**Basic Research in Computer Science**



# Hashing, Randomness and Dictionaries

**Rasmus Pagh**

**BRICS Dissertation Series** **DS-02-5**

**ISSN 1396-7002** **October 2002**

**See back inner page for a list of recent BRICS Dissertation Series publications. Copies may be obtained by contacting:**

**BRICS**
**Department of Computer Science**
**University of Aarhus**
**Ny Munkegade, building 540**
**DK–8000 Aarhus C**
**Denmark**
**Telephone: +45 8942 3360**
**Telefax: +45 8942 3255**
**Internet: BRICS@brics.dk**

**BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:**

`http://www.brics.dk`
`ftp://ftp.brics.dk`
**This document in subdirectory** `DS/02/5/`

# Hashing, randomness and dictionaries

Rasmus Pagh

PhD dissertation


Department of Computer Science
University of Aarhus
Denmark

# Hashing, randomness and dictionaries

A dissertation
presented to the Faculty of Science
of the University of Aarhus
in partial fulfilment of the requirements for the
PhD degree

by
Rasmus Pagh
June 26, 2002

# Abstract

This thesis is centered around one of the most basic information retrieval problems, namely that of storing and accessing the elements of a set. Each element in the set has some associated information that is returned along with it. The problem is referred to as the *dictionary* problem, due to the similarity to a bookshelf dictionary, which contains a set of words and has an explanation associated with each word. In the *static* version of the problem the set is fixed, whereas in the *dynamic* version, insertions and deletions of elements are possible.

The approach taken is that of the theoretical algorithms community. We work (almost) exclusively with a *model*, a mathematical object that is meant to capture essential aspects of a real computer. The main model considered here (and in most of the literature on dictionaries) is a unit cost RAM with a word size that allows a set element to be stored in one word.

We consider several variants of the dictionary problem, as well as some related problems. The problems are studied mainly from an upper bound perspective, i.e., we try to come up with algorithms that are as efficient as possible with respect to various computing resources, mainly computation time and memory space. To some extent we also consider lower bounds, i.e., we attempt to show limitations on how efficient algorithms are possible. A central theme in the thesis is *randomness*. Randomized algorithms play an important role, in particular through the key technique of *hashing*. Additionally, probabilistic methods are used in several proofs.

The thesis begins with an overview of the main problems, models, and results within its subject area. Special emphasis is put on the relation to the contributions in the thesis. It then proceeds with nine chapters describing these contributions in detail, each chapter being a self-contained paper.

In Chapter 2 we present a simple dynamic linear space dictionary, called *cuckoo hashing*, with worst case constant lookup time. An implementation inspired by our algorithm, but using weaker hash functions, is empirically shown competitive with the best dictionaries having just an average case guarantee.

In Chapter 3 we describe a simple construction of *minimal perfect hash functions* for sets of size $n$: $x \mapsto (f(x) + d_{g(x)}) \bmod n$, where functions $f$ and $g$ are chosen from universal families, and the array $d$ contains $O(n)$ integers. A function of this form that is 1-1 on a given set can be found in expected time $O(n)$. The complexity of evaluating these functions turns out to be optimal with respect to the number of memory accesses and the number of multiplications.

Many algorithms employing hashing have been analyzed under the assump-

tion that hash functions behave like truly random functions. In Chapter 4 it is shown how to implement hash functions that can be evaluated on a RAM in constant time, and behave like truly random functions on any set of $n$ inputs, with high probability. The space needed to represent a function is $O(n)$ words, which is optimal and a polynomial improvement over previous results.

A static dictionary with constant query time, using space very close to the *information theoretical minimum* is the main result of Chapter 5. If $n$ denotes the number of elements in the set, the redundancy of the data structure is $o(n)$ bits, plus a negligible term depending on the word size of the RAM.

It is known how to construct a linear space static dictionary with constant lookup time in $O(n)$ expected time. In Chapter 6 we show how to construct such dictionaries *deterministically* in $O(n \log n)$ time. The running time of the best deterministic algorithm prior to this work was $\Omega(n^2)$. Using a standard dynamization technique, the first deterministic dynamic dictionary with constant lookup time and sublinear update time is derived.

Deterministic dynamic dictionaries are also the subject of Chapter 7. We present a linear space deterministic dictionary accommodating membership queries in time $(\log \log n)^{O(1)}$ and updates in time $(\log n)^{O(1)}$, where $n$ is the size of the set stored. Previous solutions either had query time $(\log n)^{\Omega(1)}$ or update time $2^{\omega(\sqrt{\log n})}$ in the worst case.

A new hashing primitive, *dispersing hash functions*, is introduced in Chapter 8. Very small dispersing families are shown to exist, which paves the way for several applications in derandomization (relational join, element distinctness, and static dictionary initialization). A close relationship between dispersing families and extractors is exhibited. The chapter also contains a lower bound on program size for hash functions that are "nearly perfect".

In Chapter 9 we study the two "components" of the static dictionary problem, membership and perfect hashing, in Yao's *cell probe* model. The first space and bit probe optimal worst case upper bound is given for the membership problem. We also give a new efficient membership scheme where the query algorithm makes just one adaptive choice, and probes a total of three words. A lower bound shows that two word probes generally do not suffice (this then applies also to the RAM). For minimal perfect hashing we show a tight bit probe lower bound, and give a simple scheme achieving this performance, making just one adaptive choice. Linear range perfect hashing is shown to be implementable with the same number of bit probes, of which just *one* is adaptive.

Finally, in Chapter 10 we consider dictionaries that perform lookups by probing a *single word* of memory, knowing only the size of the data structure. We describe a Las Vegas randomized dictionary where a lookup returns the correct answer with probability $1 - \epsilon$, and uses far less space than would be possible using a deterministic lookup procedure. In a certain augmented RAM model, our data structure supports efficient *deterministic* updates, exhibiting new probabilistic guarantees on dictionary running time.

# Acknowledgements

I would like to thank the many people that have, in one way or another, been part of making my time as a student such a wonderful period.

First of all I am grateful to my advisor, Peter Bro Miltersen, for being a great source of inspiration and knowledge, and for always taking the time to listen to my ideas, good as well as bad. As I have become more independent our meetings have become sparser, but you have remained a great person to come to for feedback on any aspect of my thesis work.

A special thanks of course goes to my co-authors Torben Hagerup, Peter Bro Miltersen, Anna Östlin, Jakob Pagter, and Flemming Friche Rodler. It has been a pleasure working with you, and I have benefited greatly from your insights.

A big thank you goes to the many people whose feedback have influenced the work in this thesis: Stephen Alstrup, Gerth Stølting Brodal, Andrei Broder, Andrej Brodnik, Martin Dietzfelbinger, Rolf Fagerberg, Peter Frandsen, Gudmund Skovbjerg Frandsen and his RPDA classes of 2001 and 2002, Thore Husfeldt, Riko Jacob, Kim Skak Larsen, Johan Kjeldgaard-Pedersen, Moni Naor, Morten Nyhave Nielsen, Jakob Pagter, Theis Rauhe, Kunihiko Sadakane, Peter Sanders, Ronen Shaltiel, Mikkel Thorup, Berthold Vöcking, and all those I forgot.

The department of computer science and BRICS have provided a very nice working environment and always supported my travel, which has been an important part of becoming a computer science researcher. I would also like to thank Rajeev Motwani for inviting me to Stanford, and thank the theory group there for receiving me so well.

The people nearby are important to the working environment. During the years I have been lucky to share offices with Jacob Elgaard, Anders Møller, Niels Damgaard, Daniel Damian, Martin Drozda, Jakob Pagter, Vanessa Teague, Flemming Friche Rodler, and Pablo Arrighi.

Last, but not least, I would like to thank my friends and family. Without a life outside of computer science I wouldn't have made it this far. Special thanks go to my parents Jørgen and Bodil for giving me a great upbringing, and to Anna for all her love.

*Rasmus Pagh,*
*Århus, June 26, 2002.*

# Contents

# Chapter 1

## Overview

The demand for rapid access to information has grown tremendously over the last few decades. During the same period, the amount of information to be handled has exploded. The end of this trend seems nowhere in sight. Computer scientists and engineers are met with the challenge of designing systems whose performance keeps up with these ever increasing requirements. A theoretical understanding of the inherent possibilities and limitations of computing equipment is instrumental in this quest.

This thesis is centered around one of the most basic information retrieval problems, namely that of storing and accessing the elements of a set, where each element has some associated information. This retrieval problem is mostly referred to as the *dictionary* problem, due to the similarity to a bookshelf dictionary, which contains a set of words and has an explanation associated with each word. Another example of a dictionary is the database of all residents in Denmark, indexed by the set of so-called CPR numbers. Given such a number, public institutions need to retrieve information associated with it (name, address, tax record, and so on). The importance of dictionaries is not so much due to direct applications like this, but rather due to the large number of algorithms and data structures that have a dictionary as a substructure. In many of these applications, the performance of the dictionary is critical.

We study several variants of the dictionary problem, as well as some related problems. Section 1.1 states and discusses these problems. The problems are studied mainly from an upper bound perspective, i.e., we try to come up with algorithms that are as efficient as possible with respect to various computing resources, mainly computation time and memory space. To some extent we also consider lower bounds, i.e., attempt to show limitations on how efficient algorithms are possible.

The approach taken is that of the theoretical algorithms community. We work (almost) exclusively with a *model* that is meant to capture essential aspects of a real computer. This should preferably say something about the performance of algorithms on present and future computers. At the same time it should be reasonably simple, such that it is possible to handle mathematically. The models relevant to this thesis are presented in Section 1.2. Section 1.3 surveys the known results on dictionaries and related problems in various models, with special emphasis on their relation to the contributions in the thesis.

## 1.1 Problems

The algorithmic problems considered in this thesis are all related to sets. We consider finite subsets of some set $U$, called the *universe*. This section first categorizes abstract data structures for sets according to four orthogonal criteria. We then state some other problems related to sets.

### 1.1.1 Abstract data structures for sets

We are interested in data structures storing a set $S \subseteq U$. Such data structures can be categorized according to what queries are supported, whether they are static or dynamic, whether the universe $U$ is finite, and whether associated information is supported.

**Queries supported.** Three of the most basic queries on sets are:

- MEMBERSHIP. Does $x$ belong to $S$?
- REPORT. What are the elements of $S$?
- SIZE. What is the size of $S$?

If $U$ is an ordered set, several other queries become relevant:

- RANGE REPORTING. What are the elements in $S$ in the range $[x, y]$?
- PREDECESSOR. What is the largest element in $S$ smaller than $x$ (if any)?
- SUCCESSOR. What is the smallest element in $S$ larger than $x$ (if any)?

We will mainly be interested in data structures supporting membership queries. However, the more general queries on ordered sets will also show up. The report and size queries are usually trivial to implement optimally, so we will not explicitly discuss them.

**Static or dynamic.** A *static* data structure is one that does not change over time. In our case a static data structure represents a fixed set. A *dynamic* data structure changes over time as a result of updates. The updates we consider are:

- INSERT. Include the element $x$ in the set.
- DELETE. Remove the element $x$ from the set.

Data structures supporting either INSERT or DELETE, but not both, are called *semi-dynamic*.

Many other update operations could be considered, in particular union and intersection with another set, and set difference. However, in general it is not known how to perform these operations more efficiently than through insertion and deletion of single elements, without sacrificing the performance of membership queries. We thus restrict our attention to these basic updates.

**Finite or infinite universe.** Some set data structures deal with subsets of an infinite universe $U$, e.g., the set of all strings over some alphabet. Others deal with a finite universe, such as integers in some range, or fixed length strings. The latter will be the case for all data structures considered in this thesis.

**Associated information.** In many applications of set data structures, each element of $S$ has some information associated with it (sometimes called *satellite information*). Insertions then specify both an element and some associated information. The membership query is replaced by the lookup query.

- LOOKUP. Does $x$ belong to $S$? If so, what is its associated information?

Set data structures with associated information are called *dictionaries*. We will only consider the case where the associated information is an element from some finite set. However, most dictionaries are easily extended to support associated information from infinite sets, such as the set of strings.

### 1.1.2 Set related problems

We now consider some problems that are in one way or another connected with sets. The first problem, perfect hashing, will play an important role in several parts of the thesis, whereas the other problems are mainly *applications* of set data structures, or of the underlying techniques such as hashing.

**Data structures for perfect hashing.** An important tool in designing efficient data structures for membership queries is *perfect hashing*. This technique consists of associating with any set $S$ a function $h : U \to \{1, \ldots, r\}$ that is 1-1 on $S$. The function $h$ is called a *perfect hash function* for $S$. The range $\{1, \ldots, r\}$ indexes an array where each element $x \in S$ is written in entry number $h(x)$. Deciding membership of $x$ is then a matter of looking up entry number $h(x)$ in the array.

Except for degenerate cases, a perfect hash function $h$ for a set must depend on the set. That is, such a function must be found and stored. The representation can be thought of as a data structure capable of answering function value queries. Since want to always be able to find a perfect hash function, $h$ must come from some family $\mathcal{H}$ that contains a perfect hash function for any set (of less than some maximum size). Such a family is called *perfect*. For a given perfect family $\mathcal{H}$, the data structure problem is to represent a function $h \in \mathcal{H}$ and support function value queries.

- EVALUATE. What is the value of $h$ on input $x$?

Perfect hashing can also be part of a set data structure. In that case the hash function represented is 1-1 on the set stored. If the set data structure is dynamic, the perfect hash function must change along with the set.

**Relational operations.** A *relational join* is the operation of computing the intersection of two sets, having information associated with elements. The combined information from the two sets must be associated with the elements of the result. Join is a key operation in relational databases.

Other operations of interest in relational databases are *union* of sets and *set difference*. Clearly, all of these operations can be performed using a dictionary, in a number of insertion and lookup operations proportional to the size of the sets.

**Element distinctness and set cardinality.** The *element distinctness* problem is to decide whether a list of elements from some universe $U$ are all distinct. A more general problem is to determine the *cardinality* of the set formed by the elements in a list. These problems can be solved using a set data structure, with a number of insertion and membership operations proportional to the length of the list.

## 1.2 Models

In order to theoretically study problems as those stated in the previous section, one needs a computational *model* that describes the setting in which computation takes place. Theoretical study of a problem in a certain model has the advantage of being *timeless*, in the sense that the model does not change over time. This is in contrast to experimental work, where even small changes in computer architecture may render previous work obsolete. Of course, we also want the model to be *relevant*, i.e., say something about real-world computation. A good model is one that is highly relevant and will stay so in the future.

Unfortunately, good modeling is not easy. There are many (sometimes competing) models that try to capture different aspects of computers not to mention other artifacts of interest in computer science such as circuits and networks. Some are more realistic than others, but all models used in theoretical computer science are *simplified* versions of (an exact model of) a real computer, in which only the "essential" features of a computer are considered. Simplification serves two main purposes:

- **Clarity.** Results are more easily understood, and can more easily be compared.
- **Analyzability.** Simpler models are often easier to analyze than more complex ones.

In the following, we describe a number of models relevant to this thesis.

### 1.2.1 RAM models

Nearly all models considered in this thesis are so-called *word RAM* models that reflect the architecture of traditional single processor computers. The memory of such computers is an array of bit strings called *words*. (In some related, less

realistic RAM models, memory cells can contain arbitrarily large integers.) All computation takes place in a central processing unit (CPU) that has a constant number of words of local memory (registers), on which computation can be performed. Words from main memory can be loaded into registers, and the content of registers can be written to main memory. In both cases, a certain register specifies the *address* (index in the array) of the main memory word in question. This capability is called *indirect addressing*, and distinguishes these models from, for example, pointer machine models. The capability to access memory words in arbitrary patterns is the explanation for the name "RAM", which stands for *random access machine*.

