Two Results on Distribution-Sensitive Data Structures

By

John Howat

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario

July 2008

Your file Votre référence
ISBN: 978-0-494-44107-7
Our file Notre référence
ISBN: 978-0-494-44107-7

Abstract

Distribution-sensitive data structures take advantage of underlying patterns in a sequence of operations in order to reduce their running times. The distribution-sensitive dictionary problem, though well-studied, often uses additional space in the form of pointers. We present a dictionary data structure that eliminates a great deal of space overhead while maintaining a notion of distribution-sensitivity in the expected sense. The problem of distribution-sensitive one-dimensional range searching is not well-studied in the literature, however. Before we show how to construct a distribution-sensitive data structure for this problem, we first define the kinds of query distributions to which our data structure will be sensitive, namely a variant of the working set property.

Acknowledgements

I would first like to thank my supervisors, Prosenjit Bose and Pat Morin, for their support throughout my studies. They have proven to be excellent supervisors and were always able to point me to relevant literature and offer insightful comments on my work. In fact, they were kind enough to provide me with opportunities for research even before my graduate studies began, and for this I am particularly grateful. My thanks are also due to Michiel Smid for his helpful comments on this thesis.

I also thank the other members of the Computational Geometry Laboratory at Carleton University for providing a stimulating environment in which to work, along with all of the fine chocolates and espresso necessary to keep graduate students like myself productive.

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

Contents

Abstract ii

Acknowledgements iii

1 Introduction 1

1.1 Motivation 2

1.2 Problem Statements 3

1.2.1 Dictionary Operations with Low Space Overhead 3

1.2.2 Range Searching in One Dimension 4

1.3 Contributions 4

1.4 Organization of the Thesis 5

2 Background 6

2.1 Distribution-Sensitive Data Structures 6

2.1.1 Optimum Binary Search Trees 7

2.1.2 Splay Trees 7

2.1.3 Applications of Distribution-Sensitivity 9

2.1.4 The Working Set Structure 10

2.2 Implicit Data Structures 12

2.2.1 Definition and Issues 13

2.2.2 Examples of Implicit Data Structures 14

2.3 Range Searching 15

2.3.1 Data Structures for Range Searching 16

2.3.2 Range Searching in One Dimension 17

2.4 Other Results 18
2.4.1 Treaps 18
2.4.2 Bounded-Universe Dictionaries 18
2.4.3 Doubling Search 19
2.5 Expectation and Amortization 20
2.5.1 Expected Running Time 20
2.5.2 Amortized Running Time 20

3 Dictionaries with Low Space Overhead **23**
3.1 Introduction 23
3.2 Modifying the Working Set Structure 24
3.2.1 Removing the Queues 24
3.2.2 Shrinking the Trees 27
3.3 Supporting Predecessor Searches 28
3.3.1 The Naïve Approach 28
3.3.2 A Tradeoff 29
3.4 Conclusion 31

4 Range Searching in One Dimension **33**
4.1 Introduction 33
4.1.1 The Dynamic Finger Property 34
4.2 Generalizing the Working Set Property 34
4.2.1 Defining the Working Set Number 35
4.2.2 Notes on the Working Set Number 36
4.3 The Data Structure 37
4.3.1 Overview 37
4.3.2 Shifting 39
4.3.3 Searching 39
4.3.4 Analysis 42
4.3.5 Improvements 44
4.3.6 Refining the Analysis 45
4.4 Supporting Counting Queries 47

4.5 Conclusion 49

5 Conclusion **50**

5.1 Summary of Results 50

5.2 Future Work 51

Bibliography **53**

List of Figures

4.1 Determining the working set number of an interval 36
4.2 A schematic of the range reporting data structure 38
4.3 The intervals found and reported during a search 40
4.4 The analysis from Theorem 4.3 is tight 46

Chapter 1

Introduction

In this thesis, we address the notion of distribution-sensitivity in data structures. Data structures are often analyzed in terms of the worst-case time for operations performed on them. Data structures for the dictionary problem, for instance, provide the operations INSERT, DELETE and SEARCH on a set of totally ordered elements. Several such data structures exist. For example, balanced binary trees such as AVL trees [1] and red-black trees [22] both perform all three dictionary operations in worst-case time $O(\log n)$ when n elements are stored in the dictionary. As is well-known, these results are optimal in the comparison tree model [30, page 412].

After analyzing the worst-case cost per operation on a data structure, we might be tempted to bound the cost of a sequence of operations based on the worst-case cost per operation. However, this might be an overestimate. Even in the case of the dictionary problem, where the worst-case cost of $O(\log n)$ is in fact optimal, we still may be able to do better over the sequence as a whole. Observe that when analyzing a sequence of operations, we are effectively being given additional information: the pattern of operations itself. Distribution-sensitive data structures take advantage of underlying patterns in a sequence of operations in order to reduce the time complexity of the sequence as a whole.

In this chapter, we begin by motivating the use of distribution-sensitive data structures in Section 1.1. The particular problems we address in this thesis are outlined in Section 1.2, and the contributions of this thesis are discussed in Section 1.3.

The chapter ends with an overview of the remainder of the thesis in Section 1.4.

1.1 Motivation

The idea of taking advantage of a pattern in a sequence of queries to a data structure is a very useful one. In many applications, queries have some kind of pattern; it is rare for queries to be uniformly random. A typical example that motivates the usefulness of distribution-sensitivity is that of databases.

Imagine an employee database that stores employees' names, salaries, departments, and so on. If one needed to create a report about the number of individuals at different salary levels by department, they would probably proceed by making a number of queries to the database. The first query might be to find the number of employees in the sales department who make between $50,000 and $55,000, then $55,000 and $60,000, then $60,000 and $65,000, and so on. This process would then get repeated for, say, the marketing department, the accounting department, and so on. This is in contrast to querying a random salary range and a random department until all possibilities are covered. The systematic approach is seemingly a much more "natural" sequence of queries to make, and so if possible, we would like to make it execute quickly. There is hope for this, since there is, in fact, an underlying pattern: two queries are likely to be "close" to each other in some sense, because they tend to differ only by a salary range or a department.

One might also imagine a database for a registrar at a university. While all students–past and present–must have their records stored in the database, current students are much more likely to make queries into the database. Furthermore, on the rare occasion that an alumnus makes a query, they are probably unlikely to make another one any time soon, while if a current student makes a query, they are much more likely to make an additional one soon (during course registration, for instance). Thus, the sequence of queries is skewed towards those that have recently made queries. We would thus like queries that have been made recently to run fast if they are made again soon. This is another example of a natural pattern we might find in a sequence of queries.

The patterns in a sequence of queries need not be so subtle. In fact, the most obvious pattern might be to be given explicit *a priori* knowledge of the frequency of queries in the sequence. Ideally, we would like the most frequently executed query to be executed the fastest. Unfortunately, such knowledge of the query distribution is rarely given in advance.

In any case, taking advantage of patterns in the distribution of queries to a data structure can result in reduced time costs and is thus useful in areas such as databases.

1.2 Problem Statements

This thesis proposes data structures for two different problems. The first is that mentioned above, namely the dictionary operations. The second problem is the related but distinct problem of range searching in one dimension. In this section, we outline the problems addressed in this thesis.

1.2.1 Dictionary Operations with Low Space Overhead

As we shall see in the next chapter, distribution-sensitive dictionaries are already a fairly well-studied problem. However, current solutions to this problem are not space-efficient. While many such dictionaries use space linear in the number of items stored in the dictionary, they still require the use of a linear number of pointers.

Consider a standard implementation of a binary search tree. Each node in the tree might have two pointers: one pointer to its left child, and one pointer to its right child. For n nodes, this is a total of $2n$ pointers. If we assume that the tree stores machine words in the nodes, each node thus requires 3 words and the total size of the tree is thus $3n$ words. Therefore, to store n elements we must use three times the storage required by the data. Viewed another way, two-thirds of our storage is devoted entirely to pointers!

The problem we address is to reduce or eliminate the use of pointers while maintaining some sense of distribution-sensitivity.

1.2.2 Range Searching in One Dimension

Related to the dictionary problem is the problem of range searching. Instead of specifying a particular element we would like to search for, we instead specify a range of elements and ask that everything in that range be reported. We consider only the one-dimensional case, where the data stored are real numbers and the queries are intervals on the real line.

Unlike the dictionary problem, distribution-sensitive range searching is not a well-studied problem. Before we present a data structure for the problem, we start by defining the type of patterns our data structure will take advantage of.

The problem we address is to define the type of pattern the data structure will be sensitive to in a useful way, and then construct a data structure that is sensitive to such a pattern.

1.3 Contributions

The contributions of this thesis are two-fold.

Distribution-sensitive dictionaries with low space overhead. We present a dictionary with worst-case insertion and deletion times $O(\log n)$ and expected search time $O(\log t(x))$, where x is the element being searched for and $t(x)$ is the number of distinct accesses to the dictionary since x was last accessed. The space overhead for this data structure is $O(\log \log n)$, *i.e.*, $O(\log \log n)$ additional words of memory aside from the data itself are required. Current data structures that can match this query time include the splay tree [37] (in the amortized sense) and the working set structure [26] (in the worst case), although these require $3n$ and $5n$ pointers, respectively.[1]

Distribution-sensitive range searching in one dimension. We present a data structure for one-dimensional range searching that answers queries in amortized expected time $O(\log t(x) + k) + o(\log n)$, where k is the number of points reported. Here,

[1] This assumes that nodes in binary search trees use three pointers: one to each child and one to the parent.

$t(x)$ is analogous to the $t(x)$ defined above, but it is further multiplied by the number of points reported the last time an interval intersected with the query interval. We also show how to modify this data structure to support range counting queries in amortized expected time $O(\log t(x) + \log k) + o(\log n)$. Both of these data structures are of linear size.

1.4 Organization of the Thesis

The rest of the thesis is organized in the following way. Chapter 2 presents a review of the literature for the results used in the remainder of the thesis. In particular, we more rigorously discuss types of distribution-sensitivity, distribution-sensitive data structures, implicit data structures, and range searching. We also review the state-of-the-art for distribution-sensitive dictionaries and for implicit data structures, since the results of this thesis depend on them.

Chapter 3 shows how to construct a data structure to solve the first problem mentioned above, namely that of reducing the space-overhead of a distribution-sensitive dictionary. Chapter 4 solves the second problem: distribution-sensitive range searching in one dimension.

The thesis concludes with Chapter 5 which summarizes our results and gives several possible directions for future research.

Chapter 2

Background

In this chapter, we present a review of the literature pertaining to the results of the thesis. In particular, Section 2.1 discusses the notion of distribution-sensitive data structures, Section 2.2 discusses implicit data structures, and Section 2.3 gives an overview of data structures for the range searching problem. We also discuss two other data structures that we will need to use in this thesis in Section 2.4, as well as some tools for the analysis of algorithms–expectation and amortization–in Section 2.5.

2.1 Distribution-Sensitive Data Structures

In this section, we discuss some results from the literature on distribution-sensitive data structures. We briefly discuss optimum binary search trees and then turn our attention to splay trees and their associated properties such as static optimality, the working set property, the static and dynamic finger properties, and key independent optimality. Splay trees are also conjectured to have the dynamic optimality and unified properties, which we also discuss. We also survey some existing distribution sensitive data structures.

2.1.1 Optimum Binary Search Trees

Distribution-sensitive data structures have been studied since at least 1971, when Knuth described optimum binary search trees [29]. As the name implies, these are binary search trees which achieve optimal query times over a sequence of operations in the sense that they perform no more comparisons than any other comparison-based data structure on the same elements.

Unfortunately, there are two very important restrictions placed on optimum binary search trees. The first restriction is that they are static: after they are constructed initially, no elements may be added or removed from them. This limitation stems from the fact that the tree constructed is done so using an $O(n^2)$ dynamic programming algorithm. Therefore, of the three dictionary operations INSERT, DELETE and SEARCH, only SEARCH is supported. The second restriction is that the frequencies of queries for each element must be specified in advance.

2.1.2 Splay Trees

The splay tree was introduced by Sleator and Tarjan [37] and overcomes both of the limitations of optimum binary search trees while providing several additional nice properties. Splay trees are binary search trees that make use of restructuring rules after a query is executed. Such restructuring rules take the form of rotations. We do not describe the exact rules here, except to note that they serve to keep frequently accessed elements near the root of the tree.

