^{c-1} 1/p_i} &\geq U \\
\iff & & \log(1/p_i)^{\log^{c-1} 1/p_i} &\geq \log U \\
\iff & & \log^{c-1}(1/p_i)\log(1/p_i)^{1/p_i} &\geq \log U \\
\iff & & \log^c(1/p_i) &\geq \log U \\
\iff & & 1/p_i &\geq 2^{\log^{1/c} U} \\
\iff & & p_i &\leq 1/2^{\log^{1/c} U}
\end{aligned}$$

Of course, we must be careful about how many "fake" elements we add. The next lemma shows that this number is not too large.

Lemma 7.3. *If we ensure the distance between every element i and its predecessor is at most $(1/p_i)^{\log^{c-1} 1/p_i}$ for some $c \geq 2$, then at most $O\left(2^{\log^{1/c} U}\right)$ "fake" elements are added to the data structure.*

Proof. Observe that there are most two elements of U with probability between 1 and $1/2$, at most four elements with probability between $1/2$ and $1/4$, at most eight elements with probability between $1/4$ and $1/8$, and so on. In general, there are at most 2^{i+1} elements with probability between $1/2^i$ and $1/2^{i+1}$. In the worst case, we always place a "fake" element. Therefore, we place at most 2^{i+1} "fake" elements on elements with probabilities between $1/2^i$ and $1/2^{i+1}$, but we never place a "fake" element for elements with probability at most $1/2^{\log^{1/c} U}$. The number of "fake" elements is thus $\sum_{i=0}^{\log^{1/c} U} 2^i = O\left(2^{\log^{1/c} U}\right)$. □

By Lemma 7.3, the total space requirement is $O\left((n + 2^{\log^{1/c} U}) \log\log\log U\right)$.

This result has an extra factor of $O(\log\log\log U)$ in the space requirements, but has one interesting property that the structure in Theorem 7.2 does not. Observe that an individual query for element i can be executed in time $O(c \log\log 1/p_i)$ time. This is in contrast to Theorem 7.2, where all queries take either $\Theta(1)$ time or $\Theta(\log\log U)$ time because it is answered in either a hash table or a y-fast trie. As a result, this technique can be used to support arbitrarily weighted elements in $\mathcal{U}$. Suppose each element $i \in \mathcal{U}$ has a real-valued weight $w_i > 0$ and let $W = \sum_{i=0}^{U-1} w_i$. By assigning each element probability $p_i = w_i/W$, we achieve a query time of $O(c \log\log(W/w_i))$, which is analogous to the $O(\log W/w_i)$ query time of biased search trees [10].

Theorem 7.4. *Suppose we are given a positive real weight w_i for each element i in a universe $\mathcal{U}$ of size U, such that the sum of all weights is W. There exists a data structure that performs a predecessor search for item i in time $O(c \log\log(W/w_i))$ using $O\Big((n + 2^{\log^{1/c} U}) \log\log\log U\Big)$ space for any $c \geq 2$.*

7.3 Supporting Linear Space

In this section, we describe how to achieve space $O(n)$ by accepting a larger query time of $O\Big(\sqrt{H}\Big)$. We begin with a brief note concerning input entropy vs. output entropy.

Input vs. Output Distribution. Until now, we have discussed the *input* distribution, *i.e.*, the probability that $i \in \mathcal{U}$ is the *query*. We could also discuss the *output* distribution, *i.e.*, the probability that $i \in \mathcal{U}$ is the *answer* to the query. This distribution can be defined by letting $p_i^* = 0$ if $i \notin S$ and $p_i^* = \sum_{j=s_i}^{s_{i+1}-1} p_j$ otherwise.

Suppose we can answer a predecessor query for i in $\mathrm{tim}\Big(\log\log 1/p^*_{\mathrm{pred}(i)}\Big)_{\mathrm{d}(i)}\Big)$ where $\mathrm{pred}(i)$ is the predecessor of i. Then it follows that the expected query time is $\sum_{i\in\mathcal{U}} p_i O\Big(\log\log p^*_{\mathrm{pred}(i)}\Big)$. Since $p_i \leq p^*_{\mathrm{pred}(i)}$ for all i, we know that this is at most $\sum_{i\in\mathcal{U}} p_i O(\log\log 1/p_i)$, *i.e.*, at most the logarithm of the entropy of the input distribution. It therefore suffices to consider the output distribution.