A finite sequence of computational operations (called *instructions*) specifies an algorithm. An instruction can be described in a fixed number of machine words. The next instruction to execute is determined by a special register (called the *program counter*) that is incremented by 1 after each instruction, but may also be set (conditionally) to some other value, facilitating jumps in the instruction sequence.

We now discuss four independent aspects of our RAM models: Computation, memory, randomization, and (bounded) parallelism.

### Computation

A main distinction between various word RAM models lies in what kinds of basic instructions are available, what their cost is, and in what way instructions are allowed to depend on the word length.

**Instruction sets.** In describing various instruction sets, we will consider the words of a RAM both as bit strings and as integers in the usual binary encoding (signed or unsigned). Below, some classes of instruction sets are described.

- **`C`-like instruction sets.** Perhaps the most common word RAM models have an instruction set that is a subset of the primitive operations provided by imperative programming languages such as `C`. This is often referred to as a "standard instruction set". In particular, such instruction sets include addition, subtraction, comparison with zero, bitwise boolean operations, and shifts. They may also include multiplication and division. For historical reasons division is usually left out. Many papers pay special attention to avoiding multiplication, as it is the only instruction not in the circuit class $AC^0$. Historically, it was also an expensive operation on real computers, but the gap to the cost of other instructions seems to be diminishing. In this thesis we mainly consider a RAM with a `C`-like instruction set including multiplication but not division.

- **Extended instruction sets.** In some cases, a standard instruction set does not (easily) allow a certain computation on registers to be performed efficiently. It may then be of interest to study the effect of extending the instruction set by this operation. If the benefit is sufficient, processor designers might then choose to include the operation in the instruction set.

In some cases, like the extended instruction sets considered in this thesis, the computational complexity (or circuit complexity) of performing the operation is not known. Then, of course, it is not immediate that hardware implementation will ever be an option. However, an efficient algorithm for such a model will provide a *conditional* result: If the complexity of the operation is sufficiently low, there is an efficient hardware/software implementation. When completely disregarding the complexity of the operations, we have an interesting information theoretic model for showing lower bounds, that seems to have received little attention. One reason may be that current lower bound techniques apply also to the stronger "cell probe" model described below.

- **Circuit based models.** Some models have an unbounded number of instructions. In the *Circuit RAM* model [AMRT96], for example, any function of the registers with a polynomial size circuit is an instruction. (The cost of executing an instruction is then its unbounded fan-in circuit depth.) In the more restricted $AC^0$ *RAM* model, only functions having polynomial size, *constant depth*, unbounded fan-in circuits are instructions.

**The cost of performing instructions.** There are several ways of assigning cost to the execution of instructions. Usually the cost is thought of as time, but it could also model, for example, energy consumption. The most common cost measure, and the one employed in this thesis, is *unit cost*, i.e., execution of any instruction has cost 1. This has been a fairly good approximation to real computers. However, from a theoretical point of view this does not give realistic asymptotical results, as the time for computing some functions grows with the word length. One class of instructions that (theoretically) scales to arbitrary word length, is that of functions with polynomial size, constant depth, unbounded fan-in circuits, i.e., the instruction set of the $AC^0$ RAM. This is the reason for using the $AC^0$ RAM together with the unit cost measure.

One of the most conservative cost measures is to charge a cost proportional to the number of bits "involved in the operation" (e.g., excluding any initial zeros in arithmetic operations). This is the *log cost* measure, which originates in the study of RAMs where registers can contain arbitrary integers. For the *circuit depth* measure (mentioned above), the cost is simply the smallest circuit depth of a polynomial size, unbounded fan-in circuit for the instruction. Both of these models seem to have received relatively little attention in the data structures literature.

**Uniformity.** Some algorithms employ constants that depend on the word length. One can think of these constants as computed at "compile time". In the terminology of Ben-Amram and Galil [BAG97], an algorithm is *uniform* if constants depending on $w$ can be computed in time polynomial in $w$. If suitable constants are merely known to exist, the algorithm is called *weakly nonuniform*. The algorithms of Chapters 6 and 7 in this thesis are weakly nonuniform.

### Memory

There are three figures of interest in connection with the memory of algorithms on RAMs: The *word length*, the *space usage*, and the *cost* of accessing memory words.

**Word length.** The word length is the length of the bit string in a machine word. It is usually denoted by $w$. When studying set data structures, the word length is often linked to the size of the universe $U$, such that the elements of $U$ are (or can be described by) bit strings of $w$ bits. The assumption that the elements considered have a size that matches the word size is sometimes referred to as the *trans-dichotomous* assumption [FW94], as it bridges the dichotomy between the parameters of the problem and the model in a reasonable manner.

**Space usage.** The *space* used by an algorithm is simply the largest index of any memory register accessed during its computation.

**Access cost.** The most common cost measure for memory accesses, and the one used in the papers comprising this thesis, is *unit cost*, where the cost of every memory access is 1. Again, there are theoretical arguments that this assumption is not realistic for large memories. Indeed, the development of hardware in the last few decades has resulted in a *memory hierarchy*, where different parts of memory take different time to access. There are small amounts of very fast memory and large amounts of slower memory.

One model that tries to capture this is the *log cost* model, which assigns a cost of $\log(i)$ to access memory word number $i$. However, it has been more popular to think of memory as divided into several parts with different access costs. Usually, accesses to slow parts of memory are done in a *batched* fashion, where *blocks* of consecutive words are moved to a faster memory area. This models the behavior of real computers, which deal with the latency of accessing slow parts of memory by transferring many words at a time. The most popular model for this, called the *I/O model* [AV88], considers only two levels of memory (*internal* and *external*). In the pure model, only block transfers between internal and external memory (*I/Os*) are counted, and the cost of internal computation is ignored. This is to some extent justified by the fact that the time for accessing a block of memory on a disk (the prototypical external memory) is millions of times larger than the time for executing a CPU instruction.

The I/O model can be used to derive algorithms that are optimal with respect to transfers between two memory levels. In many computer systems, the block transfers between two specific neighboring layers of the memory hierarchy are the performance bottleneck. Recently, a lot of attention has been devoted to so-called *cache oblivious* algorithms [FLPR99], that are simultaneously optimal with respect to any two neighboring levels of the memory hierarchy.

#### Randomization

Randomized algorithms, and how they compare with deterministic ones, is a key theme in this thesis. For randomized algorithms we use a model where a special instruction assigns random and independent values to a specified number of bits in a register.

For many years, randomized algorithms have not been implementable on most computers, due to the lack of a source of random bits. In practice they have been used extensively with deterministic "pseudorandom" number generators. However, in most cases this is merely a heuristic with no proven performance guarantee. This may be about to change, as chipsets with hardware random number generators are becoming increasingly widespread (probably mainly thanks to applications in cryptography).

Whether it is physically feasible to generate a truly random and independent sequence of bits is a matter of debate. Using weaker, more realistic random sources for various randomized algorithms has received some attention, but we do not go into this.

An idealized model that is sometimes used when analyzing hashing algorithms is the *random oracle model*. In this model, algorithms have access to a truly random function from the set of machine words to the set of machine words that can be evaluated in constant time. A justification for this model is the empirical observation that a truly random function can in practice often be replaced by a pseudo-random function.

#### Parallelism

While this thesis does not deal with truly parallel computing models, we do in some cases point out algorithmic possibilities for *limited* parallelism. For example, in the external memory of a computer with two parallel disks, one can make two I/Os simultaneously on the disks. Limited parallelism in accessing different banks of internal memory, and when performing independent threads of computation are also clear possibilities. Exploiting such parallelism seems to be a trend in emerging computer systems. Of course, limited parallelism will not asymptotically speed up algorithms. Still, it is of interest to explore to what extent it can be used to speed up basic data structures such as dictionaries, where a factor of two in speed may mean a lot.

### 1.2.2 Lower bound models

We now consider some models that are variants of the above, and have been introduced mainly for the purpose of showing lower bounds.

**Comparison-based models.** Some models considered when dealing with sets put restrictions on what can be done with elements of the set. In *comparison-based* models, set elements can be moved and copied in memory, and pairs of elements can be compared (with respect to some ordering of the universe). No other operations involving set elements are allowed.

It is often relatively easy to show lower bounds on the number of comparisons needed by comparison-based algorithms, and in many cases there are also matching upper bounds, achieved by algorithms where the time complexity is dominated by the time for comparisons. The lower bound then means that one needs to explore non-comparison-based algorithms to hope for better performance.

**Implicit models.** Other models for set problems put restrictions on what can be stored in memory. In *implicit* data structures, only elements of the set itself can be stored. The most restrictive implicit model allows each element to occur exactly once. An argument in favor of such implicit data structures is that they are space efficient. However, in some cases the space usage of nonimplicit data structures can be considerably lower (see Chapter 5).

The main reason for studying implicit data structures is probably the fact that lower bounds can be shown. For example, when the size of $U$ is large, a sorted list is an optimal implicit data structure for the membership problem, by a result of Yao [Yao81]. Yao's lower bound technique extends to a more general class of data structures, which we call *semi-explicit*. In such data structures pointers are allowed, elements may be repeated, and it is allowed to use space that is any function of the size of the set stored.

**The cell probe model.** In the *cell probe* model, the complexity measure is the number of probes to memory cells. The memory layout is like that of word RAM models. Computation of how to perform the memory probes, etc., is for free.

The cell probe model was introduced by Yao [Yao81] for the purpose of showing lower bounds that apply to RAM models. Upper bounds in the cell probe model show limits on how good lower bounds can be achieved. They can also be a first step towards upper bounds in more realistic models. One special case of the cell probe model that has received special attention is the *bit probe* model, where each cell contains a single bit. The cell probe model is considered in Chapter 9.

## 1.3 Questions

The basic question when considering an algorithmic problem in some model of computation is: What is the cost of an optimal algorithm for solving the problem. Often there are several cost measures, which may be considered individually or in combination. Of interest is also how the cost in different models relate.

In this section we survey some of the most interesting questions concerning the membership and dictionary problems in various models that have been studied. In a few cases we also consider other problems from Section 1.1. Special attention is paid to the relation to the contributions of the thesis.

We start by surveying some basic results that form the starting point of a more detailed investigation. We then proceed to discuss in detail the following

questions:

- **How few memory probes?** The number of memory probes is crucial for the performance of algorithms on modern computers, where memory access is often a bottleneck. It is thus of interest to know the exact number of memory probes needed for certain basic tasks, such as lookup in a dictionary. Also, most lower bounds for data structures have been stated in terms of the number of memory probes.

- **How adaptive?** With limited parallelism being a trend in emerging computers, it is of interest to have algorithms whose operations are to some extent independent. We consider the adaptivity of query algorithms, i.e., the way in which memory probes depend on the results of earlier memory probes.

- **How little computation?** The amount of computation necessary to carry out a task is probably the most researched (and one of the least understood) complexity measures in computer science. In this thesis the question mainly concerns the cost of instructions in various RAM models.

- **How little space?** The space usage of a data structure is a parameter of great practical importance. In recent years there has been considerable interest in data structures whose memory usage is near the lower bound provided by information theory. We consider this question for membership and perfect hashing.

- **How little randomness?** Randomness is a resource, just as time and space. It may not be available, or we may wish to use as little as possible. The question is therefore to what extent, and at what cost in other complexity measures, algorithms can be derandomized.

- **What kind of performance guarantee?** Several types of guarantees can be made on the performance of algorithms. Bounds may hold in the amortized sense or for every operation, and for randomized algorithms they may hold in the expected sense or hold with high probability. Also, bounds may hold only under some assumption on the input.

There are, of course, dependencies among the above questions. For example, given sufficient space the answers to most of the other questions become trivial. Also, the presence of some amount of randomness seems important to good upper bounds for many of the questions. Asking these questions separately therefore only makes sense if one always keeps the other questions in mind.

### 1.3.1 Basics

We now describe some basic results on the problems described in Section 1.1, roughly "what every algorithmicist should know" (and a bit more). We restrict our attention to a RAM with unit cost instructions and unit cost memory accesses, and to the case where the universe equals the set of machine words

(the trans-dichotomous assumption). Unless stated otherwise, these results are for the C-like instruction set described in Section 1.2.

We start with the most general (and hardest) problems, and move towards easier problems. In particular, upper bounds have implications downwards, and lower bounds have implications upwards.

### Predecessor queries

Predecessor queries and successor queries are of equivalent algorithmic difficulty in virtually any conceivable model. It is therefore common to consider just predecessor queries.

**Search trees.** The classic data structure for answering predecessor queries is the balanced binary search tree, that supports predecessor queries as well as dynamic updates in worst case $O(\log n)$ time. This is optimal among comparison-based data structures.

One approach to improving the query time compared to binary trees is to uses trees that have larger degree and thus smaller height. Such trees were developed for external memory applications in the 1970s [BM72]. However, it seems that the utility of this approach for internal memory algorithms was not recognized until Fredman and Willard's paper on fusion trees [FW93]. The crux of fusion trees is to have a small data structure residing in each node, such that the correct search path can be determined in constant time per node. Fredman and Willard show how to do this for nodes of degree $(\log n)^{\Omega(1)}$, which means that the tree has height (and thus search time) $O(\log n/\log\log n)$. Rebalancing can be done in amortized time $O(\log n/\log\log n)$ per update.

The approach of Fredman and Willard generalizes (see, e.g., [Hag98b]) to trees with nodes of degree $w^{\Omega(1)}$, giving height and query time $O(\log_w n)$ with linear space usage and linear time deterministic construction. This dynamizes to allow deterministic updates in the same time bound in at least two situations: 1. For the $AC^0$ instruction set [AMT99, Hag98b], and 2. When the algorithm has access to a certain auxiliary table, depending only on $w$ and $\lceil \log n \rceil$, of size $2^{n^{\Omega(1)}}$ words [FW94, Wil00].

**Recursion on smaller universes.** Search trees are a way of doing recursion on the size of the set handled. Another approach is to do recursion on the size of the universe. This was first explored by van Emde Boas [vEB75], and the space usage was subsequently improved by Willard's Y-fast tries [Wil83]. Both data structures support predecessor queries in time $O(\log w)$. Van Emde Boas trees are deterministic and support insertions and deletions in time $O(\log w)$. However, the space usage is $2^{\Omega(w)}$ words. Using a dynamic dictionary, Y-fast tries improve the space usage to $O(n)$ words. The time for queries and updates remains the same, assuming that the dictionary uses constant time for all operations. As will be seen below, this implies that the bounds become randomized.

In the static case, the query time was improved to $O(\log w/\log\log w)$ by Beame and Fich [BF99a]. Their data structure uses space $n^{O(1)}$ and can also be

deterministically constructed in time $n^{O(1)}$. Using this data structure for word length $w = 2^{O(\sqrt{\log n \log\log n})}$, and the generalized static fusion tree for larger word length, one can achieve query time $O(\sqrt{\log n / \log\log n})$ in all cases.

**From static to dynamic.** Andersson's exponential search trees [And96] is a general deterministic transformation, turning static predecessor data structures with polynomial space usage and construction time into linear space dynamic data structures. The query time and amortized update time is $O(\log\log n)$ times larger than the query time for the static data structure. For static structures with $(\log n)^{\Omega(1)}$ query time, the query time grows only by a constant factor. This means that from the above static data structure we get a dynamic deterministic predecessor data structure supporting all operations in time $O(\sqrt{\log n / \log\log n})$.