Splay trees guarantee an amortized[1] running time of $O(\log n)$ for all three dictionary operations, though the worst-case cost of a query could be as much as $\Theta(n)$ if the tree becomes sufficiently unbalanced. In particular, a sequence of m operations on a splay tree with a total of n elements takes $O((n+m)\log n + m)$ time. However, there are many other interesting bounds on specific access sequences.

Splay trees, like optimum binary search trees, make no more comparisons than any other comparison-based data structure, to within a constant factor. This theorem about splay trees, called the *static optimality theorem*, was proved by Sleator and

[1] See Section 2.5 for an explanation of amortized query times.

Tarjan [37]. Observe that splay trees achieve this without *a priori* knowledge of the access frequencies, *i.e.*, the probability that a particular key will be queried. It is also conjectured that splay trees execute any sequence as asymptotically fast as other binary search tree, even when arbitrary rotations are allowed after accesses. This is known as the *dynamic optimality conjecture* and is a topic of ongoing research. Since it has proved rather difficult to show that splay trees are dynamically optimal, other data structures have been proposed that have been shown to be a sub-logarithmic factor from being dynamically optimal [14, 41].

One of the properties of splay trees that we will be using for the remainder of the thesis is the working set property. We slightly modify the notation of Iacono [26].

Definition 2.1 (Working set number)**.** Suppose we are given a sequence of queries $q_1, q_2, \ldots, q_m$ to a data structure containing n elements. Let $l_i(x) = \min\{\{\infty\} \cup \{j > 0 : q_{i-j} = x\}\}$. The *working set number* $t_i(x)$ is defined as

$$t_i(x) = \begin{cases} n & \text{if } l_i(x) = \infty \\ |\{q_{i-l_i(x)+1}, \ldots, q_i\}| & \text{otherwise} \end{cases}$$

An equivalent definition is that $t_i(x)$ is the number of distinct queries made between the i-th query in the access sequence and the last time x was queried, unless x has not yet been accessed in which case $t_i(x) = n$. If there are n elements currently stored in the data structure, it is clear that for any x, we have $1 \leq t_i(x) \leq n$. When working through an access sequence, we may write $t(x)$ to denote the working set number of x at the point in the access sequence corresponding to the current query.

As Sleator and Tarjan [37] observe, it would be beneficial if the time required for a query was sensitive to the working set number of that query. The motivation for such sensitivity stems from the following idea. Suppose we are given a dictionary D, but all of the queries we make to D are made to some $D' \subset D$. If this is the case, it would be nice to have query times as a function of $|D'|$ instead of $|D|$, since we can essentially ignore everything in $D \setminus D'$. Here, D' forms what is called a *working set*, and the working set number of any element in D' is at most $|D'|$. Thus, if our query time respects the working set number, our query time will be a function of $|D'|$,

as desired. In fact, the working set property implies the static optimality property discussed previously [25].

In our specific context, the *working set property* says that the query time for x is $O(\log t(x))$. It was shown by Sleator and Tarjan [37] that splay trees guarantee such a query time in the amortized sense, while the working set structure [26] guarantees this query time in the worst case.

Splay trees also have other distribution-sensitive properties. The *static finger property* states that query times are logarithmic in the rank distance between the current query and some fixed element in the tree [37]. This is in contrast to the *dynamic finger property* which states that query times are logarithmic in the rank distance between the current query and the previous query [11, 10]. Both of these properties hold for splay trees only in an amortized sense.

Splay trees are also *key independently optimal* [23]. This property roughly states that in the case where all key values in the tree are randomly assigned, then splay trees are essentially optimal in that no other binary search tree can execute the sequence faster than splay trees, except perhaps by a constant factor. Interestingly, this property is asymptotically equivalent to the working set property mentioned previously [23].

It is conjectured that splay trees also have the *unified property*[2] [26], which is a combination of the working set and dynamic finger properties. The unified property states that an access is fast if its rank distance with a recently-accessed element is small. Although it is not currently known if splay trees have this property, there is a data structure which does [26].

2.1.3 Applications of Distribution-Sensitivity

There are several examples of distribution-sensitive data structures in the literature.

Point Searching. In the planar *point searching* problem, we are given a set S of n points in the plane to preprocess so that we can quickly determine which (if

[2]As noted by Iacono [26], a "unified theorem" is also presented by Sleator and Tarjan [37]. The unified property discussed here is distinct.

any) point in S is equal to a given query point. A distribution-sensitive data structure is presented by Demaine et al. [13] that allows queries to be answered in time $O(\log d(q_i, q_{i-1}))$, where q_i is the current query and q_{i-1} is the previous query, and d is some distance function that describes the number of points in the neighbourhood of q_i and q_{i-1}. Thus, this result is a two-dimensional analogue of the dynamic finger property discussed above. To get the query times given, some restrictions must be replaced on the distance function d; a description of these restrictions goes beyond our scope.

Point Location. In the planar *point location* problem, we are given a map on the plane consisting of many cells and preprocess it to quickly determine which cell a given query point is in. Given the query distribution in advance, it is possible to construct a data structure that solves the problem on triangulations optimally up to a constant factor [3, 2, 24]. For convex subdivisions in the plane, the problem can also be solved optimally up to a constant factor [12]. If the query distribution is not given in advance, it is still possible to achieve distribution sensitivity [27]: there exists a data structure that answers queries in time $O(\log d(q_i, q_{i-1}))$, where q_i is the current query and q_{i-1} is the previous query, and d is some distance function that counts the number of triangles in a region defined by q_i and q_{i-1}.

Queaps. A complementary idea to the working set property is that of the *queueish property* [28]. Define $q(x) = n - t(x) - 1$; we say a data structure has the queueish property if the search time is $O(\log q(x))$. Iacono and Langerman [28] present a *queap*, which is a heap (priority queue) with the queueish property. The same authors also present a dictionary data structure that has a slightly weaker version of the queueish property, which states that the time for a query is $O(\log q(x)) + o(\log n)$ [28].

2.1.4 The Working Set Structure

The *working set structure* [26] is a dictionary data structure with the working set property. Since it is an integral part of both of the contributions of this thesis, we describe it here.

The working set structure stores a totally ordered set of n elements and consists of k balanced binary search trees $T_1, T_2, \ldots, T_k$ and k queues $Q_1, Q_2, \ldots, Q_k$. Each queue Q_i contains precisely the same elements as its corresponding tree T_i. The size of each tree and queue is the square of the previous one, so that $|T_i| = |Q_i| = 2^{2^i}$ for $1 \leq i < k$. T_k and Q_k contain all remaining elements, so that $|T_k| = |Q_k| = n - \sum_{i=1}^{k-1} 2^{2^i}$. Since there are n elements, we have $k = O(\log \log n)$.

Operations in the structure are facilitated with a *shift* operation. A shift is performed between two indices i and j and results in one element from T_i and Q_i being removed and a new element being placed in T_j and Q_j. This is performed in the following manner. If $i < j$, then we dequeue an element x from Q_i and delete x from T_i, and we insert x into T_{i+1} and enqueue it in Q_{i+1}. We now perform a shift from $i+1$ to j in the same manner, except that each time the element to be removed is that which is dequeued. This process continues until $i = j$ at which point we do nothing and stop. The process is symmetric if $i > j$: instead of inserting x into T_{i+1} and enqueueing it in Q_{i+1}, we instead insert it into T_{i-1} and enqueue it in Q_{i-1} and then continue shifting from $i - 1$ to j.[3] After a shift operation, the number of elements in T_i and Q_i has decreased by one, and the number of elements in T_j and Q_j has increased by one. The size of all other structures remain unchanged, although their contents may be different.

To search for an element in the working set structure, we search in $T_1, T_2, \ldots, T_k$ until it is found in T_i. If the element is not found, nothing is done. Assuming the element is found, we then delete it from T_i and remove it from Q_i and insert the element in T_1 and enqueue it in Q_1. At this point, T_1 and Q_1 have one too many elements and T_i and Q_i have one too few elements, so a shift from 1 to i is performed to restore the sizes.

Insertions are performed by inserting the element into T_1 and enqueuing it in Q_1. At this point, T_1 and Q_1 have too many elements. However, since every other tree and queue has the correct number of elements, the only place to shift to is the last index. Thus, a shift from 1 to k is performed, which may involve incrementing k if

[3]Because we are removing elements from the tail of the queue instead of the head, we must use a doubly-linked queue, and thus there are two pointers needed per element of each queue.

T_k grows to size 2^{2^k}.

Deletions are performed by searching in $T_1, T_2, \ldots, T_k$ until the element to be deleted is found in T_i. If the element is not found, nothing is done. Assuming the element is found, it is deleted from T_i and removed from Q_i. At this point, T_i and Q_i have one too few elements, but every other tree has the correct number of elements. Thus, the only place to shift from is the last index. Thus, a shift from k to i is performed.

Because of the doubly exponential sizes of the trees, we note that insertion and deletion use $O(\log n)$ time. It remains to show that the search time satisfies the working set property.

To see this, consider a search for x. Suppose that x is found in T_i and assume that $i \geq 2$ (otherwise, x is found in $O(1)$ time). If x is in T_i, then it must have been shifted there from T_{i-1} at some point. If x was shifted from T_{i-1}, then it must have done so by being dequeued from Q_{i-1}, which could only have happened after $2^{2^{i-1}}$ other dequeue operations, which corresponds to $2^{2^{i-1}}$ queries. Therefore, $t(x) \geq 2^{2^{i-1}}$. Now, due to the doubly exponential sizes of the trees, the search time for x is dominated by the search time in the last tree that is searched, *i.e.*, T_i. Since T_i is a balanced binary search tree of size 2^{2^i}, it can be searched in time $O\left(\log 2^{2^i}\right)$. We have

$$O\left(\log 2^{2^i}\right) = O\left(\log\left((2^{2^{i-1}})^2\right)\right) \leq O(\log t(x)^2) = O(\log t(x))$$

Now, at this point, a shift must also be performed, but this can be accomplished in the same time bound. Thus, the search time required by the working set structure satisfies the working set property.

2.2 Implicit Data Structures

We now turn our attention to data structures with very low space overhead. Such structures will be used as "black boxes" for the results in Chapter 3. We begin by defining more precisely the notion of low space overhead, and discuss related issues. We then briefly survey the relevant literature.

2.2.1 Definition and Issues

We begin by defining the notion of an *implicit data structure.*

Definition 2.2 (Implicit data structure)**.** A data structure is *implicit* if it uses only the space required to store its elements and $O(1)$ additional *words* of memory, where a word consists of $O(\log n)$ bits. In particular, the data structure knows the value n.

An implicit data structure can therefore be viewed as an array where each element of the array corresponds to an element contained in the data structure. In addition to the array, a constant number of words may also be maintained, and the number of elements stored is also known.[4]

One obvious example of an implicit data structure is simply an array. Such a data structure can be used to solve the dictionary problem. If the array is unsorted, searches and deletions take $O(n)$ time and insertions take $O(1)$ time. This data structure uses only n storage (as opposed to the $O(n)$ used by, *e.g.*, an AVL tree), but this comes at the cost of a high query time for searching.

The important observation here is that since we do not allow the use of a linear number of pointers (as in binary search trees) to encode structural information about our data structure, we must instead rely on other means such as permuting the order of the elements in the data structure to *implicitly* encode such structural information.

To encode structural information into the simple array implementation of a dictionary mentioned above, we could insist that the list be maintained in sorted order. This reduces searching time to $O(\log n)$ but increases insertion and deletion costs to $\Omega(n)$.

Encoding structural information in a permutation can be done in several ways. One very common technique to simulate pointers is that of *bit stealing* (see, *e.g.*, [34]). Given $2k$ elements in an array, we can permute their sorted order slightly to encode k bits. Suppose the elements in sorted order are $a_1, a_2, \ldots, a_{2k}$, and consider pairs of the form a_{2i-1}, a_{2i} for $i = 1, 2, \ldots, k$. We can store a_{2i-1}, a_{2i} in that order to denote a 0 bit, or swap them and store a_{2i}, a_{2i-1} to denote a 1 bit. An array permuted

[4]This definition corresponds to that of Franceschini and Munro [18]. The definition of Munro and Suwanda [35] allows only the value of n to be stored.

in this manner still supports a slightly modified binary search that checks elements adjacent to those normally checked; this search is asymptotically equivalent to the usual search. The bits gained from this technique can be used, for instance, to encode a pointer or a counter of some kind.