Our data structure will use series of optimal data structures for predecessor search [8] that increase doubly-exponentially in size in much the same way as Bădoiu et al. [20]. Recall that Beame and Fich [8] presented an optimal data struc-

ture for predecessor search that executes queries in time

$$O\left(\min\left\{\frac{\log\log U}{\log\log\log U}, \sqrt{\frac{\log n}{\log\log n}}\right\}\right)$$

We will maintain several such structures $D_1, D_2, \ldots$, where D_j is over the entire universe $\mathcal{U}$ but contains only 2^{2^j} elements. In particular, D_j contains the 2^{2^j} elements of highest probability that are not contained in any D_k for $k < j$. Note that here, "highest probability" refers to the highest *output* probability.

Searches are performed by doing a predecessor search in each of $D_1, D_2, \ldots$. Along with each element we also store its successor. When we receive the predecessor of the query in D_j, we check its successor to see if that successor is larger than the query. If so, the predecessor in D_j is the answer to the query. Otherwise, the real predecessor is somewhere between the predecessor in D_j and the query, and can be found by continuing the search. This technique is essentially the same modification described by Bose et al. [12] to make the working-set structure [20] handle predecessor queries.

We now consider the search time in this data structure. Suppose the correct predecessor of the query i is found in D_j where $j > 1$ (otherwise, the predecessor was found in D_1 in $O(1)$ time). All $2^{2^{j-1}}$ elements of D_{j-1} have (output) probability greater than $p^*_{\text{pred}(i)}$, and so $p^*_{\text{pred}(i)} \leq 1/2^{2^{j-1}}$. Equivalently, j is $\log\log 1/p^*_{\text{pred}(i)} + O(1)$. The total time spent searching is

$$\sum_{k=1}^{j} O\left(\sqrt{\frac{\log 2^{2^k}}{\log\log 2^{2^k}}}\right) \leq O\left(\sqrt{\frac{2^j}{j}}\right) \leq O\left(\sqrt{\log 1/p^*_{\text{pred}(i)}}\right)$$

Therefore, since $p_i \leq p^*_{\text{pred}(i)}$ for all i, the expected query time is

$$\sum_{i \in \mathcal{U}} p_i O\left(\sqrt{\log 1/p^*_{\text{pred}(i)}}\right) \leq \sum_{i \in \mathcal{U}} p_i O\left(\sqrt{\log 1/p_i}\right) \leq O\left(\sqrt{H}\right)$$

The final step above follows from Jensen's inequality and the fact that the square root is a concave function. To determine the space used by this data structure, observe that every element stored in S is stored in exactly one D_j. Since each D_j uses space linear in the number of elements stored in it, the total space usage is $O(n)$.

Theorem 7.5. *Suppose we are given a probability distribution with entropy H over the possible queries in a universe of size U. There exists a data structure that performs predecessor searches over subsets of $\mathcal{U}$ in expected time $O\left(\sqrt{H}\right)$ using $O(n)$ space.*

Observe that we need not know the exact distribution D to achieve the result of Theorem 7.5; it suffices to know the sorted order of the keys in terms of non-increasing probabilities. Furthermore, since the result of Beame and Fich [8] is in fact dynamic, we can even obtain a bound similar to the working-set property, using the same technique as Bădoiu et al. [20]: a predecessor search for i can be answered in time $O\left(\sqrt{\log w(i)}\right)$ where $w(i)$ is the number of distinct predecessors reported since the last time the predecessor of i was reported. This result also uses $O(n)$ space.

7.4 Conclusion and Open Problems

In this chapter, we have introduced the idea of biased predecessor search. Two different categories of data structures are considered: one with query times that are logarithmic in the entropy of the query distribution (with space that is a function of U), and one with linear space (with query times that are strictly larger than logarithmic).

There are several possible directions for future research.