In the original formulation of exponential search trees, update bounds are amortized, but Andersson and Thorup have shown how to make the update bounds hold in the worst case [AT00]. The transformation itself introduces only standard $AC^0$ instructions.

**Lower bound.** The above static upper bounds are met by lower bounds of Beame and Fich [BF99a] that hold for space $n^{O(1)}$. In particular, the best possible bound on query time in terms of $w$ is $\Theta(\log w / \log\log w)$, and the best possible bound in terms of $n$ is $\Theta(\sqrt{\log n / \log\log n})$.

### Range queries

Range queries can easily be performed using a predecessor dictionary. By maintaining an ordered linked list of the set elements, the elements in an interval can be reported each in constant time, plus the time for an initial predecessor query. However, as pointed out by in [MNSW98] the lower bounds applying to predecessor queries do not hold for range queries. In particular, constant time per element reported was achieved with a static data structure of $O(nw)$ words. The space usage was improved to the optimal $O(n)$ words by Alstrup et al. [ABR01]. This data structure has expected construction time $O(n\sqrt{w})$.

It appears that all published dynamic data structures supporting efficient range queries have also supported predecessor queries. Extending data structures that support fast membership queries to support fast range queries seems to be an interesting uncharted territory in data structures.

### Membership queries

Membership queries are at the core of this thesis. In some cases the best data structures for membership actually answer predecessor queries (and this is important in the underlying recursion). However, in most cases more efficient solutions are known. When nothing else is mentioned, the membership data structures discussed below can be extended to dictionaries with the same performance.

**Space inefficient solutions.** A word can be regarded as a string of $d$-bit "characters". These characters can be used to guide a search in a $2^d$-ary tree, called a *trie* [Fre60]. The time for a search is $O(w/d)$, but $2^d$ pointers are needed in each node, so the space usage is $\Omega(2^d n)$ in the worst case. Thus, saving a factor $d$ in time compared to a binary trie costs a factor $\Theta(2^d)$ in space. In particular, if $n^{O(1)}$ space usage is desired, the query time is $\Omega(w/\log n)$.

Choosing $d = w$ we get a *bit vector* which has *constant* time queries and updates. Also, initialization to the empty set can be done in constant time, using a folklore trick often posed as a problem in algorithms classes (see, e.g., [AHU75, Exercise 2.12]). Bit vectors waste a large amount of space unless $w$ is close to $\log n$.

**The hashing heuristic.** Hashing emerged as a heuristic for rapidly conducting searches for elements in a dictionary. Knuth [Knu98] attributes the original idea of hashing to Hans Peter Luhn, who described it in an internal IBM memorandum in 1953. Hashing was first described in open literature by Dumey [Dum56]. The method uses a "hash function" $h$ defined on $U$ to decide where to store elements. The initial idea is to try to store an element $x \in S$ in entry $h(x)$ of an array (called the *hash table*). In general, several elements may have the same hash function value, so one needs some way of resolving collisions. Many such collision resolution schemes have been considered – among the most prominent are chained hashing, linear probing, and double hashing. Such algorithms were historically analyzed in the random oracle model, that is, assuming that the hash function is a truly random function. (Equivalently, one can assume that the *keys* inserted are random and independent.) If the hash table has size $n/\alpha$, for $0 < \alpha \leq 1$, it is said to have *load factor* $\alpha$. When $\alpha = 1 - \Omega(1)$, all the mentioned collision resolution schemes result in expected constant time for updates and queries. The dependence on $\alpha$ varies among the collision resolution schemes, and has been studied in great detail. Chapter 2 describes some of these results – for more comprehensive overviews we refer to the books of Gonnet [Gon84] and Knuth [Knu98].

Most hash functions used in practice have been fixed and deterministic. However, the behavior of hashing algorithms turns out to closely follow the behavior predicted by the analysis assuming a truly random hash function. Of course, for worst case inputs every key may have the same hash function value, so using a fixed hash function is merely a heuristic. Another point of view is that the analysis of hashing schemes using a fixed hash function is done under the assumption that the keys inserted are independently random. Though there may be some randomness in real-life inputs, the assumption of independence is rarely realistic.

**Provably good families of hash functions.** One could in principle implement hashing schemes using a truly random hash function, but the space needed to store a general function is enormous (comparable to that of a bit vector). Therefore this approach is not feasible. Hash functions that can be stored in a reasonable amount of space are needed.

In the late 1970s, Carter and Wegman succeeded in removing distribution assumptions from the analysis of chained hashing [CW79]. The essential observation is that less than a totally random hash function suffices. If the the hash function is chosen from a family of functions such that any *pair* of distinct elements in $U$ have a low probability of colliding, it is "universally applicable". Carter and Wegman gave a construction of such *universal* families whose functions can be stored in $O(1)$ machine words and evaluated in constant time.

Since the result of Carter and Wegman, many hash function classes have been proposed, e.g. [ADfM$^+$97, BCFM00, Chi94, Df96, DfGMP92, DfMadH90, GW97, IMRV99, LS98, Sie89], and their performance analyzed in various settings. For example, Schmidt and Siegel [SS89, SS90a] analyzed the expected insertion time of linear probing and double hashing, implemented using the hash functions of Siegel [Sie89]. However, such analyses have been made only for particular hashing schemes (or relatively narrow classes of hashing schemes), and have only dealt with certain performance parameters.

**Perfect hashing.** Universal hashing placed chained hashing on a solid theoretical basis. However, bounds were still only *expected* constant, and large deviations from the expectation could very well occur. A tantalizing possibility is to construct (and possibly maintain) a *perfect* hash function $h$, which is 1-1 on the set stored. Lookup of $x$ could then be performed by inspecting entry $h(x)$ of a hash table (which would need to be updated along with the hash function). If the perfect hash function can be evaluated in worst case constant time, the time for lookups is also constant in the worst case.

It is interesting to note that the construction of perfect hash functions with $O(n)$ space usage (including space for the hash table) eluded researchers in the late 1970s and early 1980s. Such a perfect hash function construction in the case $|U| = n^{O(1)}$ had been found by Tarjan and Yao [TY79]. A simple fact, easily derived from the definition of universality, reduces the general case to this one: A random function from a universal family with range of size $n^2$ is 1-1 on a set of size $n$ with probability at least $1/2$. Thus, composition of a function from a universal family with a Tarjan-Yao hash function yields a perfect hash function for any set.

Fredman, Komlós and Szemerédi [FKS84] published the first construction of a perfect family of hash functions with $O(n)$ word representation. They used the fact just mentioned, although they did not explicitly refer to universal hashing. Instead of using Tarjan and Yao's result, they show that one can get the perfect hash function by replacing the linked lists in chained hashing by perfect hash functions from a universal family and corresponding tables. Two algorithms for construction of a perfect hash function were given: A randomized one running in expected time $O(n)$, and a deterministic one with a running time of $O(n^3w)$. This static dictionary is often referred to as the *FKS scheme*. It can be made dynamic, supporting insertions and deletions in amortized expected constant time [DfKM$^+$94].

**Other set problems**

All the "set related problems" mentioned at the end of Section 1.1 can be solved in linear expected time using a dictionary with expected constant time operations. The best deterministic time bounds with reasonable space usage seem to be those achieved by sorting the elements. The currently fastest deterministic sorting algorithm runs in time $O(n \log\log n)$ [Han02] and uses linear space. For small and large word length there are faster sorting algorithms. For word length $w = O(\log n)$ radix-sorting can be done in linear time. With the $AC^0$ instruction set, deterministic sorting is possible in linear time when $w = n^{\Omega(1)}$ [Hag98b].

We now go on to look at the questions asked at the beginning of the section.

### 1.3.2 How few memory probes?

The number of memory probes performed by an algorithm is a well-studied complexity measure, especially from a lower bound perspective.

**Cell probe complexity.** The model in which memory probe lower bounds are usually shown is Yao's cell probe model. These bounds then also apply to RAM models. Many such lower bounds for dynamic and static data structure problems are known, e.g., [BF99a, FS89, AHR98].

One nonconstant lower bound for (trans-dichotomous) membership data structures is known. It appears in an unpublished manuscript by Sundar [Sun93], and is an amortized lower bound of $\Omega(\frac{\log\log_w n}{\log\log\log_w n})$ memory accesses per operation for any such *deterministic* data structure using $n^{O(1)}$ words of space, when the word length is $w = \Omega(\log n \log\log n)$.

In Chapter 9 we show that, in general, at least *three* memory probes are needed for membership queries in linear space data structures. This matches the upper bound achieved by the FKS scheme. In particular, this means that perfect hashing must require at least two memory probes. This is again achieved by the FKS scheme, but only in the case where the range of the function is some constant factor larger than $n$. For range of size $n$ one can instead use the perfect family of hash functions described in Chapter 3, which matches the FKS perfect hash function family in all other respects. A more space efficient, but less practical, solution is given in Chapter 9.

The worst case bit probe complexity of membership, when using within a constant factor of optimal space, was shown in [BMRV00] to be $\Omega(w - \log n)$. This is matched by an upper bound given in Chapter 9. The upper bound is probably mainly of theoretical interest.

If one considers randomized query procedures, the lower bounds mentioned above can be broken. Buhrman et al. [BMRV00] prove the existence of an $O(nw)$ bit Monte Carlo membership data structure where one bit probe suffices to answer a query. The probability that the answer is correct can be made arbitrarily close to 1. This data structure does *not* extend to a dictionary with the same performance. A stronger Las Vegas data structure, that returns "*don't*

*know"* rather than wrong answers, is presented in Chapter 10. It looks up one word rather than one bit, and has a space usage of $O(nw)$ words. This data structure extends to a dictionary with essentially the same performance. For both data structures there is no known implementation that is also efficient in terms of the amount of computation. If one desires the amount of computation for a query to be $w^{O(1)}$, the presently best space usage is a factor $2^{(\log w)^{O(1)}}$ higher than above [TS]. Other explicit upper bounds for membership, in the case where few bit probes are needed, are shown in [RRR01].

**I/O complexity.** Both upper and lower bounds in the (pure) I/O model are expressed in terms of the number of accesses to external memory (I/Os). There are two differences to the cell probe model: 1. $B$ data items, rather than one, fit in one external memory cell. 2. Computation can only take place on data in internal memory.

The fact that many data items fit in one external memory block makes the job of implementing hashing based dictionaries easier. For example, schemes like linear probing and chained hashing will rarely require more than one I/O (at least when using sufficiently strong classes of hash functions such as $O(\log n)$-wise independent families, see [Sie89] for constructions). Predecessor queries can also be done in fewer memory accesses, namely in $O(\log_B n)$ I/Os, using B-trees. Though any algorithm in this thesis could in principle be implemented on external memory, none of them would take advantage of the fact that a memory access yields a block of data elements, rather than just one.

### 1.3.3 How adaptive?

The adaptivity of algorithms is a performance measure that seems to have received relatively little attention. Of course, algorithms for parallel machines often exhibit a certain kind of low adaptiveness, namely that the threads of computation on different processors do not depend much on each other. Here, we consider a much more restricted kind of parallelism, namely that made possible when consecutive instructions in a program are independent. Such parallelism seems to be increasingly exploited in modern processors. The setting that we consider is one in which independent memory accesses can be done in parallel. This kind of parallelism does not yet seem to be exploited by single processor computers, except in the I/O model where memory accesses to different disks can be done in parallel.

As stated above, the FKS trans-dichotomous membership data structure uses three memory accesses, which is optimal. The lower bound shown in Chapter 9 in fact states that, if constant lookup time is desired, then at least two memory accesses must depend on the results of previous memory accesses. In the FKS scheme the location of the third memory access depends on the result of *both* two previous memory accesses. It is shown in Chapter 9 that in the *cell probe* model this can be improved such that the locations of the two last memory accesses depend only on the result of the first memory access (which is to a fixed location). This means that these memory accesses can be done in parallel. On the RAM something slightly weaker can be achieved: $O(1)$

memory accesses to a relatively small hash function description of $n^\epsilon$ words, where $\epsilon > 0$ can be any constant, followed by two independent accesses to hash tables of size $O(n)$. A dynamic version of this hashing scheme is described in Chapter 2.

The randomized trans-dichotomous membership data structure described in Chapter 10 that looks up only one word is of course nonadaptive. However, the space usage is $O(nw)$ words rather than $O(n)$ words. Another efficient nonadaptive query scheme is possible in the bit probe model, where the optimal $O(w - \log n)$ bit probes and optimal space can be combined with nonadaptivity [BMRV00]. This data structure is shown to exist, but there is no known efficient way of computing how to perform the bit probes. If one desires the amount of computation for a query to be $w^{O(1)}$, the presently best space usage is $n \cdot 2^{(\log w)^{O(1)}}$ bits [TS].

Whereas there is an efficient nonadaptive membership data structure in the bit probe model, this is not true for the perfect hashing problem, as shown in Chapter 9. In particular, for nonadaptive schemes there is a large difference between the membership and dictionary problems. Chapter 9 also shows that this difference vanishes if just *one* adaptive bit probe is allowed.

### 1.3.4 How little computation?

The issue of how the word RAM instruction set influences the performance of various algorithms and data structures has received some attention, both from upper and lower bound perspectives. As mentioned in Section 1.2, the motivation for studying instruction sets weaker than the standard C-like instruction set is that the unit cost assumption is (more) questionable for instructions that do not have small circuit complexity. For the C-like instruction set the multiplication and division operations are of special concern, as they are the only functions not in the constant depth circuit class $\mathrm{AC}^0$. (One could argue that a more serious scalability concern is that small, constant depth circuits for retrieving bits from memory are not possible, but we do not go into this.)

Andersson et al. [AMRT96] have shown that a unit cost RAM that allows linear space membership data structures with constant query time must have an instruction of circuit depth $\Omega(\log w / \log \log w)$. Since this matches the circuit depth of multiplication, we see that membership queries in constant time are not possible with weaker instruction sets (in the circuit-depth sense). However, some work has been done on minimizing the query time in weaker models. For the $\mathrm{AC}^0$ instruction set, there is a tight bound of $\Theta(\sqrt{\log n / \log \log n})$ on the query time [AMRT96] (a simpler proof of the upper bound, and better bounds in special cases, appear in [Hag98a]). The algorithm of the upper bound uses nonstandard $\mathrm{AC}^0$ instructions. For a C-like instruction set without multiplication and division, the best upper bound is $\sqrt{\log n (\log \log n)^{1+o(1)}}$, due to Brodnik et al. [BMM97].

A few other instruction sets have been studied from a lower bound perspective. Fich and Miltersen [FM95] studied a RAM with standard unit cost arithmetic operations but without division and bit operations, and showed that membership query time $o(\log n)$ requires $\Omega(2^w/n^\epsilon)$ words of memory for any

constant $\epsilon > 0$. Miltersen [Mil96] showed that to achieve constant query time on a RAM with bit operations but without multiplication and division, one needs $2^{\epsilon w}$ words, for some $\epsilon > 0$, when $n = 2^{o(w)}$. This matches the performance of a trie.

All word RAM algorithms in this thesis use the C-like instruction set, in some cases extended by a special instruction. In one case, namely Chapter 3 which describes a construction of perfect hash functions, special attention is paid to the number of "expensive" operations, i.e., multiplications. The lower bound of [AMRT96] means that any reasonably space efficient perfect hash function that can be evaluated in constant time must use multiplication. The FKS perfect hash function uses two multiplications (and originally also division, but this can be avoided [Df96]). It is shown in Chapter 3 that a single multiplication suffices for constant time perfect hashing.