2.2.2 Examples of Implicit Data Structures

A classic example of an implicit data structure is a heap [42]. A heap is a priority queue that supports the standard dictionary operations in addition to a MINIMUM operation that quickly reports the minimum element in the heap and a DELETE-MINIMUM operation which deletes this minimum element. A heap can be visualized as a tree that maintains a heap order on the nodes: the smallest node always appears at the root of the tree, and we maintain the invariant that a parent is always less than its children. To get an implicit data structure, observe that we can place this whole structure in an array $A[1, \ldots, n]$ by putting the root at $A[1]$ and the children of $A[i]$ in $A[2i]$ and $A[2i+1]$. By carefully implementing the priority queue operations, this leads to $O(1)$ time for finding the minimum, $O(\log n)$ time for insertion and $O(\log n)$ time for the deletion of the minimum element. Unfortunately, searching for an arbitrary element now takes $O(n)$ time.

We now turn our attention to the dictionary problem. This problem was first addressed in the setting of implicit data structures in 1980 by Munro and Suwanda [35]. They showed how to construct a dictionary supporting all operations in $O(\sqrt{n})$ time. Furthermore, it was shown that when elements are kept in some fixed partial order based on their values, the product of search time and update (*i.e.*, insertion or deletion) time must be $\Omega(n)$. By changing the type of order used, query times were improved to $O(\log n)$ for search and $O(\sqrt{n}\log n)$ for insertions and deletions, or $O\left(n^{1/3}\log n\right)$ for each operation. This was later improved to $O(\log n)$ search time and $O\left(n^{\sqrt{2/\log n}}(\log n)^{3/2}\right)$ update time [20].

At this point, the main open problem was to find an implicit dictionary with polylogarithmic[5] query times for both search and update. The first such dictionary was

[5]A function $f(n)$ is polylogarithmic if $f(n) \in O\left(\log^{O(1)} n\right)$.

described by Munro [34] and had search, insertion and deletion costs of $O(\log^2 n)$. This result from 1986 went unimproved until 2002 when an implicit dictionary supporting all operations in $O(\log^2 n/\log\log n)$ was described by Franceschini et al. [19]. This was subsequently improved to $O(\log n \log\log n)$ time, although for updates this bound is amortized [16]. The same authors improved this to $O(\log n)$ time, although again with only an amortized bound for update costs [15], which was finally improved to $O(\log n)$ worst-case time for each operation by the same authors [17].

Since then, the analyses of such dictionaries have been refined to consider moves and comparisons during operations [18]. Such results will not be needed in this thesis.

It is interesting to note that there are results for implicit (and nearly implicit) data structures that are analogous to the optimum binary search trees of Knuth [29]. Such structures were presented by Frederickson [21] and were, like optimum binary search trees, static and required *a priori* assigned weights (*i.e.*, access probabilities).

2.3 Range Searching

In this section, we address the problem of *range searching*. For the range searching problem, we are given a set P of n points and preprocess it to quickly answer queries of the form, "report all points of P inside a query range R." The query range R may take many forms: R could be a halfspace, a simplex, a ball, or an axis-parallel box, for instance. We concentrate on the latter. In this case, the problem is known as *orthogonal range searching*.

In our setting, we consider *range reporting* queries, where all points of P that are inside of the query range are to be reported. Another type of query is called a *range counting* query, where only the number of points of P that are inside of the query range is required. Both of these are special cases of *semigroup queries*. Given a commutative semigroup[6] $(S, \circ)$, we assign each point of P a value from S and if the points in the query region are $p_1, \ldots, p_k$, we report $p_1 \circ p_2 \circ \cdots \circ p_k$. If we take S to

[6]Recall that a semigroup is a set S equipped with an associative binary operation $\circ$ such that S is closed under $\circ$. A commutative semigroup is a semigroup with the additional property that $a \circ b = b \circ a$ for all $a, b \in S$.

be the set $\{1, 2, \ldots, n\}$ under the usual addition operation and map each point of P to 1, we get the usual range counting queries. Similarly, if we take S to be the set of all possible subsets of P (*i.e.*, the powerset of P) under the union operation and map each point $p \in P$ to $\{p\}$, we get the usual range reporting queries.

2.3.1 Data Structures for Range Searching

There are two relatively simple data structures to solve the range searching problem: the kd-tree and the range tree. We discuss both of these data structures here.

The kd-tree. The kd-tree [4] works by recursively dividing the point set by the median points with respect to successive axes. In the plane, for instance, the point set is first partitioned around the median x-coordinate, and then the median y-coordinate, and then the median x-coordinate again, and so on. This process results in a binary tree and the nodes of this binary tree represent regions created by the partitioning halfspaces. Range searches can then be performed by testing for intersection with regions during a search in the tree. A kd-tree takes $O(n \log n)$ time to construct and uses $O(n)$ space. A range query in d dimensions takes time $O\big(n^{1-1/d} + k\big)$, where k is the number of points inside the query range. The advantage of kd-trees is that they are relatively simple to implement and only use $O(n)$ space. Unfortunately, their query time is not good, especially for large d. As d grows larger, $1 - 1/d$ gets closer to 1, and so the query time is nearly linear. Of course, if we are willing to spend linear time, we could simply perform a linear search of the point set to answer any query.

Range trees. Range trees [5, 32, 6] take a more hierarchical approach. For example, assume the point set is from the plane for the time being. The primary tree is a binary search tree with the points at the leaves, sorted by x-coordinate. At each internal node of the primary tree, a secondary tree is then constructed in the same manner, except that only the points stored in the subtree rooted at the primary tree node are stored in the secondary tree, and the secondary tree has its leaves sorted by y-coordinate. If the points are in the plane, we are done. If the dimension is higher, this process is repeated in each secondary tree to create tertiary trees, and so on. To see

how a search is performed, assume again that the point set is in the plane, and so our query is an axis-parallel rectangle, say $[x_1, x_2] \times [y_1, y_2]$ with $x_1 \leq x_2$ and $y_1 \leq y_2$. We search in the primary tree for x_1 and x_2 using the usual method for searching a binary tree, and find the node v where the search paths diverge. We then search all of the secondary trees to the right of the path from v to x_1 and all of the secondary trees to the left of the path from v to x_2. In higher dimensions, this process continues in the secondary trees to find tertiary trees, and so on. Range trees use $O\big(n \log^{d-1} n\big)$ space in d dimensions, and can answer range reporting queries in time $O\big(\log^d n + k\big)$, where k is the number of points in the query range. Through an application of *fractional cascading* [8, 9], the query time can be reduced to $O\big(\log^{d-1} n + k\big)$. In the plane, this corresponds to a $O(\log n + k)$ query time.

2.3.2 Range Searching in One Dimension

The results in this thesis are restricted to the one-dimensional problem. In this setting, it is relatively easy to get $O(\log n + k)$ query time and $O(n)$ space: simply keep the point set stored in a sorted array. Given a query, perform a binary search in the sorted array for the endpoints of the query box[7] This can be achieved in $O(\log n)$ time. Once this is done, simply walk from one endpoint to the other and report each point inside of the query range in $O(k)$ time, for a total query time of $O(\log n + k)$, where k is the size of the output.

This solution can be modified to fit the same scheme outlined above for range trees. The points are stored as leaves in a balanced binary tree, and the node where the search paths for the endpoints of the query interval diverge is found. The search continues as in range trees, but instead of checking secondary trees, the leaves below each node are checked instead. This leads to query times of $O(\log n + k)$ with a total space cost of $O(n)$, as well.

[7] In one dimension, the query box is an interval on the real line.

2.4 Other Results

In this section, we discuss three other results on data structures that we will need for this thesis: treaps, bounded-universe dictionaries, and doubling search.

2.4.1 Treaps

Treaps [36, 40] are a combination of binary search trees and heaps; hence their name. Treaps randomly assign priorities to their nodes and maintain both a heap-order on the priorities of keys and a binary search tree order on the values of keys. This serves to support the usual searching algorithm in a binary search tree, but also maintains some sense of balance in the treap.

Due to the random nature of treaps, the running times for operations upon them are given in an expected[8], rather than worst-case, sense. Despite their simplicity, treaps boast impressive query times. We require two results by Seidel and Aragon [36] later in this thesis. First, the dictionary operations take expected $O(\log n)$ time on a treap. Second, treaps support fast insertion and deletion when supplied with certain pointers into the treap. For an insertion, if a pointer is supplied to the predecessor or successor of the element to be inserted, insertion takes expected $O(1)$ time. Similarly, deleting an element takes expected $O(1)$ time when supplied with a pointer to the element to be deleted.

Treaps have many other interesting properties, of course, but since such properties will not be used in this thesis, we direct the interested reader to the paper by Seidel and Aragon [36].

2.4.2 Bounded-Universe Dictionaries

Given a bounded universe of possible keys in a dictionary, we can construct a data structure to support query times as a function of the size of the universe rather than the number of keys currently in the dictionary. Of course, depending on the size of

[8]See Section 2.5.1 for an explanation of expected query times.

the universe and the number of keys, this may or may not be beneficial. One such example is the van Emde Boas (vEB) tree [39].

Given a universe of size N, the vEB tree supports, among other operations, the dictionary operations in $O(\log \log N)$ time. In particular, the vEB tree supports predecessor searches in the same time bound. Unfortunately, a major limitation of the vEB tree is that the space it uses is linear in N rather than the number of keys stored inside the tree.

Suppose we have a universe of size N and n keys to store. Ideally, we would like to maintain the query times of vEB trees while improving the space from $O(N)$ to $O(n)$. This can be accomplished using a data structure due to Mehlhorn and Näher [33], although the query times now hold only in the expected sense due to an application of hashing.

There are other such bounded-universe data structures that offer different properties; since they will not be used in this thesis, we do not discuss them here.

2.4.3 Doubling Search

A *doubling search* [7] is a type of search performed in a sorted array. The search is performed starting from some index i. This index is successively doubled until the element being searched for is less than the element stored at that index. At this point, the usual binary search technique can be applied between the first index and the last index searched.

More formally, suppose we have a sorted array $A[1, 2, \ldots, n]$. Suppose we begin searching for x at index i. We assume that $x \geq A[i]$. The doubling search checks $A[i], A[i+1], A[i+2], A[i+4], A[i+8], \ldots, A[i+k]$, where $A[i+k] \geq x$. At this point, a binary search is performed in $A[i \ldots i+k]$. Due to the doubling nature of the search, the first phase requires $O(\log k)$ time. Because the second phase is a binary search amongst $k+1$ elements, it requires $O(\log k)$ time as well.

In a regular array, this technique is not better than the usual binary search, since the only index we can safely start at is $i = 1$. If we chose $i > 1$, then it might be the case that $A[1] = x$ and we will be unable to find it. However, if we do have additional

information that tells us a better index to start at, the time required for the search is proportional to the distance from the starting index. This property will be useful in Chapter 4.

2.5 Expectation and Amortization

In this section, we briefly review the notions of expected running time and amortized running time.

2.5.1 Expected Running Time

Throughout this thesis, we make reference to the notion of an *expected running time.* When an algorithm makes use of randomization, it may be the case that the running time of the algorithm may vary as a function of the random choices made by the algorithm. Therefore, it no longer makes sense to talk about the running time of the algorithm, since this may vary even on the same inputs, depending on the random choices made. Instead, we analyze the expected running time of the algorithm.

We can think of the expected value as the sum, over all running times, of the probability of each running time multiplied by that running time. In other words, the expected running time is

$$\sum_{i=0}^{\infty} \Pr\{\text{running time is } i\} \times i$$

Throughout this thesis, we always qualify running times with the term "expected" if random choices are made during the course of an algorithm. Note that the expectation is over the random choices made by the algorithm and does not make any assumptions regarding the distribution of queries.