1. Is it possible to achieve $O(\log H)$ query time and $O(n)$ space?

2. The reason for desiring a $O(\log H)$ query time comes from the fact that $H \leq \log U$ and the fact that the usual data structures for predecessor searching have query time $O(\log \log U)$. Of course, the data structure of Beame and Fich [8] is faster than this. Is it possible to achieve a query time of, for example, $O(\log H / \log \log U)$?

3. What lower bounds can be stated in terms of either the input or output entropies? Clearly $O(U)$ space suffices for $O(1)$ query time, and so such lower bounds must place restrictions on space usage.

Chapter 8

Conclusion

In this final chapter, we summarize the contributions from the previous chapters as well as possible directions for future research.

8.1 Summary of Contributions

In this section, we summarize the contributions of this thesis.

In Chapter 3, we described the first binary search tree that has a stronger version of the working-set property which requires individual queries to be answered in time logarithmic in the working-set number in the worst-case. This binary search tree closes the gap between binary search trees that only have the working-set property in the amortized sense (such as splay trees [61]) and data structures which have this stronger version of the working-set property but are not binary search trees [20].

In Chapter 4, we developed a dictionary data structure with sensitivity to the distance between the query element and a temporal finger. This is the first dictionary

data structure that allows this temporal finger to be selected arbitrarily.

In Chapter 5, we strengthened the unified property so that query times need not be inflated with elements outside of the query's working-set. This is the first data structure to consider this property.

In Chapter 6, we unified results on hashing with results on predecessor search by describing a data structure with query time that is a function of the distance between the element queried and the element returned. These results are extended to the dynamic case. Several applications were considered, including the dynamic approximate nearest neighbour problem and the range searching problem (with several different types query regions).

In Chapter 7, we considered data structures for biased predecessor search. We described several data structures with query time that is logarithmic in the entropy of the distribution as well as a data structure with linear space. We also considered the case of weights on the elements of the universe instead of probabilities, as well as the case when the distribution of queries is not known in advance.

8.2 Open Problems

In this section, we summarize the open problems and directions for future work that have appeared in previous chapters.

1. In binary search trees, it seems that by enforcing good worst-case behaviour for temporally local query distributions, we sacrifice good performance on some other access sequences. Is it the case that a binary search tree that has this stronger version of the working-set property cannot achieve other

properties of, *e.g.*, splay trees? For example, what kind of sequential access bound can be achieved in this setting? Recall that the scanning bound refers to the amount of time required to execute all queries in sequential order.

2. Is it possible to answer queries in a binary search tree in worst-case time $O(\log |a_t - a_{t-1}|)$ (*i.e.*, a worst-case analogue of the dynamic finger property)? Recall that a_t and a_{t-1} are the current and previous queries, respectively. As with the working-set property, this is well-known outside of the binary search tree model.

3. Can the additive term in Theorem 4.1 be reduced? This would be interesting even for specific (ranges of) values of f. When $f = n$, for example, the best known result is $O(\log \tau_{t,n} n + \log \log n)$ [45]. The case when $f = 1$ is fully solved by Bădoiu et al. [20].

4. Is it possible to support multiple temporal fingers (*e.g.*, $O(1)$ many)? Simply searching the structures in parallel allows us to find the query element in time proportional to the logarithm of the minimum temporal distance, but it is not obvious how to quickly restructure the data structures and promote the query element to the cheapest substructure in parallel for each structure.

5. Is it possible to maintain the temporal finger property while supporting dynamic update operations?

6. One can reduce the distance measure $d_{W_t(w_t(x)^2)}(x, y)$ to $d_{W_t(w_t(x)^{1+\epsilon})}(x, y)$ in the strong unified property by changing how the substructures grow. Is it possible to reduce this further, say to $d_{W_t(O(w_t(x)))}(x, y)$?

7. We argued that it is not possible to use $d_{W_t(w_t(x))}(x, y)$ in the worst case for the strong unified property. Is this possible to achieve this in the amortized sense?

8. Can the additive term in Theorem 5.1 be reduced? It seems difficult to reduce this term below $\Omega(\log \log w_t(x))$ using an approach similar to the one presented here, since elements must shift through this many substructures.