### 1.3.5 How little space?

Space efficient versions of many data structures have appeared in recent literature. The goal is to approach the *information theoretical minimum* space needed to represent the data that is stored, while supporting efficient queries, and possibly also efficient dynamic updates. The space complexity is expressed in terms of certain parameters of the problem, in our case the sizes of the set and of the universe. For example, the number of sets of size $n$ in a universe of size $m$ is $\binom{m}{n}$. Hence, in order to have a membership data structure for each such set, at least $B = \lceil \log_2 \binom{m}{n} \rceil$ bits are needed for the data structure. The information theoretical minimum is, in other words, the smallest number of bits we could hope to use, if we want to be able to handle every set of a certain size. Here we survey space efficient data structures for membership and perfect hashing.

**Membership.** The information theoretical minimum for membership is roughly $m \log_2(em/n)$ bits. For $n = \Omega(m)$, a bit vector using $m$ bits is within a constant factor from the minimum. The FKS membership data structure uses $O(n \log m)$ bits, which is not $O(B)$ when $n = m^{1-o(1)}$. Brodnik and Munro have shown how to construct a static membership data structure with constant lookup time using $B + o(B)$ bits [BM99]. In Chapter 5 we describe a static dictionary with space usage that comes even closer to $B$, namely $B + o(n) + O(\log \log m)$ bits. The main idea behind the improvement is a generally applicable technique called *quotienting* that "compresses" each entry in a hash table to fewer bits. (After writing the paper in Chapter 5, I found out that a similar technique is described in [Knu98, Exercise 6.4.13] – however, its implications for space efficient membership data structures are not treated.) Quotienting has later been used in dictionaries supporting rank queries [RR99]. In [RRR02] these results are improved without the use of quotienting. Additionally, it is shown that in the cell probe model there is a membership data structure of $n$ words of $\lceil \log m \rceil$ bits supporting constant time queries. This follows by the above results only when $m$ is not too large.

Brodnik and Munro [BM99] also describe a dynamic version of their membership data structure. It uses $O(B)$ bits and supports updates in amortized expected constant time.

**Perfect hashing.** Since perfect hashing is mostly used in conjunction with a hash table that takes up $O(n)$ words of memory, one could argue that using the same space for a perfect hash function is quite reasonable. However, there are situations in which storage for the perfect hash function is more expensive than that for the hash table (if the hash table is in external memory, say). In this case we would like the smallest possible representation of the perfect hash function which allows efficient evaluation.

Mehlhorn has shown that $\Theta(n + \log\log m)$ bits are needed to represent perfect hash functions with linear range [Meh84]. Fredman and Komlós [FK84] tightened the bound, and a simple proof of essentially their bound has since been given [Rad92]. Schmidt and Siegel utilized compact encoding techniques to compress (essentially) the FKS perfect hash function to the optimal $O(n + \log\log m)$ bits [SS90b]. In fact, their perfect hash function is *minimal*, that is, the range is $\{0, \ldots, n-1\}$. The evaluation time is constant, but large, and the constant factor in the space usage is large, so the scheme is mainly of theoretical interest. The space usage was improved to essentially $1 + o(1)$ times optimal by Hagerup and Tholey [HT01]. They also describe a construction algorithm running in expected time $O(n + \log\log m)$.

In Chapter 8 we consider a weaker form of perfect hashing, where the requirement is that a large subset of any set can be mapped injectively by some function in the family. It is shown that in order to perform significantly better than a random function, a family of functions of size comparable to perfect families is needed. In other words, storing a "near-perfect" hash function is nearly as expensive as storing a perfect one.

Another relaxation of perfect hashing has been considered by Schmidt and Siegel [SS90b]. On input $x$, an "oblivious $t$-probe hash function" for the set $S$ computes a *set* of $t$ values, one of which (for appropriate arrangement of $S$ in a table) is the location of $x$, if $x \in S$. It was shown that $n/2^{O(t)}$ bits are needed to represent such a function, in the case $r = n$. However, it is mentioned that there is a probabilistic argument showing $O(\log n + \log\log m)$ bits to suffice when $r = (1 + \Omega(1))\, n$, for some constant $t$. Prior to the work in this thesis it seems that it has not been studied how to implement oblivious $t$-probe hash functions efficiently. Chapter 2 presents a dynamic oblivious 2-probe hashing scheme with asymptotic performance equaling that of the dynamic FKS scheme. It builds on a static data structure contained in Chapter 9. In both data structures the space used for the hash function is $O(n^\epsilon + \log\log m)$ bits, where $\epsilon > 0$ can be any constant.

### 1.3.6 How little randomness?

The success of randomized algorithms has raised some fundamental questions on the role of randomness in computing. To what extent are randomized algorithms superior to deterministic ones? When is it possible to *derandomize* an

algorithm, removing (or reducing) the use of random bits without sacrificing performance?

**Using few random bits.** The FKS perfect hash function construction algorithm requires $\Theta(n \log n + \log w)$ random bits. The amortized expected number of random bits used per operation in the dynamic version is $n$ times smaller. The presently lowest use of random bits for a dynamic trans-dichotomous membership data structure with expected constant time operations is $O(\log n + \log w)$ bits, by a result of Dietzfelbinger et al. [DfGMP92]. This number of random bits is the minimum possible if one wants to use a random hash function from a universal family, as shown by a lower bound of Mehlhorn [Meh84].

In Chapter 8 we study certain hash function families that are considerably smaller than universal families, and investigate their applicability in hashing algorithms and data structures. Such *dispersing* families do not seem to have been studied directly before, though the dispersion property of universal families has been used several times. We show a close relationship between dispersing families and *extractors*, an important tool in derandomization. Since no explicit construction of extractors with optimal parameters is known, there is also no known explicit construction of a dispersing family of (close to) minimum size. If one assumes dispersing families of hash functions to be built into the instruction set of a unit cost RAM, randomness efficient algorithms for several problems are possible. More specifically, using $O(\log(w/\log n))$ random bits, expected, and space $O(n)$ one can:

- Perform relational join in expected time $O(n)$.
- Solve the element distinctness problem in expected time $O(n)$.
- Construct a static dictionary with constant lookup time and linear space in time $O(n \log^\epsilon n)$, for any constant $\epsilon > 0$.

**Using no random bits.** Efficient deterministic membership data structures seem harder to come by than randomized ones. Randomness comes into play in the FKS scheme when we want to choose a "good" universal hash function for a set. There is an abundance of such good functions, so randomly trying different functions is an efficient strategy. The strategy in the deterministic FKS construction algorithm, on the other hand, is a brute force search. Using the *method of conditional probabilities*, Raman [Ram96] showed how to deterministically find a good universal hash function "bit by bit" in total time $O(n^2 w)$, which also becomes the total time for constructing the static FKS data structure. Alon and Naor [AN96] used another derandomization tool, *small bias probability spaces* [AGHP92, NN93], to derandomize a variant of the FKS scheme, achieving construction time $O(nw \log^4 n)$. However, a query requires evaluation of a linear function in time $\Theta(w/\log n)$, so the query time is only constant for $w = O(\log n)$. In [Mil98], Miltersen showed how error-correcting codes can be used to construct a linear space membership data structure with constant query time in $O(n^{1+\epsilon})$ time, for arbitrary constant $\epsilon > 0$. His implementation of error-correcting codes using a standard instruction set is weakly

nonuniform. Combining Miltersen's approach with the use of word parallelism, Hagerup [Hag99] achieved $O(\log\log n)$ query time and $O(n \log n)$ construction time. The main theorem in Chapter 6 is that constant query time and $O(n \log n)$ construction time can be achieved simultaneously. This was shown in [Pag00a], building on the results of Hagerup and Miltersen, which are also described in Chapter 6. The new ingredient is an improved construction algorithm for a class of perfect hash functions introduced by Tarjan and Yao [TY79].

For the dynamic case, the static result of Chapter 6 implies the following tradeoff between the time for updates and queries: Queries in time $O(t)$, insertions in time $O(n^{1/t})$, and deletions in time $O(\log n)$, for parameter $t$ smaller than $\sqrt{\log n}$. For query time $\Theta(\sqrt{\log n / \log\log n})$, the search tree data structure described in Section 1.3.1 [AT00, BF99a] supplies the fastest known deterministic update operations: $O(\sqrt{\log n / \log\log n})$ time per update. When the word length is small, this data structure supports much faster operations, e.g., for $w = \log^{O(1)} n$ the time per operation is $o((\log\log n)^2)$. This is used in a dynamic dictionary described in Chapter 7, that uses Miltersen's error-correcting code technique to reduce the universe considered to $\{0,1\}^{O(\log^2 n)}$, and then applies the data structure of [AT00, BF99a]. (Actually, the reduction depends heavily on the fact that this data structure supports predecessor queries and not just membership queries.) The result is a deterministic dynamic dictionary using $o((\log\log n)^2)$ time for queries, and time $O(\log^2 n)$ for insertions and deletions. The time bounds hold in the worst case. (In fact, the time for deletions can be improved to $o((\log\log n)^2)$ by using deletion markers and periodic global rebuilding.)

An unpublished manuscript by Sundar [Sun93] states an amortized lower bound of time $\Omega(\frac{\log\log_w n}{\log\log\log_w n})$ per operation for polynomial space deterministic membership data structures in the cell probe model, for $w = \Omega(\log n \log\log n)$. In particular, this implies the same lower bound in RAM models. Note that for $w = (\log n)^{O(1)}$, the data structure of [AT00, BF99a] has time per operation polynomially related to the lower bound.

### 1.3.7 What kind of performance guarantee?

There are several types of guarantees that can be made on the performance of algorithms. First of all, bounds may hold in the amortized sense or for every operation. Bounds for randomized algorithms may hold in the expected sense or hold with high probability. Also, bounds may hold only under some assumption on the input. For example, as mentioned in Section 1.3.1, many dynamic hashing schemes have been analyzed under the assumption that inserted keys are random and independent. In this section we survey some improvements on performance guarantees compared to the "basic" algorithms in Section 1.3.1.

Dietzfelbinger and Meyer auf der Heide [DfMadH90] have constructed a dictionary in which, for any sequence of $n^{O(1)}$ operations, the time for every operation is constant with high probability (i.e., probability at least $1 - n^{-c}$, where $c$ is any constant of our choice). A simpler dictionary with the same properties, and using far fewer random bits, was later developed [DfGMP92]. Willard [Wil00] claims to achieve better error probability, but this is probably

incorrect. He uses a random construction of a hash function family due to Siegel [Sie89], and seems to have overlooked the probability that it fails to have large independence. For static dictionaries, Bast and Hagerup [BH91] achieve $O(n)$ construction time with probability $1 - 2^{-n^\epsilon}$, for some constant $\epsilon > 0$.

In Chapter 10 we consider a dynamic dictionary in a special unit cost RAM model augmented with an instruction that can compute a given edge of an expander graph. It offers better performance than previous dictionaries when the number of lookups is $w^{1+\Omega(1)}$ times larger than the number of updates. More specifically, for a sequence of $a$ updates and $b$ lookups in a key set of size at most $n$, the time used is $O(aw^{1+o(1)}+b+t)$ with probability $1-2^{-\Omega(a+t/w^{1+o(1)})}$. A drawback of the dictionary is its space usage of $O(nw)$ words.

In Chapter 4 we show how to remove the assumption of random inputs for a very wide class of hashing schemes analyzed under the assumption of truly random hashing. More precisely we show how to simulate, with high probability, the behavior of a truly random function on any set $S$ of size $n$. Hash functions from our family can be represented in $O(n)$ words and evaluated in constant time. This means that many hashing schemes can be implemented to perform, with high probability, *exactly* as in the random oracle model. To some extent this justifies analyses done under the assumption of truly random hash functions. However, the hash function family proposed is not likely to be of practical value. This is both because of a large constant in the evaluation time, and because $O(n)$ words of memory (which is necessary to achieve random behavior on a set of size $n$) is too much in many applications.

## 1.4 A brief history of the thesis

When I started my PhD studies in February 1998, my advisor Peter Bro Miltersen pointed me to a number of papers that would be good to read, and whose results I might try to improve. I was bedazzled by the dictionary of Fredman et al. [FKS84], so I thought it was interesting when Peter suggested that I took a look at the paper of Brodnik and Munro [BM94] on a space efficient membership data structure. In the early summer of 1998 I made the first improvement in redundancy compared to [BM94], and after a number of refinements I ended up with the contents of Chapter 5.

Meanwhile, I had discovered an old and somewhat overlooked paper of Tarjan and Yao on perfect hashing [TY79]. Just before Christmas in 1998 it became clear that the Tarjan-Yao construction could be combined with universal hashing to achieve an efficient perfect hashing scheme. This was the start of Chapter 3.

The Tarjan-Yao hash functions also became the basis for a new deterministic construction of perfect hash functions for small universes in the spring of 1999. Together with results of Miltersen [Mil98] and Hagerup [Hag99] this led to the result described in Chapter 6.

In the fall of 1999 the foundations for Chapter 7 were laid, extending ideas of Chapter 6 to the dynamic case. The realization that this might be possible was probably due to a new result of Beame and Fich [BF99a] that implied very

efficient dictionaries for small universes. At the same time I became interested in probabilistic constructions, and began looking at dispersing families of functions, as described in Chapter 8. The motivation was in various applications of such families in algorithms using few random bits.

These five pieces of work formed my qualifying exam progress report. I passed the qualifying exam in December 1999.

The spring of 2000 was used for finishing papers, and for finding a good place for my stay abroad. My inquiry with Rajeev Motwani at Stanford University led to an invitation to go there for the fall semester.

Back in 1998 Peter had raised the question of whether the three memory probes used by the FKS dictionary was the minimum possible. Shortly before going to Stanford I found a proof that this was indeed the case, and this led to a study of cell probe complexity of membership and perfect hashing, as described in Chapter 9.

Also at Stanford, *cuckoo hashing* emerged as a dynamization of one of the data structures in Chapter 9. When returning to Århus in January 2001, I joined forces with my officemate Flemming Rodler in investigating the practicality of cuckoo hashing. Our findings are described in Chapter 2.

Between May and November 2001 I was on leave from my PhD studies to do "civilian military service".

The formal analysis of cuckoo hashing uses a family of hash functions by Siegel [Sie89]. An important ingredient in his construction are certain expander graphs. Similar expander graphs had been used for a new membership data structure of Buhrman et al. [BMRV00]. Together with Anna Östlin who had joined BRICS in the summer of 2001, I began looking at expander graphs and ways of improving Siegel's construction. This led to two papers in the period November 2001 to April 2002. The first one, found in Chapter 10, is a follow-up to the paper of Buhrman et al, showing how to achieve a stronger performance guarantee using more space. The second one forms Chapter 4 and is, in a certain sense, an improvement of Siegel's hash functions to optimal space.

Finally, the thesis was handed in after nearly four years of studies in June 2002.

## 1.5 Publications in the thesis

The rest of the thesis consists of (revised versions of) the below publications.

**Chapter 2.** Rasmus Pagh and Flemming Friche Rodler, *Cuckoo Hashing*, Proceedings of ESA 2001, Lecture Notes in Computer Science 2161, p. 121-133.