2.5.2 Amortized Running Time

Operations on data structures are typically analyzed in terms of their worst-case cost. Some operations, however, may be easier (in the sense that they require less time)

than others even if they are of the same type. For instance, if a data structure is in some specific configuration, the time required to insert an additional element might be very small, whereas in a slightly different configuration, the time required might be larger.

It is therefore useful to analyze the cost of a sequence of operations on a data structure and compute the total cost. If a sequence of m operations has cost $mf(n)$, then we say that the amortized cost of an operation is $mf(n)/m = f(n)$. For example, if a sequence of m operations cost a total of $O(m \log n)$, then the amortized cost of an operation is $O(\log n)$, even though a particular operation in the sequence may have cost much less or much greater than $O(\log n)$.

We occasionally abuse the notation throughout this thesis, as in Section 2.1 when discussing splay trees. We claimed that since splay trees have the working property, a search for x takes $O(\log t(x))$ amortized time. This notation, while useful, hides the so-called "startup" cost. Indeed, the working set property for splay trees only applies to sufficiently long access sequences. In particular, the length of the access sequence m must be $\Omega(n \log n)$. Put another way, the total cost of a sequence $q_1, q_2, \ldots, q_m$ of m searches on a splay tree is

$$O\left(n \log n + m + \sum_{i=1}^{m} \log t_i(q_i)\right)$$

Here, the $n \log n + m$ enforces the fact that the access sequence has size $\Omega(n \log n)$, and the summation of the working set numbers is what we would expect from the working set property. If $m \in \Omega(n \log n)$, then this is bounded by $O(m + \sum_{i=1}^{m} \log t_i(q_i))$, and so the amortized cost over these m operations is $O(\log t(x))$, as we would expect. Nevertheless, the simplified $O(\log t(x))$ notation is less cumbersome and we will use it throughout this thesis. Furthermore, some of the running times described in this thesis hold in the *amortized expected* sense; this means that the running times in the summation of actual costs hold in the expected sense.

We conclude our discussion of amortized running times with a brief description of how to prove them. In this thesis, we will make use of the *potential method.* To bound the amortized running time, we define a potential function Φ on the data structure.

Operations on the data structure will cause Φ to increase and decrease over time. The amortized cost of an operation is the actual cost of that operation, plus the change in Φ caused by that operation. If Φ is always non-negative, as we shall guarantee during our arguments, then the sum of the amortized costs is an upper bound for the total cost of the sequence, and so the amortized cost per operation is exactly as we have calculated. For a more detailed explanation and justification of the potential method, we refer the reader to the work of Tarjan [38].

Chapter 3

Dictionaries with Low Space Overhead

In this chapter, we consider the problem of constructing distribution-sensitive dictionaries with low space overhead. In doing so, we combine data structures from the literature on both distribution-sensitivity and on implicit data structures. Section 3.2 shows how to make the necessary modifications to the working set structure described in the previous chapter to reduce its space overhead, and Section 3.3 shows how to further augment the data structure to support predecessor queries. Section 3.4 summarizes our results.

3.1 Introduction

All of the distribution-sensitive structures mentioned in the previous chapter that boast optimal ($O(\log n)$) query times make extensive use of pointers since they use binary search trees at some level. Similarly, none of the implicit data structures mentioned in the previous chapter have any form of distribution-sensitivity that does not rely on *a priori* knowledge of the query distribution. Our goal here will be to remedy this situation by showing how to construct a data structure that has both the working set property and low space overhead.

The main idea is as follows. We will adapt the working set structure of Iacono [26]

described in Section 2.1 in two ways to reduce its space overhead and eliminate pointers. The first thing we will do is remove the queues from the working set structure. In order to do this, whenever we want to dequeue an element (*i.e.*, during a shift) from a queue, we will instead select an element at random from the tree. We will show that this strategy also maintains distribution-sensitivity in the expected sense.

The next modification is to shrink the trees that comprise the working set structure. Observe that the only operations we require the trees to support are insertion, deletion and searching. Thus, any dictionary data structure suffices. To reduce space overhead, we use an implicit dictionary.

3.2 Modifying the Working Set Structure

In this section, we show how to make our modifications to the working set structure. First, we explain how to remove the queues through the use of randomization. This alone saves a linear amount of storage over the working set structure of Iacono [26]. Next, we show how to use implicit dictionaries instead of binary search trees to further reduce storage requirements.

3.2.1 Removing the Queues

First, we show that selecting a random element during a shift as opposed to maintaining a queue allows us to maintain an expected query time of $O(\log t(x))$.

Lemma 3.1. *The expected search cost in the randomized working set structure is* $O(\log t(x))$.

Proof. Fix an element x and let $t = t(x)$ denote the number of distinct accesses since x was last accessed. Suppose that x is in T_i. Since T_i has size 2^{2^i}, the probability that x is not randomly selected to be removed from T_i since the last time it was accessed is at least

$$\Pr\{x \text{ not deleted from } T_i \text{ after } t \text{ accesses}\} \geq \left(1 - \frac{1}{2^{2^i}}\right)^t = \left(1 - \frac{1}{2^{2^i}}\right)^{\frac{t2^{2^i}}{2^{2^i}}} \geq \left(\frac{1}{4}\right)^{\frac{t}{2^{2^i}}}$$

Now, if $x \in T_i$, it must have been selected for removal from T_{i-1} at some point. The probability that x is in at least the i-th tree is therefore at most

$$\Pr\{x \text{ is in at least the } i\text{-th tree}\} \leq \left(1 - \left(\frac{1}{4}\right)^{\frac{t}{2^{2^i-1}}}\right)$$

An upper bound on the expectation $E[S]$ of the search cost S is therefore

$$\begin{aligned}
E[S] &= \sum_{i=1}^{k} \log(|T_i|) \times \Pr\{x \in T_i\} \\
&\leq \sum_{i=1}^{k} \log(|T_i|) \times \Pr\{x \text{ is in at least the } i\text{-th tree}\} \\
&\leq \sum_{i=1}^{k} \log\left(2^{2^i}\right)\left(1 - \left(\frac{1}{4}\right)^{\frac{t}{2^{2^i-1}}}\right) \\
&= \sum_{i=1}^{k} 2^i \left(1 - \left(\frac{1}{4}\right)^{\frac{t}{2^{2^i-1}}}\right) \\
&= \sum_{i=1}^{\lfloor \log\log t \rfloor} 2^i \left(1 - \left(\frac{1}{4}\right)^{\frac{t}{2^{2^i-1}}}\right) + \sum_{i=1+\lfloor \log\log t \rfloor}^{k} 2^i \left(1 - \left(\frac{1}{4}\right)^{\frac{t}{2^{2^i-1}}}\right) \\
&= O(\log t) + \sum_{i=1}^{k-\lfloor \log\log t \rfloor} 2^{i+\lfloor \log\log t \rfloor} \left(1 - \left(\frac{1}{4}\right)^{\frac{t}{2^{2(i+\lfloor \log\log t \rfloor)-1}}}\right) \\
&= O(\log t) + O(\log t) \sum_{i=1}^{k-\lfloor \log\log t \rfloor} 2^i \left(1 - \left(\frac{1}{4}\right)^{\frac{t}{2^{2(i+\lfloor \log\log t \rfloor)-1}}}\right) \\
&\leq O(\log t) + O(\log t) \sum_{i=1}^{k-\lfloor \log\log t \rfloor} 2^i \left(1 - \left(\frac{1}{4}\right)^{\frac{t}{2^{2^i \log t}}}\right) \\
&= O(\log t) + O(\log t) \sum_{i=1}^{k-\lfloor \log\log t \rfloor} 2^i \left(1 - \left(\frac{1}{4}\right)^{\frac{1}{t^{2^i-1}}}\right)
\end{aligned}$$

It thus suffices to show that the remaining sum is $O(1)$. We will assume that $t \geq 2$, since otherwise x can be in at most the second tree and thus be found in $O(1)$ time. Considering only the remaining sum, we have

$$\begin{aligned}
& \sum_{i=1}^{k-\lfloor \log\log t \rfloor} 2^i \left(1 - \left(\frac{1}{4}\right)^{\frac{1}{t^{2^i-1}}}\right) \\
\leq & \sum_{i=1}^{k-\lfloor \log\log t \rfloor} 2^i \left(1 - \left(\frac{1}{4}\right)^{\frac{1}{2^{2^i-1}}}\right) \\
\leq & \sum_{i=1}^{k-\lfloor \log\log t \rfloor} 2^i \left(1 - \left(\frac{1}{16}\right)^{\frac{1}{2^{2^i}}}\right) \\
\leq & \sum_{i=1}^{\infty} 2^i \left(1 - \left(\frac{1}{16}\right)^{\frac{1}{2^{2^i}}}\right)
\end{aligned}$$

All that remains to show is that this infinite sum is bounded by a decreasing geometric series (and is therefore constant). The ratio of consecutive terms is

$$\lim_{i\to\infty} \frac{2^{i+1}\left(1 - \left(\frac{1}{16}\right)^{\frac{1}{2^{2^{i+1}}}}\right)}{2^i\left(1 - \left(\frac{1}{16}\right)^{\frac{1}{2^{2^i}}}\right)} = 2\lim_{i\to\infty} \frac{\left(1 - \left(\frac{1}{16}\right)^{\frac{1}{2^{2^{i+1}}}}\right)}{\left(1 - \left(\frac{1}{16}\right)^{\frac{1}{2^{2^i}}}\right)}$$

If we substitute $u = \frac{1}{2^{2^i}}$, we find that $\frac{1}{2^{2^{i+1}}} = u^2$. Observe that as $i \to \infty$, we have $u \to 0$. Thus

$$2\lim_{i\to\infty} \frac{\left(1 - \left(\frac{1}{16}\right)^{\frac{1}{2^{2^{i+1}}}}\right)}{\left(1 - \left(\frac{1}{16}\right)^{\frac{1}{2^{2^i}}}\right)} = 2\lim_{u\to 0} \frac{\left(1 - \left(\frac{1}{16}\right)^{u^2}\right)}{\left(1 - \left(\frac{1}{16}\right)^{u}\right)} = 2\lim_{u\to 0} \frac{8\left(\frac{1}{16}\right)^{u^2} u \ln 2}{4\left(\frac{1}{16}\right)^{u} \ln 2} = 0$$

Here, the second-to-last equality follows from an application of l'Hôpital's Rule.

Therefore, the ratio of consecutive terms is $o(1)$ and the series is therefore bounded by a decreasing geometric series. An expected search time of $O(\log t(x))$ follows. □

At this point, we have seen how to eliminate the queues from the structure at a cost of an *expected* search cost. In the next section, we will show how to further reduce space overhead by shrinking the size of the trees.

3.2.2 Shrinking the Trees

Our next task is to show how to replace the binary search trees in the working set structure with implicit dictionaries that support the same operation.

The query times for the implicit dictionaries, which we call *substructures*, will determine the query times for our data structure as a whole. It is straightforward to extend the analysis of Iacono [26] to the case of dictionaries with polylogarithmic access costs. In particular, if insertion, deletion and search costs are bounded by polylogarithmic functions $I(n)$, $D(n)$ and $S(n)$ respectively, then shifting takes $O(I(n) + D(n))$ time, since a shift consists of insertions and deletions. Therefore, the query time for insertions, deletions and expected searches in our structure are bounded by $I(n)$, $D(n)$, and $S(t(x)) + I(t(x)) + D(t(x))$ respectively.

Therefore, we could use the implicit dictionary of Munro [34] to get insertion and deletion costs of $O(\log^2 n)$ with expected search cost $O(\log^2 t(x))$. We could also use the results of Franceschini and Grossi [16] similarly, or even the same authors' subsequent results [17] to produce a data structure with expected search cost $O(\log t(x))$.