9. Is it possible to maintain the strong unified property while supporting dynamic update operations?

10. Is it possible to keep the query times stated in Theorem 6.8 using space $O(n)$? Any progress in this area would immediately improve Theorems 6.10, 6.11 and 6.12.

11. For the distance-sensitive predecessor search problem, we rely on randomization (skip lists) and amortization (restructuring y-fast tries for the dynamic case). Is it possible to partially de-randomize or de-amortize the result of Theorem 6.8? One might consider replacing the skip list by the optimal finger search structure of Brodal et al. [19], but this seems to require handling restructuring the y-fast tries in a more complicated manner. Note that Theorem 6.8 implies $O(1)$ time for exact searches and therefore the lower bounds for hashing apply [33]: as a consequence, $O(1)$ worst-case time is not achievable for searches, insertions and deletions.

12. Is it possible to improve the space usage for the dynamic approximate nearest neighbour problem proved in Theorem 6.9 while maintaining the query time? Any approach that uses the structure described by Theorem 6.8 will use space $\Omega(n \log \log \log U)$.

13. Is it possible to improve the space usage for the range searching problem proved in Theorems 6.10, 6.11 and 6.12 while maintaining the query times? Any approach that uses the structure described by Theorem 6.8 will use space $\Omega(n \log\log\log U)$.

14. For the biased predecessor search problem, is it possible to achieve $O(\log H)$ query time and $O(n)$ space?

15. In the biased predecessor search problem, the reason for desiring a $O(\log H)$ query time comes from the fact that $H \leq \log U$ and the fact that the usual data structures for predecessor searching have query time $O(\log\log U)$. Of course, the data structure of Beame and Fich [8] is faster than this. Is it possible to achieve a query time of, for example, $O(\log H/\log\log U)$?

16. For the biased predecessor search problem, what lower bounds can be stated in terms of either the input or output entropies? Clearly $O(U)$ space suffices for $O(1)$ query time.

Bibliography

[1] G.M. Adelson-Velskii and E.M. Landis. An algorithm for the organization of information. *Soviet Math. Doklady*, 3:1259–1263, 1962.

[2] Peyman Afshani, Jérémy Barbay, and Timothy M. Chan. Instance-optimal geometric algorithms. In *FOCS '09: Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science*, pages 129–138, 2009.

[3] Arnon Amir, Alon Efrat, Piotr Indyk, and Hanan Samet. Efficient regular data structures and algorithms for dilation, location, and proximity problems. *Algorithmica*, 30(2):164–187, 2001.

[4] Arne A. Andersson and Mikkel Thorup. Dynamic ordered sets with exponential search trees. *Journal of the ACM*, 54(3), 2007.

[5] Sunil Arya, Theocharis Malamatos, David M. Mount, and Ka Chun Wong. Optimal expected-case planar point location. *SIAM Journal on Computing*, 37(2): 584–610, 2007.

[6] Amitabha Bagchi, Adam L. Buchsbaum, and Michael T. Goodrich. Biased skip lists. *Algorithmica*, 42:31–48, 2005.

[7] R. Bayer. Symmetric binary B-trees: Data structures and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[8] Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65(1):38–72, 2002.

[9] D. Belazzougui, A.C. Kaporis, and P.G. Spirakis. Random input helps searching predecessors. arXiv:1104.4353, 2011.

[10] Samuel W. Bent, Daniel D. Sleator, and Robert E. Tarjan. Biased search trees. *SIAM Journal on Computing*, 14:545–568, 1985.

[11] Prosenjit Bose, Karim Douïeb, and Stefan Langerman. Dynamic optimality for skip lists and B-trees. In *SODA '08: Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1106–1114, 2008.

[12] Prosenjit Bose, John Howat, and Pat Morin. A distribution-sensitive dictionary with low space overhead. In *Proceedings of the 11th International Symposium on Algorithms and Data Structures (WADS 2009), LNCS 5664*, pages 110–118, 2009.

[13] Prosenjit Bose, Karim Douïeb, Vida Dujmović, and Rolf Fagerberg. An $O(\log \log n)$-time competitive binary search tree with optimal worst-case access time. In *Proceedings of the Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 38–49, 2010.