**Chapter 3.** Rasmus Pagh, *Hash and Displace: Efficient Evaluation of Minimal Perfect Hash Functions*, Proceedings of WADS 1999, Lecture Notes in Computer Science 1663, p. 49-54.

**Chapter 4.** Anna Östlin and Rasmus Pagh, *Simulating Uniform Hashing in Constant Time and Optimal Space*, BRICS technical report RS-02-27, Aarhus University, 2002.

**Chapter 5.** Rasmus Pagh, *Low Redundancy in Static Dictionaries with Constant Query Time*, SIAM Journal on Computing 31 (2001), p. 353-363. A preliminary version appeared at ICALP 1999.

**Chapter 6.** Torben Hagerup, Peter Bro Miltersen, and Rasmus Pagh, *Deterministic Dictionaries*, Journal of Algorithms 41 (2001), p. 69-85. A preliminary version of my contribution to this paper appeared at SODA 2000.

**Chapter 7.** Rasmus Pagh, *A Trade-Off for Worst-Case Efficient Dictionaries*, Nordic Journal of Computing 7 (2000), p. 151-163. A preliminary version of this paper appeared at SWAT 1999.

**Chapter 8.** Rasmus Pagh, *Dispersing Hash Functions*, Proceedings of RANDOM 2000, Proceedings in Informatics 8, p. 53-67.

**Chapter 9.** Rasmus Pagh, *On the Cell Probe Complexity of Membership and Perfect Hashing*, Proceedings of STOC 2001, p. 425-432.

**Chapter 10.** Anna Östlin and Rasmus Pagh, *One-Probe Search*, To appear in Proceedings of ICALP 2002, Lecture Notes in Computer Science.

# Chapter 2

## Cuckoo hashing

The *dictionary* data structure is ubiquitous in computer science. A dictionary is used for maintaining a set $S$ under insertion and deletion of elements (referred to as *keys*) from a universe $U$. Membership queries ("$x \in S$?") provide access to the data. In case of a positive answer the dictionary also provides a piece of *satellite data* that was associated with $x$ when it was inserted.

A large theory, partially surveyed in Section 2.1, is devoted to dictionaries. It is common to study the case where keys are bit strings in $U = \{0,1\}^w$ and $w$ is the word length of the computer (for theoretical purposes modeled as a RAM). Section 2.2 discusses this restriction. It is usually, though not always, clear how to return associated information once membership has been determined. E.g., in all methods discussed in this chapter, the associated information of $x \in S$ can be stored together with $x$ in a hash table. Therefore we disregard the time and space used to handle associated information and concentrate on the problem of maintaining $S$. In the following we let $n$ denote $|S|$.

The most efficient dictionaries, in theory and in practice, are based on hashing techniques. The main performance parameters are of course lookup time, update time, and space. In theory there is no trade-off between these: One can simultaneously achieve constant lookup time, expected amortized constant update time, and space within a constant factor of the information theoretical minimum of $B = \log \binom{|U|}{n}$ bits [BM99]. In practice, however, the various constant factors are crucial for many applications. In particular, lookup time is a critical parameter. It is well known that one can achieve performance arbitrarily close to optimal if a sufficiently sparse hash table is used. Therefore the challenge is to combine speed with a reasonable space usage. In particular, we only consider schemes using $O(n)$ words of space.

The contribution of this chapter is a new, simple hashing scheme called Cuckoo Hashing. A description and analysis of the scheme is given in Section 2.3, showing that it possesses the same theoretical properties as the dynamic dictionary of Dietzfelbinger et al. [DfKM$^+$94]. That is, it has worst case constant lookup time and amortized expected constant time for updates. A special feature of the lookup procedure is that (disregarding accesses to a small hash function description) there are just two memory accesses, which are *independent* and can be done in parallel if this is supported by the hardware. Our scheme works for space similar to that of binary search trees, i.e., three words

per key in $S$ on average.

Using weaker hash functions than those required for our analysis, CUCKOO HASHING is very simple to implement. Section 2.4 describes such an implementation, and reports on extensive experiments and comparisons with the most commonly used methods, having no nontrivial worst case guarantee on lookup time. It seems that an experiment comparing the most commonly used methods on a modern multi-level memory architecture has not previously been described in the literature. Our experiments show CUCKOO HASHING to be quite competitive, especially when the dictionary is small enough to fit in cache. We thus believe it to be attractive in practice, when a worst case guarantee on lookups is desired. The software library LEDA [MN99], version 4.3, incorporates CUCKOO HASHING.

## 2.1 Previous work on linear space dictionaries

Hashing, first described in public literature by Dumey [Dum56], emerged in the 1950s as a space efficient heuristic for fast retrieval of information in sparse tables. Knuth surveys the most important classical hashing methods in [Knu98, Section 6.4]. The most prominent, and the basis for our experiments in Section 2.4, are CHAINED HASHING (with separate chaining), LINEAR PROBING and DOUBLE HASHING. Judging from leading textbooks on algorithms, Knuth's selection of algorithms is in agreement with current practice for implementation of general purpose dictionaries. In particular, the excellent cache usage of LINEAR PROBING makes it a prime choice on modern architectures. A more recent scheme called TWO-WAY CHAINING [ABKU99] will also be investigated. All schemes are briefly described in Section 2.4.

### 2.1.1 Analysis of early hashing schemes

Early theoretical analysis of hashing schemes was done under the assumption that hash function values are uniformly random and independent. Precise analyses of the average and expected worst case behaviors of the abovementioned schemes have been made, see for example [Gon84, Knu98]. We mention just a few facts, disregarding asymptotically vanishing terms. Note that some figures depend on implementation details – the below hold for the implementations described in Section 2.4.

We first consider the expected number of memory probes needed by the two "open addressing" schemes to insert a key in a hash table where an $\alpha$ fraction of the table, $0 < \alpha < 1$, is occupied by keys. For LINEAR PROBING the expected number of probes during insertion is $\frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$. This coincides with the expected number of probes for unsuccessful lookups, and with the number of probes needed for looking up the key if there are no subsequent deletions. A deletion rearranges keys to the configuration that would occur if the deleted key had never been inserted. In DOUBLE HASHING the expected cost of an insertion is $\frac{1}{1-\alpha}$. As keys are never moved, this coincides with the number of probes needed for looking up the key and for deleting the key. If a key has not been inserted in the hash table since the last rehash, the expected cost

of looking it up (unsuccessfully) is $\frac{1}{1-\beta}$, where $\beta$ is the fraction of keys and "deleted" markers in the hash table. If the key still has a "deleted" marker in the table, the expected cost of the unsuccessful lookup is one probe more.

For Chained Hashing with hash table size $n/\alpha$, the expected length of the list traversed during an unsuccessful lookup is $\alpha$. This means that the expected number of probes needed to insert a new key is $1 + \alpha$, which will also be the number of probes needed to subsequently look up the key (note that probes to pointers are not counted). A deletion results in the data structure that would occur if the key had never been inserted.

In terms of number of *probes*, the above implies that, for any given $\alpha$, Chained Hashing is better than Double Hashing, which is again better than Linear Probing. It should be noted, however, that the space used by Chained Hashing is larger than that in the open addressing schemes for the same $\alpha$. The difference depends on the relative sizes of keys and pointers.

The *longest* probe sequence in Linear Probing is of expected length $\Omega(\log n)$. For Double Hashing the longest successful probe sequence is expected to be of length $\Omega(\log n)$, and there is in general no sublinear bound on the length of unsuccessful searches. The expected maximum chain length in Chained Hashing is $\Theta(\log n / \log \log n)$.

Though the above results seem to agree with practice, the randomness assumptions used for the analyses are questionable in applications. Carter and Wegman [CW79] succeeded in removing such assumptions from the analysis of Chained Hashing, introducing the concept of *universal* hash function families. When implemented with a random function from Carter and Wegman's universal family, chained hashing has constant expected time per dictionary operation (plus an amortized expected constant cost for resizing the table). For a certain efficient hash function family of Siegel [Sie89], Linear Probing and Double Hashing provably satisfy the above performance bounds [SS89, SS90a]. Siegel's hash functions, summarized in Theorem 2.1, are also used in Cuckoo Hashing.

### 2.1.2 Key rearrangement schemes

A number of (open addressing) hashing schemes have been proposed that share a key feature with the scheme described in this chapter, namely that keys are moved around during insertions [Bre73, GM79, Mad80, Mal77, Riv78]. The main focus in these schemes is to reduce the average number of probes needed for finding a key in a (nearly) full table to a constant, rather than the $O(\log n)$ average exhibited by standard open addressing. This is done by occasionally moving keys forward in their probe sequences.

In our algorithm we rearrange keys in order to reduce the *worst case* number of probes to a constant. A necessary condition for this is reuse of hash function values, i.e., that keys are moved back in their probe sequence. Backward moves were not used in any previous rearrangement scheme, presumably due to the difficulty that moving keys back does not give a fresh, "random" placement. The thing that allows us to obtain worst case efficient lookups is that we do not deal with full hash tables, but rather hash tables that are less than half full.

It is shown in Chapter 9 that in this case there exists, with high probability, an arrangement that allows any key to be found in two hash table probes. We show how to efficiently maintain such an arrangement under updates of the key set.

Arrangements of keys with optimal worst case retrieval cost were in fact already considered by Rivest in [Riv78], where a polynomial time algorithm for finding such an arrangement was given. Also, it was shown that if one updates the key set, the expected number of keys that need to be moved to achieve a new optimal arrangement is constant. (The analysis requires that the hash table is sufficiently sparse, and assumes the hash function to be truly random.) This suggests a dictionary that solves a small assignment problem after each insertion and deletion. It follows from Chapter 9 and this chapter, that Rivest's dictionary achieved worst case constant lookup time and expected amortized constant update time, 8 years before an algorithm with the same performance and randomness assumption was published by Aho and Lee [AL86]. Further, we show that Siegel's hash functions suffice for the analysis. Last but not least, the algorithm we use for rearranging keys is much simpler and more efficient than that suggested by Rivest.

Another key rearrangement scheme with similarities to Cuckoo Hashing is Last-come-first-served Hashing [PM89], which has low variance on search time as its key feature. It uses the same greedy strategy for moving keys as is used in Cuckoo Hashing, but there is no reuse of hash function values.

### 2.1.3 Hashing schemes with worst case lookup guarantee

Two-Way Chaining is an alternative to Chained Hashing that offers maximal lookup time $O(\log \log n)$ with high probability (assuming truly random hash functions). The implementation that we consider represents the lists by fixed size arrays of size $O(\log \log n)$ (if a longer chain is needed, a rehash is performed). To achieve linear space usage, one must then use a hash table of size $O(n/\log \log n)$, implying that the *average* chain length is $\Omega(\log \log n)$.

Another scheme with this worst case guarantee is *Multilevel Adaptive Hashing* [BK00]. However, lookups can be performed in $O(1)$ worst case time if $O(\log \log n)$ hash function evaluations, memory probes and comparisons are possible in parallel. This is similar to the scheme described in this chapter, though we use only *two* hash function evaluations, memory probes and comparisons.

A dictionary with worst case *constant* lookup time was first obtained by Fredman, Komlós and Szemerédi [FKS84], though it was *static*, i.e., did not support updates. It was later augmented with insertions and deletions in amortized expected constant time by Dietzfelbinger et al. [DfKM$^+$94]. Dietzfelbinger and Meyer auf der Heide [DfMadH90] improved the update performance by exhibiting a dictionary in which operations are done in constant time with high probability, namely at least $1 - n^{-c}$, where $c$ is any constant of our choice. A simpler dictionary with the same properties was later developed [DfGMP92]. When $n = |U|^{1-o(1)}$ a space usage of $O(n)$ words is not within a constant factor of the information theoretical minimum. The dictionary of Brodnik and

Munro [BM99] offers the same performance as [DfKM$^+$94], using $O(B)$ bits in all cases.

## 2.2 Preliminaries

We assume that keys from $U$ fit exactly in a single machine word, that is, $U = \{0,1\}^w$. A special value $\perp \in U$ is reserved to signal an empty cell in hash tables. For DOUBLE HASHING an additional special value is used to indicate a deleted key.

Our algorithm uses hash functions from a *universal* family.

**Definition 2.1** *A family $\{h_i\}_{i\in I}$, $h_i : U \to R$, is* $(c,k)$-universal *if, for any $k$ distinct elements $x_1,\ldots,x_k \in U$, any $y_1,\ldots,y_k \in R$, and uniformly random $i \in I$, $\Pr[h_i(x_1) = y_1,\ldots,h_i(x_k) = y_k] \le c/|R|^k$.*

As an example, the family of all functions is $(1,|U|)$-universal. However, for implementation purposes one needs families with much more succinct memory representations. A standard construction of a $(2,k)$-universal family for range $R = \{0,\ldots,r-1\}$ and prime $p > \max(2^w, r)$ is

$$\{x \mapsto ((\sum_{l=0}^{k-1} a_l x^l) \bmod p) \bmod r \quad | \quad 0 \le a_0, a_1, \ldots, a_{k-1} < p\} \ . \tag{2.1}$$

If $U$ is not too large compared to $k$, there exists a space-efficient $(2,k)$-universal family due to Siegel [Sie89] that has *constant* evaluation time (however, the constant is not a small one).

**Theorem 2.1 (Siegel)** *There is a constant $c$ such that, for $k = 2^{\Omega(w)}$, there exists a $(2,k)$-universal family that uses space and initialization time $O(k^c)$, and which can be evaluated in* constant *time.*

Our restriction that keys are single words is not a serious one. Longer keys can be mapped to keys of $O(1)$ words by applying a random function from a $(O(1),2)$-universal family. There is such a family whose functions can be evaluated in time linear in the number of input words [CW79]. It works by evaluating a function from a $(O(1),2)$-universal family on each word, computing the bitwise exclusive or of the function values. (See [Tho00] for an efficient implementation). Such a function with range $\{0,1\}^{2\log(n)+c}$ will, with probability $1-O(2^{-c})$, be injective on $S$. In fact, with constant probability the function is injective on a given *sequence* of $\Omega(2^{c/2}n)$ consecutive sets in a dictionary of initial size $n$ (see [DfKM$^+$94]). When a collision between two elements of $S$ occurs, everything is rehashed. If a rehash can be done in expected $O(n)$ time, the amortized expected cost of this is $O(2^{-c/2})$ per insertion. In this way we can effectively reduce the universe size to $O(n^2)$, though the full keys still need to be stored to decide membership. When $c = O(\log n)$ the reduced keys are of length $O(\log n)$. For any $\epsilon > 0$, Theorem 2.1 then provides a family of constant time evaluable $(2, n^{\Omega(1)})$-universal hash functions, whose functions can be stored using space $O(n^\epsilon)$.

## 2.3 Cuckoo hashing

CUCKOO HASHING is a dynamization of a static dictionary described in Chapter 9. The dictionary uses two hash tables, $T_1$ and $T_2$, each of length $r$, and two hash functions $h_1, h_2 : U \to \{0, \ldots, r-1\}$. Every key $x \in S$ is stored in cell $h_1(x)$ of $T_1$ or $h_2(x)$ of $T_2$, but never in both. Our lookup function is

```
function lookup(x)
  return T1[h1(x)] = x ∨ T2[h2(x)] = x
end
```

Two table accesses for lookup is in fact optimal among all dictionaries using linear space, except for special cases, see Chapter 9.

**Remark:** The idea of storing keys in one out of two places given by hash functions previously appeared in [KLMadH96] in the context of PRAM simulation, and in [ABKU99] for TWO-WAY CHAINING.