One final technicality is that of Definition 2.2. If each substructure is allowed $O(1)$ additional words of memory, then since there are $k = O(\log \log n)$ such structures, a total overhead of $O(\log \log n)$ may be present. However, if the structure is implicit in the sense that no additional memory is used (as per the definition of Munro and Suwanda [35]), then the data structure need only have access to the number of elements that are stored in it. Since this information can be calculated by examining the index of the dictionary, only a constant amount of overhead is needed. In general, however, we assume Definition 2.2 to get

Theorem 3.2. *Suppose we have a dictionary with polylogarithmic access costs $I(n)$, $D(n)$ and $S(n)$ for insertion, deletion and search, respectively. Then there exists a data structure with access costs $I(n)$, $D(n)$ and $S(t(x)) + I(t(x)) + D(t(x))$ for insertion, deletion and expected search, respectively. If the space for the dictionary substructures is n plus an additional $s(n)$ words, the space for our structure is n plus an additional $s(n) \log \log n$ words.*

An application using worst-case optimal implicit dictionaries [17] thus proves

Corollary 3.3. *There exists a dictionary that stores only the data required for its elements in addition to $O(\log \log n)$ words. This dictionary supports insertions and deletions in worst-case $O(\log n)$ time and searches in expected $O(\log t(x))$ time, where $t(x)$ is the working set number of the query x.*

3.3 Supporting Predecessor Searches

In this section, we describe a simple modification to the data structure that makes searches more useful. This improvement comes at the cost of additional space overhead.

Until now, we have implicitly assumed that searches in our data structure are successful. If they are not, then we will end up searching in each substructure at a total cost of $O(\log n)$ and returning nothing. Unfortunately, this is not very useful. Typically, a dictionary will return the largest element in the dictionary that is smaller than the element searched for or the smallest element larger than the element searched for. This feature is simple to implement in, *e.g.*, AVL trees since we can simply examine where we fell off the tree. This trick will not work in our data structure, however, since we have many such substructures and we will have to know when to stop.

Our goal is thus the following. Given a search key x, we would (as before) like to return x in time $O(\log t(x))$ if x is in the data structure. If x is not in the data structure, we would like to return $pred(x)$ in time $O(\log t(pred(x)))$, where $pred(x)$ denotes the predecessor of x. The discussion below assumes that we are working with substructures that have access costs $O(\log n)$.

3.3.1 The Naïve Approach

To accomplish our goal of predecessor searching, we will augment our data structure with some pointers. In particular, every item in the data structure will have a pointer to its successor in the structure as a whole. During an insertion, each substructure will be searched for the inserted element for a total cost of $O(\log n)$ and the smallest

successor in each substructure will be recorded. The smallest such successor is clearly the new element's successor in the whole structure. Therefore, the cost of insertion remains $O(\log n)$. Similarly, during a deletion only the predecessor of the element being deleted will need to have its successor updated to be the successor of the deleted element. The predecessor of the deleted element can be found by searching each structure and thus the total cost of deletion remains $O(\log n)$.

During a search for x, we proceed as before. Consider searching in any particular substructure i. If the result of the search in substructure i is in fact x, then the analysis is exactly the same as before and we can return x in time $O(\log t(x))$. Otherwise, x is not in substructure i. In this case, we search substructure i for the predecessor (in that substructure) of x,[1] denoted by $pred_i(x)$. Since every element knows its successor in the structure as a whole, we can determine $succ(pred_i(x))$. If $succ(pred_i(x)) = x$, then we know that x is indeed in the structure and thus the query can be completed as usual.[2] If $succ(pred_i(x)) < x$, then we know that there is still an element smaller than x but larger than what we have seen, and so we continue searching for x. Finally, if $succ(pred_i(x)) > x$, then we have reached the largest element less than or equal to x, and so our search stops.

In any case, we shift the element we returned to first substructure as usual. The analysis of the time complexity of this search algorithm is exactly the same as before; we are essentially changing the element we are searching for during the search. The search time for the substructure we stop in dominates the cost of the search and since we return x if we found it or $pred(x)$ otherwise, the expected search time is $O(\log t(x))$ if x is in the dictionary and $O(\log t(pred(x)))$ otherwise.

3.3.2 A Tradeoff

Of course, the augmentation described incurs some additional space overhead. In particular, we now use n pointers for space overhead of $O(n)$. While the queues are

[1] Here we are assuming that substructures support predecessor queries in the same time as searching.

[2] We have found x, so we might be tempted to stop in order to save time. This does not alter the asymptotic running time, however, because we still need to perform a shift from the tree that contains x.

now gone, this is essentially no more space-efficient than the original working set structure. To fix this, observe that we can leave out the pointers for the last few trees and simply do a brute-force search at the cost of slightly higher time complexity. In fact, since pointers are being stored in the last few trees anyway, we do not bother with implicit dictionaries and simply use balanced binary search trees for them.

Suppose we leave the pointers out of the last j trees: T_{k-j+1} to T_k, where k represents the index of the last tree, as before. Now, suppose we are searching in the data structure and we get to T_{k-j+1}. At this point, we may have to search in the rest of the structure, since if we don't find the query, we may not know whether or not we have found its predecessor until we reach the end of the data structure. Our main tool is

Lemma 3.4. *Let $0 \leq j \leq k$. A predecessor search for x in the data structure takes expected time $2^j \log t(x)$ if x is in the data structure, and $2^j \log t(pred(x))$ otherwise.*

Proof. Assume the search reaches T_{k-j+1}, since otherwise our previous analyses apply. If this is the case, $t(x) \geq 2^{2^{k-j}}$, since at least $2^{2^{k-j}}$ operations have taken place.[3] Due to the doubly exponential growth of the trees, the search time is bounded by the search time in T_k. Since T_k has size at most 2^{2^k}, the expected search time is bounded by

$$O\left(\log 2^{2^k}\right) = O\left(\log 2^{2^{k-j}2^j}\right) = O\left(\log \left(2^{2^{k-j}}\right)^{2^j}\right) = O\left(2^j \log 2^{2^{k-j}}\right) \leq O\left(2^j \log t(x)\right)$$

This analysis of course applies as well when x is not in the dictionary, in which case we get a bound of $O(2^j \log t(pred(x)))$. □

Note that if the other substructures have search cost $S(n)$ that is bigger than $O(\log n)$ (but still polylogarithmic), the search time is still bounded by $2^j \times S(t(x))$ (or $2^j \times S(t(pred(x)))$). The result for polylogarithmic search costs follows from the same generalization as that in Section 3.2.

[3]This follows from the proof of Lemma 3.1, which is why the result here holds in the expected sense.

It remains to consider the space overhead in this case. Since each tree has size at most the square of the previous tree, by leaving out the j largest trees, the total space overhead used by the pointers is $O\left(n^{1/2^j}\right)$. The main result is thus

Theorem 3.5. *Given a dictionary produced by Theorem 3.2 with access costs $I(n)$, $D(n)$ and $S(t(x)) + I(t(x)) + D(t(x))$ for insertion, deletion and expected search, respectively, there exists a dictionary that supports these same operations in addition to predecessor searches. Let $0 \leq j \leq k$. During a predecessor search for x, if x is in the dictionary, it is returned in expected time $2^j \times S(t(x))$. If x is not in the dictionary, its predecessor $pred(x)$ is returned in expected time $2^j \times (S(t(pred(x))) + I(t(pred(x))) + D(t(pred(x))))$. The space required for this dictionary is $n + (k-j) \times s(n) + O\left(n^{1/2^j}\right)$.*

While the dependence on j is exponential, recall that k (and thus j) is $O(\log \log n)$ and thus this term is never too big. In particular, by picking $j = 1$ and using the data structure of Corollary 3.3, we get

Corollary 3.6. *There exists a dictionary that stores only the data required for its elements in addition to $O(\sqrt{n})$ words. This dictionary supports insertions and deletions in worst-case $O(\log n)$ time and searches in expected $O(\log t(x))$ time, where $t(x)$ is the working set number of the query x. If x is not in the dictionary, its predecessor $pred(x)$ is returned in expected time $O(\log t(pred(x)))$.*

One last consideration is that when an element is found in the last j trees, we will need to find its successor so that it is known to that element once it is back inside the first tree. However, this is easily accomplished since all of the elements in the data structure have been examined at this point, and so we can make an additional pass to find the successor.

3.4 Conclusion

We have seen how to modify the working set structure of Iacono [26] in several ways. To become more space efficient, we can remove the queues and use randomization to shift elements, while replacing the underlying binary search trees with implicit

dictionaries. To support more useful search queries, we can sacrifice some space overhead to maintain information about some portion of the elements in the dictionary in order to support returning the predecessor of any otherwise unsuccessful search queries. All such modifications maintain some notion of distribution sensitivity.

Chapter 4

Range Searching in One Dimension

In this chapter, we consider the problem of constructing a distribution-sensitive data structure for the range searching problem in one dimension. In order to reach this goal, we must first define what is meant by "distribution-sensitive," since there is no clear generalization of the working set property to intervals. Section 4.2 shows how to generalize the working set property to this setting, and Section 4.3 shows how our data structure operates. We also discuss how to support the range counting case in Section 4.4. Section 4.5 summarizes our results.

4.1 Introduction

None of the data structures mentioned in Chapter 2 that solve the range searching problem have the working set property. In fact, none of them are distribution-sensitive even if the distribution of the queries is known in advance. Our goal in this chapter is to generalize the notion of the working set number to intervals and then provide a data structure that can achieve this analogue of the working set property.

We begin by giving a modified version of the working set number that takes into account both the number of queries since the last access, as well as the size of the resulting interval at that point. A data structure similar to the working set structure [26] is then used to store the resulting intervals. In order to guarantee the working set property, the intervals need to be carefully managed.

We begin with a simple example to show that it is not difficult to get *some* notion of distribution-sensitivity in this setting.

4.1.1 The Dynamic Finger Property

The dynamic finger property is a distribution-sensitive property that states that the time required for a search is logarithmic in the rank distance[1] between the current search and the previous search. To get this property for regular dictionary searching is not too difficult: we can store the elements in sorted order in an array, and instead of starting a search from the end of the array, simply start a doubling search [7] from the index the last search ended at.

In our case, searches are intervals on the real line. However, we can use a similar approach to get the dynamic finger property. As before, all of the elements are stored in sorted order in an array. If we define the distance between two intervals $[a, b]$ and $[c, d]$, where the endpoints represent indices in a sorted array, to be $\min\{|a-c|, |a-d|, |b-c|, |b-d|\}$ (*i.e.*, the minimum rank distance between endpoints), the same doubling search technique works. If this approach is used, we need to remember both endpoints of the result after every search.

This technique leads to a query time of $O(\log d(I, I') + k)$, where I and I' are the results of the current and previous queries respectively and k is the number of points reported. Here, the distance function d is defined as above. This technique is also adaptable to range counting queries, where the $O(k)$ term is replaced with a $O(\log k)$ term. In this case, the indices of the endpoints of the query interval are found by performing binary searches after a point in the result interval has been located.

4.2 Generalizing the Working Set Property

In this section, we generalize the working set number to our setting. In particular, we define a new working set number and then discuss some of its implications.

[1]The *rank distance* between two keys is the number of keys occurring between them in sorted order.

4.2.1 Defining the Working Set Number

For the case of range searching in one dimension, we modify the working set number defined in Definition 2.1 slightly. Suppose that the query sequence consists of the intervals $q_1, q_2, \ldots, q_m$, and let $I_i = q_i \cap P$ so that I_i is the result of query i. Whereas before we defined $l_i(x) = \min\{\{\infty\} \cup \{j > 0 : q_{i-j} = x\}\}$, we now want to check for intersections rather than equality, so we let $l_i(x) = \min\{\{\infty\} \cup \{j > 0 : I_{i-j} \cap x \neq \emptyset\}\}$. With $l_i(x)$ properly defined, we might be tempted to keep the same definition of $t_i(x)$ as we had before.

Unfortunately, this alone is not sufficient. Imagine if the first query consisted of all points of P. If we used the above definition as the working set number, the working set property would require the next query to take constant time since every other interval intersects the first one. Clearly, this is not a reasonable definition, since there is a $\Omega(\log n)$ lower bound on the running time of this query.

Instead, we compute the number of queries since an interval intersected the current query as before and multiply this by the size of that interval. We have

Definition 4.1 (Working set number of an interval)**.** The working set number $t_i(x)$ of a query interval x at time i is defined as

$$t_i(x) = \begin{cases} n & \text{if } l_i(x) = \infty \\ |\{I_{i-l_i(x)+1}, \ldots, I_i\}| \times |I_{i-l_i(x)}| & \text{otherwise} \end{cases}$$

For ease of notation, let $t'_i(x) = |\{I_{i-l_i(x)+1}, \ldots, I_i\}|$ when $l_i(x) \neq \infty$, so that $t_i(x) = t'_i(x) \times |I_{i-l_i(x)}|$ if $l_i(x) \neq \infty$. As before, we use $t(x)$ to denote this modified version of the working set number when this query is the current query in a sequence, and we use $t'(x)$ in the same manner.