[14] Prosenjit Bose, Karim Douïeb, Vida Dujmović, and John Howat. Layered

working-set trees. In *Proceedings of the 9th Latin American Theoretical Informatics Symposium (LATIN 2010), LNCS 6034*, pages 686–696, 2010.

[15] Prosenjit Bose, Karim Douïeb, Vida Dujmović, John Howat, and Pat Morin. Fast local searches and updates in bounded universes. In *Proceedings of the 22nd Canadian Conference on Computational Geometry (CCCG 2010)*, pages 261–264, 2010.

[16] Prosenjit Bose, Karim Douïeb, Vida Dujmović, and John Howat. Layered working-set trees. *Algorithmica*, 63(1):476–489, 2012.

[17] Prosenjit Bose, Karim Douïeb, Vida Dujmović, John Howat, and Pat Morin. Fast local searches and updates in bounded universes. *Computational Geometry: Theory and Applications*, 2012. doi:10.1016/j.comgeo.2012.01.002.

[18] Gerth Stølting Brodal, Christos Makris, Spyros Sioutas, Athanasios Tsakalidis, and Kostas Tsichlas. Optimal solutions for the temporal precedence problem. *Algorithmica*, 33(4):494–510, 2002.

[19] Gerth Stølting Brodal, George Lagogiannis, Christos Makris, Athanasios Tsakalidis, and Kostas Tsichlas. Optimal finger search trees in the pointer machine. *Journal of Computer and System Sciences*, 67(2):381–418, 2003.

[20] Mihai Bădoiu, Richard Cole, Erik D. Demaine, and John Iacono. A unified access bound on comparison-based dynamic dictionaries. *Theoretical Computer Science*, 382(2):86–96, 2007.

[21] Timothy M. Chan. Closest-point problems simplified on the RAM. In *SODA '02:*

Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms, pages 472–473, 2002.

[22] Timothy M. Chan, Kasper Green Larsen, and Mihai Pătraşcu. Orthogonal range searching on the RAM, revisited. In *SoCG '11: Proceedings of the 27th Annual ACM Symposium on Computational Geometry*, pages 1–10, 2011.

[23] Richard Cole. On the dynamic finger conjecture for splay trees. Part II: The proof. *SIAM Journal on Computing*, 30(1):44–85, 2000.

[24] Richard Cole, Bud Mishra, Jeanette Schmidt, and Alan Siegel. On the dynamic finger conjecture for splay trees. Part I: Splay sorting $\log n$-block sequences. *SIAM Journal on Computing*, 30(1):1–43, 2000.

[25] Sébastien Colette, Vida Dujmović, John Iacono, Stefan Langerman, and Pat Morin. Distribution-sensitive point location in convex subdivisions. In *SODA '08: Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 912–921, 2008.

[26] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[27] Erik D. Demaine, John Iacono, and Stefan Langerman. Proximate point searching. *Computational Geometry: Theory and Applications*, 28(1):29–40, 2004.

[28] Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Pătraşcu. Dynamic optimality—almost. *SIAM Journal on Computing*, 37(1):240–251, 2007.

[29] Jonathan Derryberry, Don Sheehy, Maverick Woo, and Danny Dominic Sleator. Achieving spatial adaptivity while finding approximate nearest neighbors. In *CCCG '08: Proceedings of the 20th Annual Canadian Conference on Computational Geometry*, pages 163–166, 2008.

[30] Jonathan C. Derryberry. *Adapative Binary Search Trees.* PhD thesis, Carnegie Mellon University, 2009.

[31] Jonathan C. Derryberry and Daniel D. Sleator. Skip-splay: Toward achieving the unified bound in the BST model. In *WADS '09: Proceedings of the 16th Annual International Workshop on Algorithms and Data Structures*, pages 194–205, 2009.

[32] Paul F. Dietz and Rajeev Raman. A constant update time finger search tree. *Information Processing Letters*, 52(3):147–154, 1994.

[33] Martin Dietzfelbinger, Anna R. Karlin, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert Endre Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994.

[34] Vida Dujmović, John Howat, and Pat Morin. Biased range trees. In *SODA '09: Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 486–495, 2009.

[35] H. Edelsbrunner. A note on dynamic range searching. *Bulletin of the EATCS*, 15:34–40, 1981.