It is shown in Chapter 9 that if $r \geq (1+\epsilon)\, n$ for some constant $\epsilon > 0$ (i.e., the tables are to be a bit less than half full), and $h_1$, $h_2$ are picked uniformly at random from an $(O(1), O(\log n))$-universal family, the probability that there is no way of arranging the keys of $S$ according to $h_1$ and $h_2$ is $O(1/n)$. By the discussion in Section 2.2 we may assume without loss of generality that there is such a family, with constant evaluation time and negligible space usage. A suitable arrangement of the keys is shown in Chapter 9 to be computable in expected linear time, by a reduction to 2-SAT.

We now consider a simple dynamization of the above. Deletion is of course simple to perform in constant time, not counting the possible cost of shrinking the tables if they are becoming too sparse. As for insertion, it turns out that the "cuckoo approach", kicking other keys away until every key has its own "nest", works very well. Specifically, if $x$ is to be inserted we first see if cell $h_1(x)$ of $T_1$ is occupied. If not, we are done. Otherwise we set $T_1[h_1(x)] \leftarrow x$ anyway, thus making the previous occupant "nestless". This key is then inserted in $T_2$ in the same way, and so forth iteratively, see Figure 2.1(a). It may happen that this process loops, as shown in Figure 2.1(b). Therefore the number of iterations is bounded by a value "MaxLoop" to be specified in Section 2.3.1. If this number of iterations is reached, everything is rehashed with new hash functions, and we try once again to accommodate the nestless key.

Using the notation $x \leftrightarrow y$ to express that the values of variables $x$ and $y$ are swapped, the following code summarizes the insertion procedure.

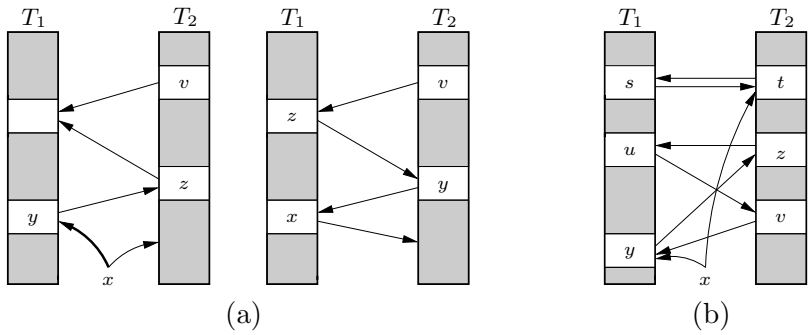


Figure 2.1: Examples of Cuckoo Hashing insertion. Arrows show possibilities for moving keys. (a) Key $x$ is successfully inserted by moving keys $y$ and $z$ from one table to the other. (b) Key $x$ cannot be accommodated and a rehash is necessary.

**procedure** insert($x$)
  **if** lookup($x$) **then return**
  **loop** MaxLoop **times**
    **if** $T_1[h_1(x)] = \perp$ **then** { $T_1[h_1(x)] \leftarrow x$; **return** }
    $x \leftrightarrow T_1[h_1(x)]$
    **if** $T_2[h_2(x)] = \perp$ **then** { $T_2[h_2(x)] \leftarrow x$; **return** }
    $x \leftrightarrow T_2[h_2(x)]$
  **end loop**
  rehash(); insert($x$)
**end**

The above procedure assumes that each table remains larger than $(1+\epsilon)\,n$ cells. When no such bound is known, a test must be done to find out when a rehash to larger tables is needed. Resizing of tables can be done in amortized expected constant time per update by the usual doubling/halving technique (see, e.g., [DfKM$^+$94]). The hash functions used will be $(O(1), \mathrm{MaxLoop})$-universal, which means that they will act almost like truly random functions on any set of keys processed during the insertion loop.

The lookup call preceding the insertion loop ensures robustness if the key to be inserted is already in the dictionary. A slightly faster implementation can be obtained if this is known not to occur.

Note that the insertion procedure is biased towards inserting keys in $T_1$. As will be seen in Section 2.4 this leads to faster successful lookups, due to more keys being found in $T_1$. This effect is even more pronounced if one uses an *asymmetric* scheme where $T_1$ is larger than $T_2$. In both cases, the insertion time is only slightly worse than that of a completely symmetric implementation. Another variant is to use a single table for both hash functions, but this requires keeping track of the hash function according to which each key is placed. In the following we consider just the symmetric two table scheme.

### 2.3.1 Analysis

Our analysis of the insertion procedure has three main parts:

1. We first exhibit some useful characteristics of the behavior of the insertion procedure.

2. We then derive a bound on the probability that the insertion procedure uses at least $t$ iterations.

3. Finally we argue that the procedure uses expected amortized constant time.

#### Behavior of the insertion procedure

The simplest behavior of the insertion procedure occurs when it does not visit any hash table cell more than once. In this case it simply runs through a sequence $x_1, x_2, \ldots$, of nestless keys with no repetitions, moving each key from one table to the other.

If, at some point, the insertion procedure returns to a previously visited cell, the behavior is more complicated, as shown in Figure 2.2. The key $x_i$ in the first previously visited cell will become nestless for the second time (occurring at positions $i$ and $j > i$ in the sequence) and be put back in its original cell. Subsequently all keys $x_{i-1}, \ldots, x_1$ will be moved back where they were at the start of the insertion (assuming that the maximum number of iterations is not reached). In particular, $x_1$ will end up nestless again, and the procedure will try placing it in the second table. At some point after this there appears a nestless key $x_l$ that is either moved to a vacant cell or a previously visited cell (again assuming that the maximum number of iterations is not reached). In the former case the procedure terminates. In the latter case a rehash must be performed, since we have a "closed loop" of $l - i + 1$ keys hashing to only $l - i$ cells. This means that the loop will run for the maximum number of iterations, followed by a rehash.

**Lemma 2.1** *Suppose that the insertion procedure does not enter a closed loop. Then for any prefix $x_1, x_2, \ldots, x_p$ of the sequence of nestless keys, there must be a subsequence of at least $p/3$ consecutive keys without repetitions, starting with an occurrence of the key $x_1$, i.e., the key being inserted.*

*Proof.* In the case where the insertion procedure never returns to a previously visited cell, the prefix itself is a sequence of $p$ distinct nestless keys starting with $x_1$. Otherwise, the sequence of nestless keys is as shown in Figure 2.2. If $p < i + j$, the first $j - 1 \geq \frac{i+j-1}{2} \geq p/2$ nestless keys form the desired sequence. For $p \geq i + j$, one of the sequences $x_1, \ldots, x_{j-1}$ and $x_{j+i-1}, \ldots, x_p$ must have length at least $p/3$. □

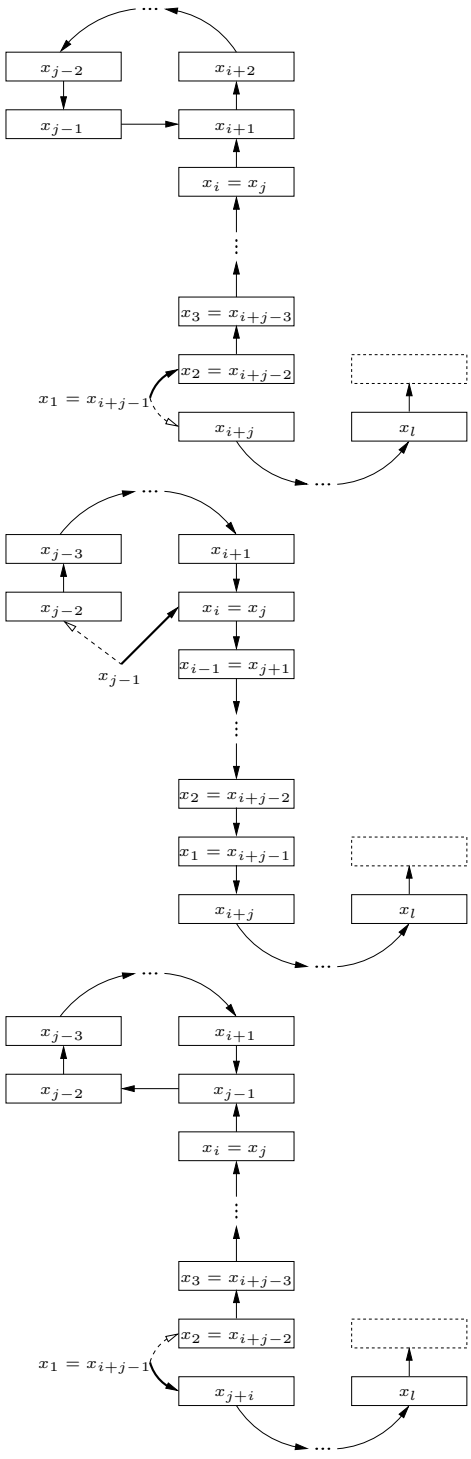


Figure 2.2: Stages of an insertion of key $x_1$, involving the movement of keys $x_1, \ldots, x_l$. Boxes correspond to cells in either of the two tables, and arcs show possibilities for moving keys. A bold arc shows where the nestless key is to be inserted.

### Probability bounds

We now consider the probability that the insertion loop runs for at least $t$ iterations. For $t >$ MaxLoop the probability is of course 0. Otherwise, by the above analysis, iteration number $t$ is performed in two (not mutually exclusive) situations:

1. The insertion procedure has entered a "closed loop", i.e., $x_l$ in Figure 2.2 was moved to a previously visited cell, for $l \leq 2t$.

2. The insertion procedure has processed a sequence of at least $(2t-1)/3$ consecutive nestless keys starting with the newly inserted key.

In the first situation let $v \leq l$ denote the number of distinct nestless keys. The number of ways in which the closed loop can be formed is less than $v^2 r^{v-1} n^{v-1}$ ($v^2$ possible values for $i$ and $j$, $r^{v-1}$ possible choices of cells, and $n^{v-1}$ possible choices of keys other than $x_1$). Since $v \leq$ MaxLoop, the hash functions are $(c, v)$-universal. This means that each possibility occurs with probability at most $c^2 r^{-2v}$. Summing over all possible values of $v$, and using $r/n > 1+\epsilon$, we get that the probability of situation 1 is at most:

$$\sum_{v=3}^{l} v^2 r^{v-1} n^{v-1} c^2 r^{-2v} \leq \frac{c^2}{rn} \sum_{v=3}^{\infty} v^2 (n/r)^v < \frac{13\, c^2/\epsilon}{n^2} = O(1/n^2) \ .$$

The above derivation follows a suggestion of Sanders and Vöcking [SV01], and improves the $O(1/n)$ bound in the conference paper corresponding to this chapter [PR01a].

In the second situation there is a sequence of distinct nestless keys $b_1, \ldots, b_v$, $v \geq (2t-1)/3$, such that $b_1$ is the key to be inserted, and such that for either $(\beta_1, \beta_2) = (1, 2)$ or $(\beta_1, \beta_2) = (2, 1)$:

$$h_{\beta_1}(b_1) = h_{\beta_1}(b_2),\ h_{\beta_2}(b_2) = h_{\beta_2}(b_3),\ h_{\beta_1}(b_3) = h_{\beta_1}(b_4), \ldots \tag{2.2}$$

Given $b_1$ there are at most $n^{v-1}$ possible sequences of $v$ distinct keys. For any such sequence and any of the two choices of $(\beta_1, \beta_2)$, the probability that the $b-1$ equations in (2.2) hold is bounded by $c\, r^{-(v-1)}$, since the hash functions were chosen from a $(c, \text{MaxLoop})$-universal family. Hence the probability that there is *any* sequence of length $v$ satisfying (2.2), and thus the probability of situation 2, is bounded by

$$2c^2 (n/r)^{v-1} \leq 2c^2 (1+\epsilon)^{-(2t-1)/3+1} \ . \tag{2.3}$$

### Concluding the analysis

From now on we restrict attention to MaxLoop $= O(n)$. From (2.3) it follows that the expected number of iterations in the insertion loop is bounded by

$$\begin{aligned} & 1 + \sum_{t=2}^{\text{MaxLoop}} 2c^2 \left(1+\epsilon\right)^{-(2t-1)/3+1} + O(1/n^2) \qquad (2.4) \\ & \leq 1 + O(\tfrac{\text{MaxLoop}}{n^2}) + 2c^2 \sum_{t=0}^{\infty} ((1+\epsilon)^{-2/3})^t \\ & = O(1 + \tfrac{1}{1-(1+\epsilon)^{-2/3}}) \\ & = O(1 + 1/\epsilon) \ . \end{aligned}$$

Finally, we consider the cost of rehashing, which occurs if the insertion loop runs for $t =$ MaxLoop iterations. By the previous section, the probability that this happens because of entering a closed loop is $O(1/n^2)$. Setting MaxLoop $= \lceil 3 \log_{1+\epsilon} n \rceil$, the probability of rehashing without entering a closed loop is, by (2.3), at most

$$2c^2 \left(1+\epsilon\right)^{-(2\,\text{MaxLoop}-1)/3+1} = O(1/n^2)$$

Altogether, the probability that any given insertion causes a rehash is $O(1/n^2)$. In particular, the $n$ insertions performed during a rehash all succeed (i.e., cause no further rehash) with probability $1 - O(1/n)$. The expected time used per insertion is $O(1)$, so the total expected time for trying to insert all keys is $O(n)$. As the probability of having to start over with new hash functions is bounded away from 1, the total expected time for a rehash is $O(n)$. Thus, for any insertion the expected time used for rehashing is $O(1/n)$.

Summing up, we have shown that the expected time for insertion is bounded by a constant. The small probability of rehashing in fact implies that also the *variance* of the insertion time is constant.

## 2.4 Experiments

To examine the practicality of Cuckoo Hashing we experimentally compare it to three well known hashing methods, as described in [Knu98, Section 6.4]: Chained Hashing (with separate chaining), Linear Probing and Double Hashing. We also consider Two-Way Chaining [ABKU99].

The first three methods all attempt to store a key $x$ at position $h(x)$ in a hash table. They differ in the way collisions are resolved, i.e., in what happens when two or more keys hash to the same location.

Chained Hashing. A chained list is used to store all keys hashing to a given location.

Linear Probing. A key is stored in the next empty table entry. Lookup of key $x$ is done by scanning the table beginning at $h(x)$ and ending when either $x$ or an empty table entry is found. When deleting, some keys may have to be moved back in order to fill the hole in the lookup sequence, see [Knu98, Algoritm R] for details.

Double Hashing. Insertion and lookup are similar to Linear Probing, but instead of searching for the next position one step at a time, a second hash function value is used to determine the step size. Deletions are handled by putting a "deleted" marker in the cell of the deleted key. Lookups skip over deleted cells, while insertions overwrite them.

The fourth method, Two-Way Chaining, can be described as two instances of Chained Hashing. A key is inserted in one of the two hash tables, namely the one where it hashes to the shortest chain. A cache-friendly implementation, as recently suggested in [BM01], is to simply make each chained list a short, fixed size array. If a longer list is needed, a rehash must be performed.

### 2.4.1 Previous experimental results.

Although the dictionaries with worst case constant lookup time surveyed in Sect. 2.1 leave little to improve from a theoretical point of view, large constant factors and complicated implementation hinder their direct practical use. For example, in the "dynamic perfect hashing" scheme of [DfKM$^+$94] the upper bound on space is $35n$ words. The authors of [DfKM$^+$94] refer to a more practical variant due to Wenzel that uses space comparable to that of binary search trees.