Figure 4.1 shows an example of how to compute the working set number of a query interval.

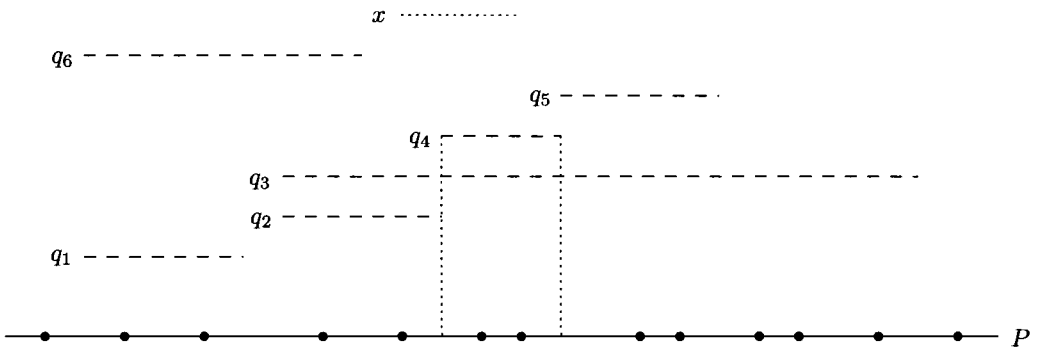

Figure 4.1: Determining the working set number of an interval x at time 7. The previous queries are the dashed lines. The smallest j such that I_{i-j} intersects x is $j = 3$, since x intersects I_4 but not I_5 or I_6. Thus, $l_7(x) = 3$ and so $t'_7(x) = |\{I_5, I_6, I_7\}| = 3$. Since $I_{7-l_7(x)} = I_4$ has size 2, the working set number $t_7(x)$ is $3 \times 2 = 6$.

4.2.2 Notes on the Working Set Number

Observe that in the example given above where the entire point set P is in the first query interval, the working set number is no longer a constant. Rather, since the result interval has size n, the working set number is n as well. Thus, our data structure will only need to answer this query in $O(\log n)$ time. This seems natural since no point to be reported is any more recent than any other.

One immediate question is how this definition of working set number compares to the usual definition. To answer this, consider the access sequence consisting of the queries $1, 2, \ldots, n, 1, n, 1, n, \ldots$ on a dictionary containing the n elements $1, \ldots, n$. If the access sequence has length $m = \Omega(n \log n)$, the first n accesses take $O(n \log n)$ time because none of the elements have been seen yet. However, after that, each access takes only constant time because the working set number is constant. The whole sequence thus has cost $O(n \log n + m)$, or $O(m)$ since $m \in \Omega(n \log n)$. Thus, the cost is linear in the size of the sequence.

In our setting, we might have the numbers $1, \ldots, n$ in our data structure and perform the analogous interval searches $[0.5, 1.5), [1.5, 2.5), \ldots, [n-0.5, n+0.5)$ followed by the searches $[0.5, 1.5), [n-0.5, n+0.5), [0.5, 1.5), [n-0.5, n+0.5), \ldots$ to create a sequence of length $m = \Omega(n \log n)$. As before, the first n searches take time $O(n \log n)$,

but after that the working set number is constant because the result interval has size 1 and the same interval is queried every other search. Thus, this sequence has total cost $O(m)$ as well. Therefore, on this type of singleton sequence, we do no worse than that achieved by dictionaries.

There are, unfortunately, two less-than-desirable properties of this definition. First, the definition allows us only to minimize $l_i(x)$, not $|I_{i-l_i(x)}|$. Therefore, recent-but-large intervals may make the working set number quite big. This leads us to the second problem, which is that if the whole range is searched for, the data structure is essentially "reset" and the next search will take $\Theta(\log n)$ time.

4.3 The Data Structure

In this section, we describe a data structure that can answer one-dimensional range reporting queries in time $O(\log t(x) + k) + o(\log n)$, where x is the query interval and $t(x)$ is defined as in Definition 4.1. Here, k represents the number of reported points.

4.3.1 Overview

Our data structure consists of three distinct parts that are interconnected. The first part is a sorted array $A[1 \ldots n]$ containing all of the points of P in sorted order. The second part consists of the working set structure of Iacono [26], which we denote by $W = \langle T_1, T_2, \ldots, T_k \rangle$.[2] Our working set structure W differs from that of Iacono in two ways. First, W stores intervals as opposed to points. Second, the number of intervals stored in T_i may be anywhere from 0 to 2^{2^i}.

The trees in W will be implemented as treaps [36]; we shall see why when discussing our modified shifting operation later. All of the elements of A that represent endpoints of an interval in the working set structure have pointers to their corresponding intervals.[3] Conversely, each interval in the working set structure maintains

[2]The queues are present in W, but they are managed as described by Iacono [26] and in Section 2.1 and thus we do not duplicate our explanation.

[3]In the case of intervals containing only one point, we could simply store a flag indicating that this interval is singleton.

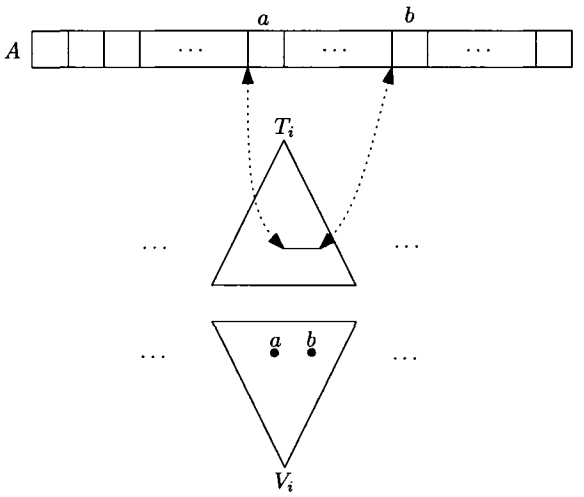

Figure 4.2: A schematic of the range reporting data structure. The interval with endpoints at indices a and b in A is stored in T_i and there are pointers between these array elements and the interval in the tree. The vEB tree V_i stores the indices of the endpoints.

a pointer to its endpoints in A. The final part of our data structure is a set of van Emde Boas (vEB) trees [39] $V_1, V_2, \ldots, V_k$. The universe of keys[4] for the vEB trees consists of the n points of P viewed as endpoints of intervals; this can be accomplished by using the n indices of A. We maintain the invariant that the elements of T_i are exactly the same as the elements of V_i for all i. In addition, the union of all intervals of W is exactly the set of points P and all of the intervals in W are disjoint.

Initially, W holds only one interval that stretches from the first point of P in sorted order to the last. This interval is stored in T_1. Therefore, by our construction above, this interval has pointers to the first and last elements of A. Similarly, the first and last elements of A have a pointer to the interval in W. Finally, the endpoints of this interval are the only elements in V_1.

Figure 4.2 presents a schematic of the data structure.

[4]Note that because of the way we are now using a bounded-universe data structure, we are restricting ourselves to the static version of the range searching problem; no points may be inserted or deleted.

4.3.2 Shifting

Before we discuss the search algorithm, we must first describe how the shift operation is performed in our data structure. Suppose we are shifting from T_i. If T_i has size less than 2^{2^i}, then nothing is done. Otherwise, after dequeuing an element from the queue corresponding to T_i, we delete it from T_i and V_i. We then insert it into the next tree (say, T_{i+1}) by first searching for its endpoint in V_{i+1} to produce its predecessor in V_{i+1}, and thus a pointer to its predecessor in T_{i+1}. Since T_{i+1} is a treap,[5] we can insert the element given a pointer to its predecessor in expected constant time. This process then repeats as in a usual shift operation. However, unlike the previous shift operation, our shift operation stops as soon as we insert an element into a tree T_j with $|T_j| < 2^{2^j}$. This stems from the fact that the T_i no longer have size exactly 2^{2^i}, but rather have size *at most* 2^{2^i}; we elaborate on this point below.

4.3.3 Searching

A search for a query interval x is performed in the following way. First, a search for x is performed in W. For our purposes, a search is considered to be successful if an interval in W intersects the query interval x, *i.e.*, whereas a normal search for x terminates when an item $y \in W$ is found such that $x = y$, we now terminate when an interval $y \in W$ is found such that $x \cap y \neq \emptyset$. Since the intervals in the working set structure are disjoint, the intervals form a total order.

If the search is not successful, we have searched all of W and report that no points were found. Assume the search is successful, and denote by I the first interval found in W such that $I \cap x \neq \emptyset$. By our construction, I has pointers into the array that indicate its endpoints. Suppose the endpoints of I are located at $A[i]$ and $A[j]$ with $i \leq j$. We now perform a doubling search [7] in $A[i \ldots j]$ until we find a point that is contained in the query interval x. Such a search is performed by checking the indices $i, i-1, i-2, i-4, \ldots$ and $j, j+1, j+2, j+4, \ldots$. To ensure that we do not go too far in one direction, we alternate by checking $i, j, i-1, j+1, i-2, j+2, \ldots$. At this

[5]The data structure of Levcopoulos and Overmars [31] also give us this property, even in the worst-case; however, since we will require a data structure that uses randomization later, we simply use treaps.

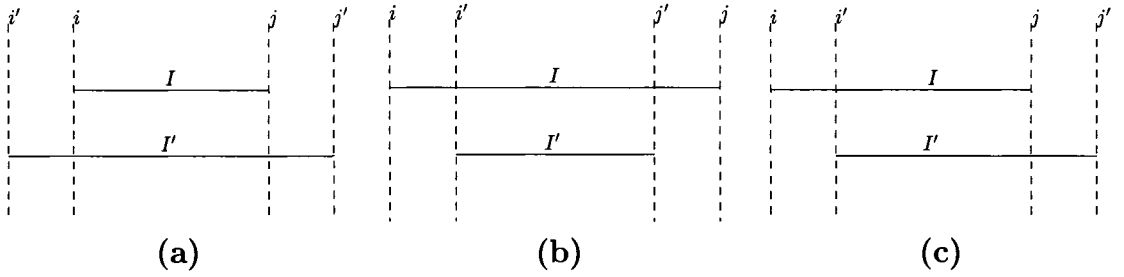

Figure 4.3: The intervals found (I) and reported (I') during a search. **(a)** I is contained inside I'. **(b)** I' is contained inside I. **(c)** I and I' straddle each other.

point, we can walk on both sides of this index and report all points in $P \cap x$. Suppose the reported points are $A[i' \ldots j']$ with $i' \leq j'$.

Once we have reported the points, we must reorganize W to maintain the interval working set property. To do this, consider the interval $I = A[i \ldots j]$ that we found and the interval $I' = A[i' \ldots j']$ that we reported. We would like to shift all of I' to T_1 in W, as we did in the working set structure of Iacono [26]. However, we must also maintain the invariant that all intervals stored in W are disjoint, and the union of all intervals contains P.

In order to guarantee these invariants, we create a new interval that stores only I'. We then consider the other intervals that are impacted by this. Our actions depend entirely on I and I'. There are three cases to consider: either I is contained in I', I' is contained in I, or I and I' straddle each other. The cases when I and I' share an endpoint are covered by the first two cases. Refer to Figure 4.3.

Case 1. Assume I is contained inside I'; see Figure 4.3(a). In this case, we delete I from the tree of W that it was found in and the corresponding vEB tree. At this point, we would like to shift the interval I', but we observe that intervals elsewhere in W intersect $I' \setminus I$. In order to remedy this, we first have to walk along A and delete[6] the intervals in $I' \setminus I$ by following the pointers from A to the corresponding trees in W. When an interval is deleted from a tree in W, we also make sure to delete

[6]By performing these deletions, some trees may store fewer than the usual number of elements.

it from the corresponding vEB tree. Such intervals may include those that straddle the endpoints of I', in which case these intervals will need to be trimmed; this can be accomplished by redefining their endpoints and making the corresponding changes in their vEB trees. At this point, I' is disjoint with all intervals in W and so we place it in T_1 and perform a shift.