[36] Amr Elmasry. A priority queue with the working-set property. *International Journal of Foundations of Computer Science*, 17(6):1455–1465, 2006.

[37] Amr Elmasry, Arash Farzan, and John Iacono. A unifying property for distribution-sensitive priority queues. In *Combinatorial Algorithms: 22nd International Workshop, IWOCA 2011*, pages 209–222, 2011.

[38] Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *FOCS '78: Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science*, pages 8–21, 1978.

[39] Sariel Har-Peled. *Geometric approximation algorithms*. American Mathematical Society, 2011.

[40] John Iacono. New upper bounds for pairing heaps. In *Scandinavian Workshop on Algorithm Theory (LNCS 1851)*, pages 32–45, 2000.

[41] John Iacono. *Distribution Sensitive Data Structures*. PhD thesis, Rutgers, The State University of New Jersey, 2001.

[42] John Iacono. Key-independent optimality. *Algorithmica*, 42(1):3–10, 2005.

[43] John Iacono. A static optimality transformation with applications to planar point location. arXiv:1104.5597, 2011.

[44] John Iacono and Stefan Langerman. Proximate planar point location. In *SoCG '03: Proceedings of the 19th Annual ACM Symposium on Computational Geometry*, pages 220–226, 2003.

[45] John Iacono and Stefan Langerman. Queaps. *Algorithmica*, 42(1):49–56, 2005.

[46] Donald B. Johnson. A priority queue in which initialization and queue operations take $O(\log \log D)$ time. *Theory of Computing Systems*, 15(1):295–309, 1981.

[47] Alexis Kaporis, Christos Makris, Spyros Sioutas, Athanasios Tsakalidis, Kostas Tsichlas, and Christos Zaroliagis. Improved bounds for finger search on a RAM. In *ESA '03: Proceedings of the 11th Annual Eureopean Symposium on Algorithms, LNCS 2832*, pages 325–336, 2003.

[48] D.E. Knuth. Optimum binary search trees. *Acta Informatica*, 1:14–25, 1971.

[49] E.M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2): 257–276, 1985.

[50] Kurt Mehlhorn. Nearly optimal binary search trees. *Acta Informatica*, 5(4): 287–239, 1975.

[51] Kurt Mehlhorn. A best possible bound for the weighted path length of binary search trees. *SIAM Journal on Computing*, 6(2):235–295, 1977.

[52] Kurt Mehlhorn and Stefan Näher. Bounded ordered dictionaries in $O(\log \log N)$ time and $O(n)$ space. *Information Processing Letters*, 35(4):183–189, 1990.

[53] Yakov Nekrich. Data structures with local update operations. In *SWAT '08: Proceedings of the 11th Scandinavian Workshop on Algorithm Theory*, pages 138–147, 2008.

[54] Mark H. Overmars. Efficient data structures for range searching on a grid. *Journal of Algorithms*, 9(2):254–275, 1988.

[55] Mihai Pătraşcu and Mikkel Thorup. Time-space trade-offs for predecessor search. In *STOC '06: Proceedings of the 38th Annual ACM Symposium on Theory of Computing*, pages 232–240, 2006.

[56] Mihai Pătraşcu and Mikkel Thorup. Randomization does not help searching predecessors. In *SODA '07: Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 555–564, 2007.

[57] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[58] Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996.

[59] C.E. Shannon. A mathematical theory of communication. *Bell Systems Technical Journal*, 27:379–423 and 623–565, 1948.

[60] Daniel Dominic Sleator and Robert Endre Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[61] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[62] R.E. Tarjan. Sequential access in splay trees takes linear time. *Combinatorica*, 5:367–378, 1985.

[63] Robert Endre Tarjan. *Data structures and network algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983.

[64] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

[65] Chengwen Chris Wang, Jonathan Derryberry, and Daniel Dominic Sleator. $O(\log \log n)$-competitive dynamic binary search trees. In *SODA '06: Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 374–383, 2006.

[66] Robert Wilber. Lower bounds for accessing binary search trees with rotations. *SIAM Journal on Computing*, 18(1):56–67, 1989.

[67] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983.