According to [KL96] the implementation of this variant in the LEDA library [MN99], described in [Wen92], has average insertion time larger than that of AVL trees for $n \leq 2^{17}$, and more than four times slower than insertions in chained hashing. (On a Linux PC with an Intel® Pentium® 120 MHz processor.) The experimental results listed in [MN99, Table 5.2] show a gap of more than a factor of 6 between the update performance of chained hashing and dynamic perfect hashing, and a factor of more than 2 for lookups. (On a 300 MHz SUN ULTRA SPARC.)

Silverstein [Sil98] reports that the space upper bound of the dynamic perfect hashing scheme of [DfKM$^+$94] is quite pessimistic compared to what can be observed when run on a subset of the DIMACS dictionary tests [McG]. He goes on to explore ways of improving space as well as time, improving both the observed time and space by a factor of roughly three. Still, the improved scheme needs 2 to 3 times more space than an implementation of linear probing to achieve similar time per operation. Silverstein also considers versions of the data structures with packed representations of the hash tables. In this setting the dynamic perfect hashing scheme was more than 50% slower than linear probing, using roughly the same amount of space.

Is seems that recent experimental work on "classical" dictionaries (that do not have worst case constant lookup time) is quite limited. In [KL96] it is reported that chained hashing is superior to an implementation of dynamic perfect hashing in terms of both memory usage and speed.

### 2.4.2 Data structure design and implementation

We consider positive 32 bit signed integer keys and use 0 as $\perp$. The data structures are *robust* in that they correctly handle attempts to insert an element

already in the set, and attempts to delete an element not in the set. During rehashes this is known not to occur and slightly faster versions of the insertion procedure are used.

Our focus is on achieving high performance dictionary operations with a reasonable space usage. By the *load factor* of a dictionary we will understand the size of the set relative to the memory used. (For Chained Hashing, the notion of load factor traditionally disregards the space used for chained lists, but we desire equal load factors to imply equal memory usage.) As seen in [Knu98, Figure 44] the speed of Linear Probing and Double Hashing degrades rapidly for load factors above 1/2. On the other hand, none of the schemes improve much for load factors below 1/4. As Cuckoo Hashing only works when the size of each table is larger than the size of the set, we can only perform a comparison for load factors less than 1/2. To allow for doubling and halving of the table size, we allow the load factor to vary between 1/5 and 1/2, focusing especially on the "typical" load factor of 1/3. For Cuckoo Hashing and Two-Way Chaining there is a chance that an insertion may fail, causing a "forced rehash". If the load factor is larger than a certain threshold, somewhat arbitrarily set to 5/12, we use the opportunity to double the table size. By our experiments this only slightly decreases the average load factor.

Apart from Chained Hashing, the schemes considered have in common the fact that they have only been analyzed under randomness assumptions that are currently impractical to realize. However, experience shows that rather simple and efficient hash function families yield performance close to that predicted under stronger randomness assumptions. We use a function family from [DfHKP97] with range $\{0,1\}^q$ for positive integer $q$. For every odd $a$, $0 < a < 2^w$, the family contains the function $h_a(x) = (ax \bmod 2^w) \operatorname{div} 2^{w-q}$. Note that evaluation can be done very efficiently by a 32 bit multiplication and a shift. However, this choice of hash function restricts us to consider hash tables whose sizes are powers of two. A random function from the family (chosen using C's `rand` function) appears to work fine with all schemes except Cuckoo Hashing. For Cuckoo Hashing we experimented with various hash functions and found that Cuckoo Hashing was rather sensitive to the choice of hash function. It turned out that the exclusive or of three independently chosen functions from the family of [DfHKP97] was fast and worked well. We have no good explanation for this phenomenon. For all schemes, various alternative hash families were tried, with a decrease in performance.

All methods have been implemented in C. We have striven to obtain the fastest possible implementation of each scheme. Specific choices made and details differing from the references are:

Chained Hashing. C's `malloc` and `free` functions were found to be a performance bottleneck, so a simple "freelist" memory allocation scheme is used. Half of the allocated memory is used for the hash table, and half for list elements. If the data structure runs out of free list elements, its size is doubled. We store the first element of each linked list directly in the hash table. This often saves one cache miss. It also slightly improves memory utilization, in the expected sense. This is because every

non-empty chained list is one element shorter and because we expect more than half of the hash table cells to contain a linked list for the load factors considered here.

DOUBLE HASHING. To prevent the tables from clogging up with deleted cells, resulting in poor performance for unsuccessful lookups, all keys are rehashed when 2/3 of the hash table is occupied by keys and "deleted" markers. The fraction 2/3 was found to give a good tradeoff between the time for insertion and unsuccessful lookups.

LINEAR PROBING. Our first implementation, like that in [Sil98], employed deletion markers. However, we found that using the deletion method described in [Knu98, Algoritm R] was considerably faster, as far fewer rehashes were needed.

TWO-WAY CHAINING. We allow four keys in each bucket. This is enough to keep the probability of a forced rehash low for hundreds of thousands of keys, by the results in [BM01]. For larger collections of keys one should allow more keys in each bucket, resulting in general performance degradation.

CUCKOO HASHING. The architecture on which we experimented could not parallelize the two memory accesses in lookups. Therefore we only evaluate the second hash function after the first memory lookup has shown unsuccessful.

Some experiments were done with variants of CUCKOO HASHING. In particular, we considered ASYMMETRIC CUCKOO, in which the first table is twice the size of the second one. This results in more keys residing in the first table, thus giving a slightly better average performance for successful lookups. For example, after a long sequence of alternate insertions and deletions at load factor 1/3, we found that about 76% of the elements resided in the first table of ASYMMETRIC CUCKOO, as opposed to 63% for CUCKOO HASHING. There is no significant slowdown for other operations. We will describe the results for ASYMMETRIC CUCKOO when they differ significantly from those of CUCKOO HASHING.

### 2.4.3 Setup

Our experiments were performed on a PC running Linux (kernel version 2.2) with an 800 MHz Intel® Pentium® III processor, and 256 MB of memory (PC100 RAM). The processor has a 16 KB level 1 data cache and a 256 KB level 2 "advanced transfer" cache. As will be seen, our results nicely fit a simple model parameterized by the cost of a cache miss and the expected number of probes to "random" locations. They are thus believed to have significance for other hardware configurations. An advantage of using the Pentium® processor for timing experiments is its `rdtsc` instruction which can be used to measure time in clock cycles. This gives access to very precise data on the behavior of algorithms. In our case it also supplies a way of discarding

measurements significantly disturbed by interrupts from hardware devices or the process scheduler, as these show up as a small group of timings significantly separated from all other timings. Programs were compiled using the `gcc` compiler version 2.95.2, using optimization flags `-O9 -DCPU=586 -march=i586 -fomit-frame-pointer -finline-functions -fforce-mem -funroll-loops -fno-rtti`. As mentioned earlier, we use a global clock cycle counter to time operations. If the number of clock cycles spent exceeds 5000, and there was no rehash, we conclude that the call was interrupted, and disregard the result (it was empirically observed that no operation ever took between 2000 and 5000 clock cycles). If a rehash is made, we have no way of filtering away time spent in interrupts. However, all tests were made on a machine with no irrelevant user processes, so disturbances should be minimal. On our machine it took 32 clock cycles to call the `rdtsc` instruction. These clock cycles have been subtracted from the results.

### 2.4.4 Results

#### Dictionaries of stable size

Our first test was designed to model the situation in which the size of the dictionary is not changing too much. It considers a sequence of mixed operations generated at random. We constructed the test operation sequences from a collection of high quality random bits publicly available on the Internet [Mar]. The sequences start by insertion of $n$ distinct random keys, followed by $3n$ times four operations: A random unsuccessful lookup, a random successful lookup, a random deletion, and a random insertion. We timed the operations in the "equilibrium", where the number of elements is stable. For load factor 1/3 our results appear in Figure 2.3, which shows an average over 10 runs. As Linear Probing was consistently faster than Double Hashing, we chose it as the sole open addressing scheme in the plots. Time for forced rehashes was added to the insertion time. The results had a large variance, over the 10 runs, for sets of size $2^{12}$ to $2^{16}$. Outside this range the extreme values deviated from the average by less than about 7%. The large variance sets in when the data structure starts to fill the level 2 cache. We believe it is due to other processes evicting parts of the data structure from cache.

As can be seen, the time for lookups is almost identical for all schemes as long as the entire data structure fits in level 2 cache, i.e., for $n < 2^{16}/3$. After this the average number of random memory accesses (with the probability of a cache miss approaching 1) shows up. This makes linear probing an average case winner, with Cuckoo Hashing and Two-Way Chaining following about 40 clock cycles behind. For insertion the number of random memory accesses again dominates the picture for large sets, while the higher number of in-cache accesses and more computation makes Cuckoo Hashing, and in particular Two-Way chaining, relatively slow for small sets. The cost of forced rehashes sets in for Two-Way Chaining for sets of more than a million elements, at which point better results may have been obtained by a larger bucket size. For deletion Chained Hashing lags behind for large sets due to random memory

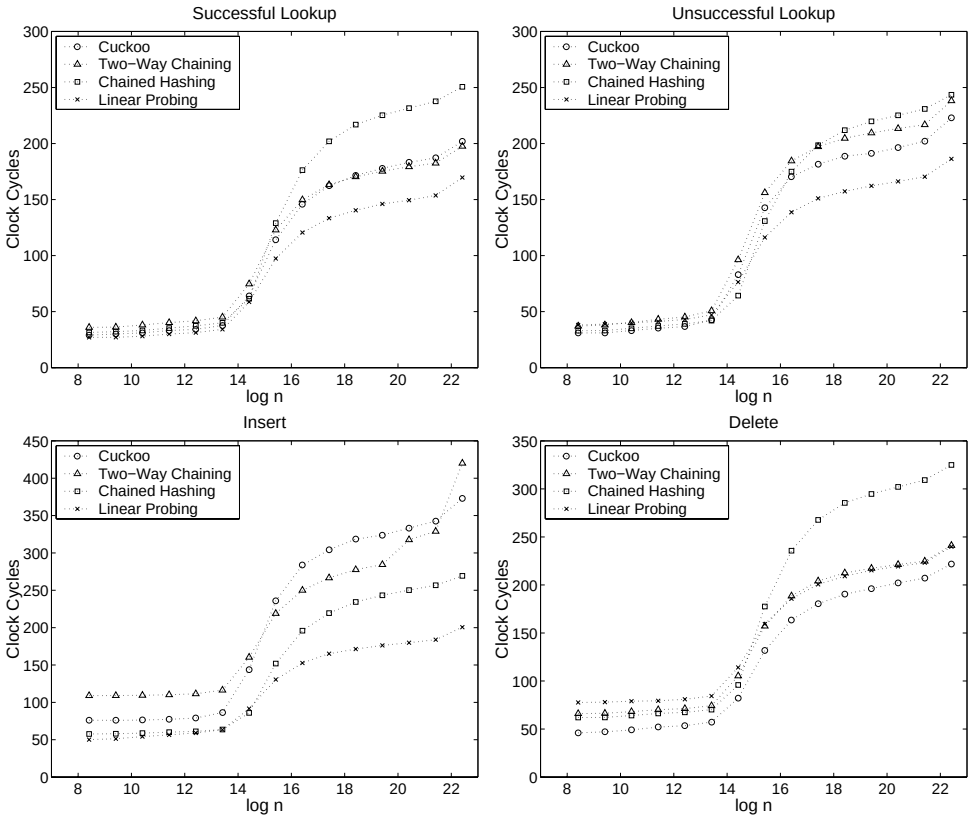


Figure 2.3: The average time per operation in equilibrium for load factor 1/3.

accesses when freeing list elements, while the simplicity of Cuckoo Hashing makes it the fastest scheme. We suspect that the slight rise in time for the largest sets in the test is due to saturation of the bus, as the machine runs out of memory and begins swapping.

At this point we should mention that the good cache utilization of Linear Probing and Two-Way Chaining depends on the cache lines being considerably larger than keys (and any associated information placed together with keys). If this is not the case, it causes the number of cache misses to rise significantly. The other schemes discussed here do not deteriorate in this way.

## Growing and shrinking dictionaries

The second test concerns the cost of insertions in growing dictionaries and deletions in shrinking dictionaries. This will be different from the above due to the cost of rehashes. Together with Figure 2.3 this should give a fairly complete picture of the performance of the data structures under general sequences of operations. The first operation sequence inserts $n$ distinct random keys, while the second one deletes them. The plot is shown in Figure 2.4. For small sets the time per operation seems unstable, and dominated by memory allocation overhead (if minimum table size $2^{10}$ is used, the curves become monotone). For

sets of more than $2^{12}$ elements the largest deviation from the averages over 10 runs was about 6%. Disregarding the constant minimum amount of memory used by any dictionary, the average load factor during insertions was within 2% of 1/3 for all schemes except Chained Hashing whose average load factor was about 0.31. During deletions all schemes had average load factor 0.28. Again the fastest method is Linear Probing, followed by Chained Hashing and Cuckoo Hashing. This is largely due to the cost of rehashes.

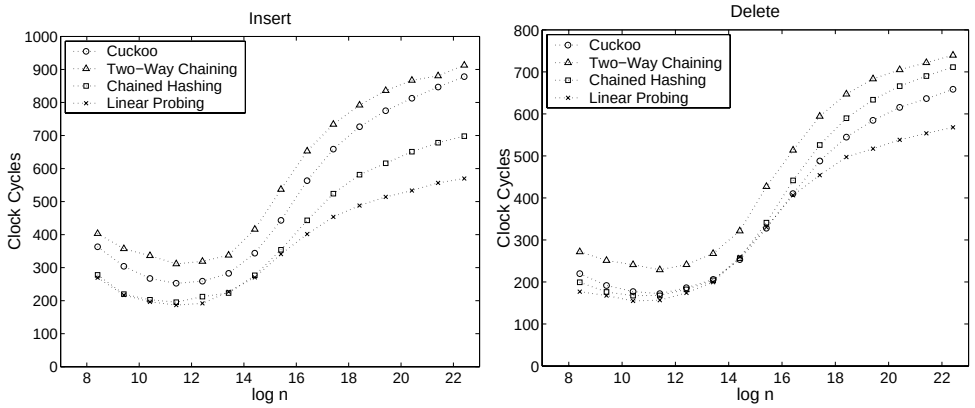


Figure 2.4: The average time per insertion/deletion in a growing/shrinking dictionary for average load factor $\approx 1/3$.

### DIMACS tests

Access to data in a dictionary is rarely random in practice. In particular, the cache is more helpful than in the above random tests, for example due to repeated lookups of the same key, and deletion of short-lived keys. As a rule of thumb, the time for such operations will be similar to the time when all of the data structure is in cache. To perform actual tests of the dictionaries on more realistic data, we chose a representative subset of the dictionary tests of the 5th DIMACS implementation challenge [McG]. The tests involving string keys were preprocessed by hashing strings to 32 bit integers, as described in Section 2.2. This preserves, with high probability, the access pattern to keys. For each test we recorded the average time per operation, not including the time used for preprocessing. The minimum and maximum of six runs can be found in Tables 2.5 and 2.6, which also lists the average load factor. Linear probing is again the fastest, but mostly just 20-30% faster than the Cuckoo schemes.