Case 2. Assume I' is contained inside I; see Figure 4.3(b). In this case, we delete I from the tree of W that it was found in. This, however, results in all points in $I \setminus I'$ no longer being covered by intervals in W. To fix this, we create two new intervals $[i, i']$ and $[j', j]$ and place them in the same tree that I was found in. This may cause the tree to grow beyond its capacity, in which case we perform a shift. In any event, by the disjointness invariant, these intervals do not overlap any other intervals in W since they were part of I before. At this point, I' is disjoint with all intervals in W and every point in P is covered by some interval in W, so we place I' in T_1 and perform a shift. Unlike in the previous case, however, there is no longer room in the tree I was found to shift. In fact, we may need to shift to the end of W; this will be addressed in the analysis.

Case 3. Assume I and I' straddle each other; see Figure 4.3(c). As before, we delete I from the tree of W that it was found in and would like to shift I' to the beginning of W. This, however, results in two problems: first, $[i, i']$ will no longer be covered by intervals in W, and second, $[j, j']$ will intersect possibly several intervals in W. To remedy this situation, we combine the approaches used in the previous two cases. First, we delete all intervals in the range $[j, j']$ in the exact same way as Case 1. This restores the disjointness invariant. Next, we create a new interval $[i, i']$ in the exact same way as Case 2. This restores the invariant that all points are covered by an interval in W. As before, we can now place I' in T_1 and perform a shift. As in Case 2, there may no longer be room in the tree I was found to shift, but we will address this in the subsequent analysis.

4.3.4 Analysis

We are now ready to analyze the data structure. The search algorithm essentially proceeds in two phases: first, we find an interval in the working set structure that intersects the query interval, and then we update some intervals to maintain the data structure.

First, consider the work done searching in the working set structure. Assume an intersecting interval is found in the l-th tree. In the analysis of Iacono [26], we observed that to be in tree l, the element found must have been dequeued from the previous structure during a shift, at which point $2^{2^{l-1}}$ queries had taken place. However, this is not true in our structure, since we may shift more than one interval after a query. After a query, the interval that is reported is placed at the beginning of W, and the other intervals (of which there are at most two) remain in T_l. The (at most) two oldest intervals in T_l are then placed into T_{l+1}, and so on. Therefore, each query is responsible for moving at most three elements down the tree. Thus, we can guarantee that at least $2^{2^{l-1}}/3$ queries have occurred if our search ends in T_l. All such queries are disjoint with the current query interval, since otherwise we would have found an intersecting interval before tree l. Therefore, $t'(x) \geq 2^{2^{l-1}}/3$. Since the search cost is still dominated by that of the last structure, the cost is $O\left(\log 2^{2^l}\right) = O\left(\log 2^{2^{l-1}}\right) = O(\log t'(x))$.

Next, we consider the cost of moving around the sorted array and splitting intervals as necessary. Observe that once we have found the interval in the previous step, we perform a doubling search in time $O\left(\log |I_{i-l_i(x)}|\right)$ to find the interval to be reported, and report it in time $O(k)$, where k is the number of points to be reported. All k points are put into a new interval and the corresponding pointers are set. This can be achieved by walking through the sorted array in $O(k)$ time. Now, we consider the three cases in Figure 4.3. For now, we ignore the cost of shifting.

In the first case, I is contained inside I' as in Figure 4.3(a). To maintain the disjointness invariant, we must delete all of the intervals in the range $[i', i]$ and $[j, j']$. We can accomplish this by following the pointers from A to the corresponding tree of W and deleting the interval. Since the trees of W are treaps, the intervals can be deleted in expected $O(1)$ time because we are providing a pointer to the element

to be deleted. We still have to remove the extra intervals from the vEB trees, and for now we will take the straightforward approach of deleting each in turn at cost $O(k \log\log n)$; this will be improved using an amortization argument later.

In the second case, I' is contained inside I as in Figure 4.3(b). In order to ensure that all points are covered by intervals of W, we walk from either endpoint of I to the closest endpoint of I' and create a new interval which is then placed in T_l and V_l. Since we have already searched up to T_l, the insertion into T_l does not increase the asymptotic running time, while the insertion into V_l costs $O(\log\log n)$.

In the third case, I and I' straddle each other as in Figure 4.3(c). In order to ensure disjointness, we first delete intervals in the range $[j, j']$ in the exact same way as the first case at cost $O(k \log\log n)$. Next, we ensure that all points are covered by creating a new interval $[i, i']$ in the exact same way as the second case at cost $O(\log\log n)$. The total cost is thus $O(k \log\log n + \log\log n)$.

After splitting the intervals as necessary, we must actually report the result; this can clearly be accomplished in $O(k)$ time by walking along the array. Thus, at this point, the total time is

$$O\big(\log t'(x) + \log |I_{i-l_i(x)}| + (k+1)\log\log n + k\big)$$

Now, recall that $t(x) = t'_i(x) \times |I_{i-l_i(x)}|$, and so $\log t'(x) + \log |I_{i-l_i(x)}| = \log t(x)$. Thus, the total time is

$$O(\log t(x) + k \log\log n + k)$$

However, we now need to perform a shift from T_1 to T_l. Unfortunately, T_l may already have size 2^{2^l}, since we needed to add intervals in the second and third cases. In fact, we may need to shift all the way to the last tree in W before we can guarantee that each tree has the proper size.

Consider shifting from i to $i+1$; the other direction is symmetric. First, we delete the oldest element from T_i in $O(1)$ time, since we are given a pointer to it. Next, we delete the same element from V_i at cost $O(\log\log n)$. We then search for this element in V_{i+1} in $O(\log\log n)$ time to find (a pointer to) its predecessor. Given this predecessor, we can then insert into T_{i+1} in $O(1)$ time. Thus, to shift by a single

index costs $O(\log\log n)$ time in total, and so to shift all $O(\log\log n)$ indices takes time $O((\log\log n)^2)$. The total time for a query is therefore

$$O\big(\log t(x) + k\log\log n + (\log\log n)^2 + k\big)$$

Next, we consider the space required by this data structure. The working set structure W requires space $O(n)$, as does the array A. All that remains are the vEB trees. Each such tree is over a universe of size n and thus has size $O(n)$, and since there are $k = O(\log\log n)$ trees, the total space used by the vEB trees is $O(n\log\log n)$.

Putting these results together, we get

Theorem 4.2. *There exists a static data structure for one-dimensional range searching of size $O(n\log\log n)$ with expected search time*

$$O\big(\log t(x) + k\log\log n + (\log\log n)^2 + k\big)$$

Of course, this result falls a little short of our goal because of the dependencies on n. Our goal now will be to reduce this running time as much as possible. Another problem is the super-linear space; ideally, we would like to use only $O(n)$ space. This concern will also be addressed.

4.3.5 Improvements

We now discuss how to improve the result of Theorem 4.2.

First, we will improve the space requirements from $O(n\log\log n)$ to $O(n)$ by noting that if we replace vEB trees with the bounded dictionaries of Mehlhorn and Näher [33] as discussed in Section 2.4, the space required by each of $V_1, V_2, \ldots, V_k$ is proportional to the number of keys stored inside of them, rather than the size of the universe.[7] Thus, the space required for the i-th bounded dictionary V_i is $O\left(2^{2^i}\right)$, and the total space required is thus $O(n)$.

[7]Alternatively, we could keep the vEB trees and use the worst-case data structure of Levcopoulos and Overmars [31] instead of the treaps in W. This approach makes the data structure completely deterministic, but comes at the cost of using $O(n\log\log n)$ space.

We now provide a more careful analysis of the query time. We begin with a simple observation. The $O(k \log\log n)$ term in the running time comes from deleting the (at most) k intervals that are inside the reported interval I' but are not in the interval I found in W. Deleting each of these costs $O(\log\log n)$, and so deleting all of them costs $O(k \log\log n)$. However, by deleting these intervals, we have reduced the total number of intervals stored in the structure.

Define a potential function Φ to be the number of intervals stored in W times $c \log\log n$, that is, $\Phi(D) = c|W| \log\log n$ where D represents the data structure and c is some sufficiently large constant. Suppose we must delete $j < k$ intervals as described above. Then $\Phi(D) \geq cj \log\log n$ and the potential drops by $cj \log\log n$ when these j intervals are deleted. Furthermore, any search adds at most two intervals to W, and so the potential is increased by at most $2c \log\log n$. Thus, the amortized cost of performing these deletions is $O(\log\log n)$. Of course, we still may need to perform a large shift, so the $O((\log\log n)^2)$ term remains and the $O(\log\log n)$ is absorbed by it.

Because the potential assigned to the initial data structure is $O(\log\log n)$ since only one interval is present and since the potential is always non-negative, the total time required for the query sequence $q_1, q_2, \ldots, q_m$ is

$$O\left(\log\log n + \sum_{i=1}^{m} (\log t_i(q_i) + k_i) + m\,(\log\log n)^2\right)$$

Here, k_i denotes the number of points reported for query q_i. We thus have our main result

Theorem 4.3. *There exists a static data structure for one-dimensional range searching of size $O(n)$ with amortized expected search cost $O(\log t(x) + (\log\log n)^2 + k)$.*

4.3.6 Refining the Analysis

At this point, it would be nice to remove the additive $O((\log\log n)^2)$, but we now show that to do so would require a different approach than the one described here. Thus, the analysis presented here is tight.

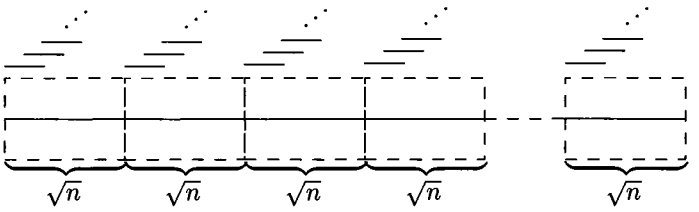

Figure 4.4: The analysis from Theorem 4.3 is tight. A bad access sequence for range searching queries consists of first filling the tree with $\sqrt{n}$ intervals of size $\sqrt{n}$ and then querying slightly overlapping intervals in sequence within a block of size $\sqrt{n}$.

Consider the following access sequence on our data structure that contains the points $P = \{1, 2, \ldots, n\}$. Initially, the first tree holds an interval containing all of the points. We begin by filling the entire working set structure in the following way. First, we query $[1, \sqrt{n})$, then $[\sqrt{n}+1, 2\sqrt{n})$ and so on until $[n-\sqrt{n}+1, n)$. This produces $\sqrt{n}$ intervals and therefore fills up all but the last tree of W.

After this initialization, we begin the access sequence illustrated in Figure 4.4. We query $[1,3), [2,4), [3,5), \ldots, [\sqrt{n}-2, \sqrt{n})$ in the first block, then $[\sqrt{n}, \sqrt{n}+2), \sqrt{n}+1, \sqrt{n}+3), \ldots, [2\sqrt{n}-2, 2\sqrt{n})$ in the second block, and so on. The first query in each block will find one of the $\sqrt{n}$-sized blocks from the first part of the access sequence, and the remaining queries within a block will find the previous query in the block in W and thus fall into the third case. In any event, we are guaranteed that the number of intervals in W will never decrease.

How long does this access sequence take? The initial $\sqrt{n}$ queries to fill the structure each take $\Theta(\sqrt{n})$ time, since each must report $\sqrt{n}$ points. For the remaining n queries, we examine the running times block-by-block. Since queries in different blocks do not intersect each other, the total running time is simply the sum of the running times for each block.

Within a block, the first query has working set number at least $\sqrt{n}$ since at least $\sqrt{n}$ queries have been made since the last access in this block. Since only a constant number of points are reported, the first query in each block thus takes $\Theta(\log n)$ time, ignoring the cost of shifting for now. Each of the remaining queries has constant working set number, since they each intersect the previous query and have constant size. However, since there are always at least $\sqrt{n}$ intervals in W, shifting the newly

created interval takes $\Theta((\log\log n)^2)$ time. Thus, the cost of each of these queries is $\Theta((\log\log n)^2)$.

Since there are $\sqrt{n}$ blocks, the total time for the access sequence within the blocks is

$$\Theta\left(\sqrt{n}\left(\log n + \left(\sqrt{n}-1\right)(\log\log n)^2\right)\right)$$

The initial $\sqrt{n}$ queries each take $\Theta(\sqrt{n})$ time, and so the total time for the entire access sequence is

$$\Theta\left(n + \sqrt{n}\left(\log n + \left(\sqrt{n}-1\right)(\log\log n)^2\right)\right)$$

Since there are a total of $\Theta(n)$ queries in the access sequence, the amortized time of an access is therefore

$$\Theta\left(1 + \left(1/\sqrt{n}\right)\log n + (\log\log n)^2 - \left(1/\sqrt{n}\right)(\log\log n)^2\right)$$

This expression is $\Theta((\log\log n)^2)$, and we therefore cannot hope to do any better with our data structure.

4.4 Supporting Counting Queries

The data structure described in Theorem 4.3 can also be used to support range counting queries in the same time bound: simply count the number of points reported. However, the $O(k)$ term in the query time is a rather high price to pay considering we are now only asking for a count of the elements in the query range, and not a list of the elements themselves. We now address how to modify the data structure to support range counting queries with a better time bound.

Previously, we used $O(k)$ time to walk along A and output points as well as delete and create intervals as required. To speed this process up, we will instead use an additional vEB tree V that contains all of the endpoints of intervals currently in W along with pointers to the relevant intervals in W. We maintain the same invariants as before, namely that all intervals in W are disjoint and cover all of the point set P.

Searches begin in the same way as before and find an interval $I \in W$ that intersects the query interval. Let I' be the interval to be reported. At this point, we consider the same three cases: I is contained inside I', I' is contained inside I, and I and I' straddle each other.

In the first case, we delete I from W and begin performing predecessor and successor searches in V to find the intervals in $I' \setminus I$. As before, there may be up to two intervals that contain the endpoints of I', in which case we can simply trim them. To find the endpoints of I', a doubling search [7] is performed from the endpoints of I. We then remove I from W, delete its endpoints from V and insert the endpoints of I' in their place. I' is then shifted in W as before.

In the second case, we perform a binary search in I to find the endpoints of I'. We then delete I from W and insert the endpoints of I' into V. The old endpoints of I in V must be updated to point to the (at most) two new intervals on either side of I' in I, which must then be shifted. As before, I' is then shifted in W.

Finally, in the third case, we combine the techniques used in the previous two cases. First, we perform a binary search in I to find a point in I'. We then create a new interval for those points in I but not I' and delete the intervals in I' but not I by performing predecessor and successor searches. The other endpoint of I' can be found using a doubling search as before, and so we can then perform the necessary shifting.

It remains to analyze the cost of this structure, which we again do using the potential method. Our potential function Φ is the same as before, namely the number of intervals stored in W times $c \log\log n$, that is, $\Phi(D) = c|W| \log\log n$ where D represents the data structure and c is some sufficiently large constant. The key observation is that all but at most two of the predecessor and successor searches that we perform cause intervals to be deleted from W. Thus, for each search (which has actual cost $O(\log\log n)$), the potential decreases by $c \log\log n$. Therefore, the amortized cost of all such searches is $O(\log\log n)$.

Note that in the second case, the endpoints of I' are found using binary search in I, which we can afford because the working set number allows us to pay $O(\log |I|)$. Therefore, the amortized cost of the deletions and searches is $O(\log\log n)$, which,

as before, is absorbed by the $O((\log\log n)^2)$ term. However, in the first and third cases we may need to perform a doubling search in I' and this has cost $O(\log k)$. We therefore have

Theorem 4.4. *There exists a static data structure for one-dimensional range counting of size $O(n)$ with amortized expected search time $O(\log t(x) + \log k + (\log\log n)^2)$.*

4.5 Conclusion

We have seen how the working set property can be generalized to the problem of range searching in one dimension. In doing so, we described a data structure that supports one-dimensional range queries in $O(\log t(x) + k) + o(\log n)$ time and linear space and a data structure that supports one-dimensional range counting queries in $O(\log t(x) + \log k) + o(\log n)$ and linear space. Both of these data structures are static.

In the language of Iacono and Langerman [28], both of these data structures have the *weak working set property.*

Chapter 5

Conclusion

In this chapter, we summarize the results in this thesis and explore directions for future research.

5.1 Summary of Results

The contributions of this thesis are two-fold.

Distribution-sensitive dictionaries with low space overhead. Given a dictionary with insertion, deletion and search costs $I(n)$, $D(n)$ and $S(n)$ respectively, we present a dictionary with insertion and deletion times $I(n)$ and $D(n)$ and expected search time $S(t(x)) + I(t(x)) + D(t(x))$, where x is the element being searched for and $t(x)$ is the number of distinct accesses to the dictionary since x was last accessed. The space required for this data structure is $n + s(n) \log\log n$ where $s(n)$ denotes the space cost of the original dictionary. When a search for x is unsuccessful, $pred(x)$, the predecessor of x, can still be found in expected time $2^j \left(S(t(pred(x))) + I(t(pred(x))) + D(t(pred(x)))\right)$ using only $n + (k - j) \times s(n) + O\left(n^{1/2^j}\right)$ space. Both of these results can be combined with the best known implicit dictionaries to yield a dictionary that stores only the data required for its elements in addition to $O(\sqrt{n})$ words and supports insertions and deletions in worst-case $O(\log n)$ time and searches in expected $O(\log t(x))$ time, where $t(x)$ is the working set number

of the query x. If x is not in the dictionary, its predecessor $pred(x)$ can be returned in expected time $O(\log t(pred(x)))$.

Distribution-sensitive range searching in one dimension. We present a data structure for range searching of size $O(n)$ that answers range queries in amortized expected time $O(\log t(x) + k) + o(\log n)$, where k is the number of points reported and $t(x)$ is the working set number of the query interval as defined in Definition 4.1. This data structure can be extended to one of the same size that instead supports range counting queries in amortized expected time $O(\log t(x) + \log k) + o(\log n)$.

5.2 Future Work

There are several directions for future research.

For the problem of distribution-sensitive dictionaries with low space overhead, the following are possible directions for future research:

1. The idea of relying on the properties of the substructures (in this case, implicitness) proved fruitful. A natural question to ask, then, is what other substructure properties can carry over to the dictionary as a whole in a useful way? Other substructures could result in a combination of the working set property and some other useful properties.

2. In this thesis, we concerned ourselves with the working set property. There are other types of distribution sensitivity, such as the *dynamic finger property*, which means that query time is logarithmic in the rank difference between successive queries. One could also consider a notion complementary to the idea of the working set property, namely the *queueish property* [28], wherein query time is logarithmic in the number of items *not* accessed since the query item was last accessed. Are there implicit dictionaries that provide either of these properties? Could we provide any of these properties (or some analogue of them) for other types of data structures?

3. Is there a space-efficient or implicit data structure that does not rely on randomization but guarantees a *worst case* time complexity of $O(\log t(x))$?

For the problem of distribution-sensitive one-dimensional range searching, the following are possible directions for future research:

1. Are there other possible definitions of the working set number for intervals that lead to attainable query times? One observation is that our definition takes into account both the number of queries and their size; this is a similar notion to that of the unified property [26], but we only minimize on the number of queries. It would be nice to minimize on, say, the sum of the time since the last intersecting result and its distance or size. There does not seem be a straightforward use of the unified structure of Iacono [26] that provides this, however.

2. Can the results of this thesis be improved by reducing the $O((\log \log n)^2)$ term in the running time of either the range searching or counting case? Alternatively, can the results of this thesis be made deterministic, non-amortized, or both? Ideally, any modifications should continue to use only linear space.

3. Are there data structures for the same problem with different distribution-sensitive properties? One easy example is the dynamic finger property, which we saw in Chapter 4. What other notions can be generalized?

4. How can we generalize from the one-dimensional range searching problem to higher dimensions? In the plane, for instance, how might one define the working set number of a, say, orthogonal range query? One possibility would be the index of the most recent intersecting query times the number of points reported in that query. However, it does not seem obvious how to restructure the point set after queries. It would also be interesting to combine this problem with the first and achieve some sort of dynamic finger property in two dimensions.

5. What lower bounds can be stated for one dimensional range searching in terms of the distribution of queries?

Bibliography

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

[2] Sunil Arya, Theocharis Malamatos, and David M. Mount. A simple entropy-based algorithm for planar point location. In *SODA '01: Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 262–268, 2001.

[3] Sunil Arya, Theocharis Malamatos, and David M. Mount. Entropy-preserving cuttings and space-efficient planar point location. In *SODA '01: Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 256–261, 2001.

[4] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[5] J.L. Bentley. Multidimensional divide-and-conquer. *Commun. ACM*, 23(4):214–229, 1980.

[6] J.L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244–251, 1979.

[7] Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87, 1976.

[8] B. Chazelle and L.J. Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1:133–162, 1986.

[9] B. Chazelle and L.J. Guibas. Fractional cascading: II. applications. *Algorithmica*, 1:163–191, 1986.

[10] Richard Cole. On the dynamic finger conjecture for splay trees. part ii: The proof. *SIAM J. Comput.*, 30(1):44–85, 2000.

[11] 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 J. Comput.*, 30(1):1–43, 2000.

[12] Sébastien Collette, 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.

[13] Erik D. Demaine, John Iacono, and Stefan Langerman. Proximate point searching. *Comput. Geom. Theory Appl.*, 28(1):29–40, 2004.

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

[15] Gianni Franceschini and Roberto Grossi. Optimal cache-oblivious implicit dictionaries. In *ICALP '03: Proceedings of the 30th International Colloqium on Automata, Languages and Programming*, pages 316–331, 2003.

[16] Gianni Franceschini and Roberto Grossi. Implicit dictionaries supporting searches and amortized updates in $O(\log n \log \log n)$ time. In *SODA '03: Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 670–678, 2003.

[17] Gianni Franceschini and Roberto Grossi. Optimal worst-case operations for implicit cache-oblivious search trees. In *WADS '03: Proceedings of the 30th International Workshop on Algorithms and Data Structures*, pages 114–126, 2003.

[18] Gianni Franceschini and J. Ian Munro. Implicit dictionaries with $O(1)$ modifications per update and fast search. In *SODA '06: Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 404–413, 2006.

[19] Gianni Franceschini, Roberto Grossi, J. Ian Munro, and Linda Pagli. Implicit B-trees: New results for the dictionary problem. In *FOCS '02: Proceedings of the 43rd Symposium on Foundations of Computer Science*, pages 145–154, 2002.

[20] Greg N. Frederickson. Implicit data structures for the dictionary problem. *J. ACM*, 30(1):80–94, 1983.

[21] Greg N. Frederickson. Implicit data structures for the weighted dictionary problem. In *FOCS '81: Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*, pages 133–139, 1981.

[22] 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.

[23] John Iacono. Key independent optimality. In *ISAAC '02: Proceedings of the 13th International Symposium on Algorithms and Computation*, pages 25–31, 2002.

[24] John Iacono. Optimal planar point location. In *SODA '01: Proceedings of the 12th Annual ACM-SIAM symposium on Discrete Algorithms*, pages 340–341, 2001.

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

[26] John Iacono. Alternatives to splay trees with $O(\log n)$ worst-case access times. In *SODA '01: Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 516–522, 2001.

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

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

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

[30] Donald E. Knuth. *The art of computer programming, volume 3: sorting and searching.* Addison-Wesley, Reading, 2nd edition, 1998.

[31] Christos Levcopoulos and Mark H. Overmars. A balanced search tree with $O(1)$ worst-case update time. *Acta Inf.*, 26(3):269–277, 1988.

[32] George S. Lueker. A data structure for orthogonal range queries. In *FOCS '78: Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 28–34, 1978.

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

[34] J Ian Munro. An implicit data structure supporting insertion, deletion, and search in $O(\log^2 n)$ time. *J. Comput. Syst. Sci.*, 33(1):66–74, 1986.

[35] J. Ian Munro and Hendra Suwanda. Implicit data structures for fast search and update. *Journal of Computer and System Sciences*, 21(2):236–250, 1980.

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

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

[38] R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1986.

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

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

[41] 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.

[42] John W.J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7:347–348, 1964.