### The number of cache misses during insertion

We have seen that the number of random memory accesses (i.e., cache misses) is critical to the performance of hashing schemes. Whereas there is a very precise understanding of the probe behavior of the classic schemes (under suitable randomness assumptions), the analysis of the expected time for insertions in

| | Joyce | | Eddington | |
|---|---|---|---|---|
| LINEAR | 42 - 45 | (.35) | 26 - 27 | (.40) |
| DOUBLE | 48 - 53 | (.35) | 32 - 35 | (.40) |
| CHAINED | 49 - 52 | (.31) | 36 - 38 | (.28) |
| A.CUCKOO | 47 - 50 | (.33) | 37 - 39 | (.32) |
| CUCKOO | 57 - 63 | (.35) | 41 - 45 | (.40) |
| TWO-WAY | 82 - 84 | (.34) | 51 - 53 | (.40) |

Figure 2.5: Average clock cycles per operation and load factors for two DIMACS string tests.

| | 3.11-Q-1 | | Smalltalk-2 | | 3.2-Y-1 | |
|---|---|---|---|---|---|---|
| LINEAR | 99 - 103 | (.30) | 68 - 72 | (.29) | 85 - 88 | (.32) |
| DOUBLE | 116 - 142 | (.30) | 77 - 79 | (.29) | 98 - 102 | (.32) |
| CHAINED | 113 - 121 | (.30) | 78 - 82 | (.29) | 90 - 93 | (.31) |
| A.CUCKOO | 166 - 168 | (.29) | 87 - 95 | (.29) | 95 - 96 | (.32) |
| CUCKOO | 139 - 143 | (.30) | 90 - 96 | (.29) | 104 - 108 | (.32) |
| TWO-WAY | 159 - 199 | (.30) | 111 - 113 | (.29) | 133 - 138 | (.32) |

Figure 2.6: Average clock cycles per operation and load factors for three DIMACS integer tests.

Section 2.3.1 is rather crude, establishing just a constant upper bound. One reason that our calculation does not give a very tight bound is that we use a pessimistic estimate on the number of key moves needed to accommodate a new element in the dictionary. Often a free cell will be found even though it could have been occupied by another key in the dictionary. We also pessimistically assume that a large fraction of key moves will be spent backtracking from an unsuccessful attempt to place the new key in the first table.

Figure 2.7 shows experimentally determined values for the average number of probes during insertion for various schemes and load factors below 1/2. We disregard reads and writes to locations known to be in cache, and the cost of rehashes. Measurements were made in "equilibrium" after $10^5$ insertions and deletions, using tables of size $2^{15}$ and truly random hash function values. It is believed that this curve is independent of the table size (up to vanishing terms). The curve for LINEAR PROBING does not appear, as the number of non-cached memory accesses depends on cache architecture (length of the cache line), but it is typically very close to 1. The curve for CUCKOO HASHING seems to be $2 + 1/(4 + 8\alpha) \approx 2 + 1/(4\epsilon)$. This is in good correspondence with (2.4) of the analysis in Section 2.3.1. As noted in Section 2.3, the insertion algorithm of CUCKOO HASHING is biased towards inserting keys in $T_1$. If we instead of starting the insertion in $T_1$ choose the start table at random, the number of cache misses decreases slightly for insertion. This is because the number of free cells in $T_1$ increases as the load balance becomes even. However, this also means a slight increase in lookup time. Also note that since insertion checks

if the element is already inserted, CUCKOO HASHING uses at least two cache misses. The initial lookup can be exploited to get a small improvement in insertion performance, by inserting right away when *either* cell $T_1[h_1(x)]$ or $T_2[h_2(x)]$ is vacant. It should be remarked that the highest possible load factor for TWO-WAY CHAINING is $O(1/\log\log n)$.

Since lookup is very similar to insertion in CHAINED HASHING, one could think that the number of cache misses would be equal for the two operations. However, in our implementation, obtaining a free cell from the freelist may result in an extra cache miss. This is the reason why the curve for CHAINED HASHING in the figure differs from a similar plot in Knuth [Knu98, Figure 44].

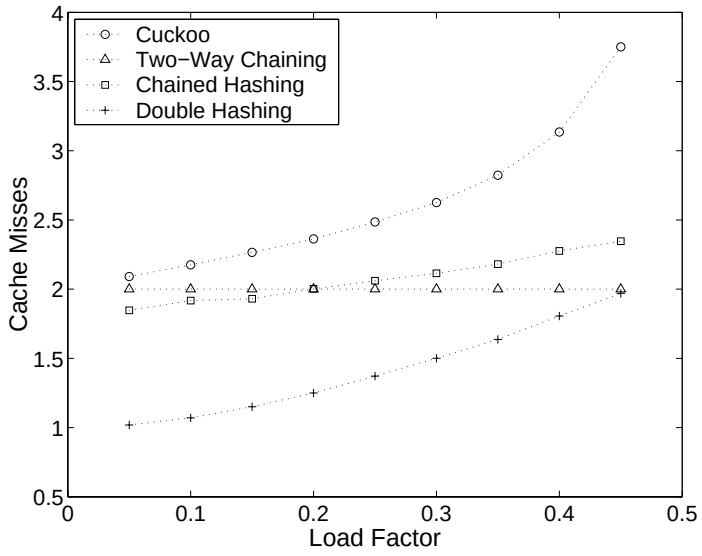


Figure 2.7: The average number of random memory accesses for insertion.

### 2.4.5 Model

In this section we look at a simple model of the time it takes to perform a dictionary operation, and note that our results can be explained in terms of this model. On a modern computer, memory speed is often the bottleneck. Since the operations of the investigated hashing methods mainly perform reads and writes to memory, we will assume that cache misses constitute the dominant part of the time needed to execute a dictionary operation. This leads to the following model of the time per operation.

$$\text{Time} = O + N \cdot R \cdot (1 - C/T) \ , \tag{2.5}$$

where the parameters of the model are described by

- O – Constant overhead of the operation.
- R – Average number of memory accesses.

- C – Cache size.
- T – Size of the hash tables.
- N – Cost of a non-cache read.

The term $R \cdot (1 - C/T)$ is the expected number of cache misses for the operations with $(1 - C/T)$ being the probability that a random probe into the tables results in a cache miss. Note that the model is not valid when the table size $T$ is smaller than the cache size $C$. The size $C$ of the cache and the size $T$ of the dictionary are well known. From Figure 2.7 we can, for the various hashing schemes and for a load factor of 1/3, read the average number $R$ of memory accesses needed for inserting an element. Note that several accesses to consecutive elements in the hash table are counted as one random access, since the other accesses are then in cache. The overhead of an operation, $O$, and the cost of a cache miss, $N$, are unknown factors that we will estimate.

Performing experiments, reading and writing to and from memory, we observed that the time for a read or a write to a location known not to be in cache could vary dramatically depending on the state of the cache. For example, when a cache line is to be used for a new read, the time used is considerably higher if the old contents of the cache line has been written to, since the old contents must then first be moved to memory. For this reason we expect parameter $N$ to depend slightly on both the particular dictionary methods and the combination of dictionary operations. This means that $R$ and $T$ are the only parameters not dependent on the methods used.

| Method | $N$ | $O$ |
|---|---|---|
| Cuckoo | 71 | 142 |
| Two-Way | 66 | 157 |
| Chained | 79 | 78 |
| Linear | 88 | 89 |
| Average | 76 | - |

Figure 2.8: Estimated parameters according to the model for insertion.

Using the timings from Figure 2.3 and the average number of cache misses for insert observed in Figure 2.7, we estimated $N$ and $O$ for the four hashing schemes. As mentioned, we believe the slight rise in time for the largest sets in the tests of Figure 2.3 to be caused by other non-cache related factors. So since the model is only valid for $T \geq 2^{16}$, the two parameters were estimated for timings with $2^{16} \leq T \leq 2^{23}$. The results are shown in Table 2.8. As can be seen from the table, the cost of a cache miss varies slightly from method to method. The largest deviation from the average is about 15%.

To investigate the accuracy of our model we plotted in Figure 2.9 the estimated curves for insertion together with the observed curves used for estimating the parameters. As can be seen, the simple model explains the observed values quite nicely. The situation for the other operations is similar.

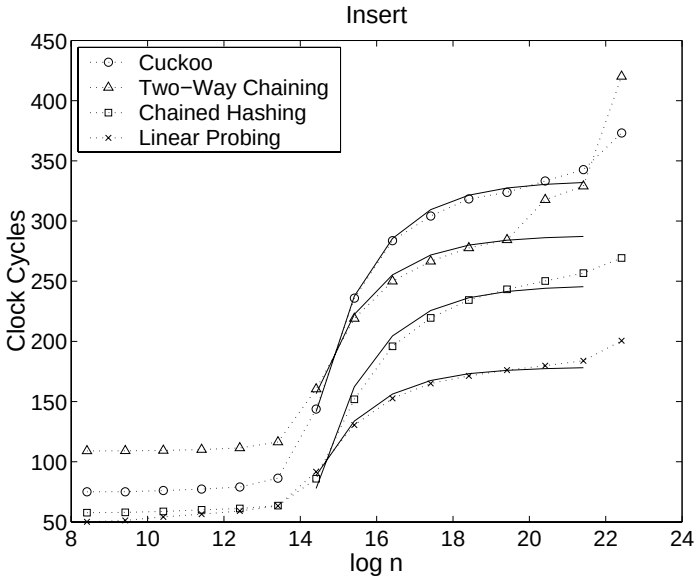


Figure 2.9: Model versus observed data.

Having said this, we must admit that the values of $N$ and $O$ estimated for the schemes cannot be accounted for. In particular, it is clear that the true behavior of the schemes is more complicated than suggested by the model.

## 2.5 Conclusion

We have presented a new dictionary with worst case constant lookup time. It is very simple to implement, and has average case performance comparable to the best previous dictionaries. Earlier schemes with worst case constant lookup time were more complicated to implement and had worse average case performance. Several challenges remain. First of all an explicit practical hash function family that is provably good for the scheme has yet to be found. For example, future advances in explicit expander graph construction could make Siegel's hash functions practical. Secondly, we lack a precise understanding of why the scheme exhibits low constant factors. In particular, the curve of Figure 2.7 needs to be explained. Another point to investigate is whether using more tables yields practical dictionaries. Experiments in [PR01b] suggest that space utilization could be improved to more than 80%, but it remains to be seen how this would affect insertion performance.

# Chapter 3

## Minimal perfect hashing with optimal evaluation complexity

This chapter deals with families of hash functions that are *perfect* for the size $n$ subsets of the finite universe $U = \{0, \ldots, u-1\}$. For any subset $S$ of $U$ of size $n$, such a perfect family contains a function that is 1-1 on $S$ ("perfect" for $S$). We consider perfect families with range $\{0, \ldots, a-1\}$, where $a \geq n$.

A perfect hash function $h$ for a set $S$ can be used to associate with each element $x \in S$ some associated information (of fixed size), such that the time needed for accessing the associated information is essentially that of evaluating the perfect hash function: Store the information associated with $x$ in entry $h(x)$ of an $a$-element array. If one also wants to be able to verify that $x \in S$ then $x$ should be written along with its associated information. This abstract data structure is called a (static) dictionary.

The attractiveness of using perfect hash functions for the above depends on several characteristics of the perfect family of hash functions.

1. The efficiency of evaluation, in terms of computation and the number of probes into the description.

2. The complexity of finding a perfect function for a given set in the family.

3. How close to $n$ the size of the range of the functions can be chosen.

4. The space is needed to store a function.

It turns out that, for suitable perfect families of hash functions, the answers to all of these questions are satisfactory in the sense that it is possible to do (more or less) as well as one could hope for. In practice, nevertheless, heuristics that typically work well seem to be used rather than theoretically optimal schemes. It is thus still of interest to find families with good properties from a theoretical as well as from a practical point of view.

This chapter presents a perfect family of hash functions that, in a unit-cost word RAM model with multiplication, matches the best results regarding items 2 and 3 above, and at the same time improves upon the best known efficiency of evaluation. The (constant) amount of computation required for

evaluating a hash function is about halved. In particular, only one multiplication is used, which is optimal if constant evaluation time is desired. Also, the number of *adaptive* probes into the function description (probes depending on the argument to the function and/or previous probes) is reduced from two to one, which is optimal. The latter is particularly important if the perfect hash function resides on secondary storage where seek time is the dominant cost. The family is surpassed in space usage (by rigorously analyzed schemes) only by families that are rather complicated and inefficient with respect to the evaluation of the function.

Another attractive feature of our family is the simple closed form of its functions:

$$x \mapsto (f(x) + d_{g(x)}) \bmod n$$

where functions $f$ and $g$ are chosen from universal families, and the array $d$ contains $O(n)$ integers from $\{0, \ldots, n-1\}$. (The functions $f$ and $g$ need not be totally independent, which is why one multiplication suffices to compute them both.) A function of this form that is 1-1 on a given set can be found in expected time $O(n)$.

## Previous work

Czech, Havas and Majewski provide a comprehensive survey of perfect hashing [CHM97]. We review some of the most important results.

Fredman, Komlós and Szemerédi [FKS84] showed that it is possible to construct space efficient perfect hash functions with range size $a = O(n)$ that can be evaluated in constant time. Their model of computation is a word RAM with unit cost arithmetic operations and memory lookups, and where an element of $U$ fits into one machine word. Dietzfelbinger [Df96] showed that multiplication, addition and standard bit operations suffice to implement the FKS hash functions (in particular, division is not necessary). Henceforth we will concentrate on hash functions with range $a = O(n)$ that can be *evaluated in constant time* on a RAM with this standard set of operations. The result of a multiplication is assumed to span two words.

An FKS hash function has a description that occupies $O(n)$ machine words. At the expense of an extra level of indirect addressing, it can be turned into a *minimal* perfect hash function, that is, one with range $a = n$. The description size is then $6n$ words (in a straightforward implementation), and at least two words are probed adaptively during evaluation. Schmidt and Siegel [SS90b] utilize compact encoding techniques to compress the hash function description to $O(n + \log\log u)$ bits, which is optimal within a constant factor. The scheme is hard to implement, and the evaluation time is prohibitive (although constant). The space usage was improved to essentially $1 + o(1)$ times optimal by Hagerup and Tholey [HT01]. However, both schemes are mainly of theoretical interest. In the following, we will consider families whose functions can be stored in $O(n)$ machine words.

The fastest algorithm for deterministically constructing a perfect hash function with constant evaluation time runs in time $O(n \log n)$, see Chapter 6. Randomized construction algorithms offer better expected performance. An FKS

perfect hash function can be constructed in optimal expected time $O(n)$ if the algorithm has access to a source of random bits, e.g., in the form of an instruction that sets a machine word to a uniformly random value. Among randomized algorithms a goal is to limit the number of random bits used. The best result in this respect is due to Dietzfelbinger et al. [DfGMP92], who use $O(\log n + \log\log u)$ random bits to achieve expected $O(n)$ time construction. Another goal is to make the construction proceed in linear time with high probability, rather than just in the expected sense [BH91, DfGMP92, DfMadH90]. We will not pursue these two goals, but settle for $O(n)$ expected construction time.

Some work on minimal perfect hashing has been done under the assumption that the algorithm can pick and store truly random functions [CHM92, HM93]. Since the space requirements for truly random functions makes this unsuitable for implementation, one has to settle for pseudo-random functions in practice. Empirical studies show that limited randomness properties are often as good as total randomness. Fox et al. [FCH92, FHCD92] studied some families that share several features with the one presented here. Their results indicate convincing practical performance, and suggest that it is possible to bring down the storage requirements further than proved here. However, it should be warned that doing well in most cases may be much easier than doing well in the worst case. In [CHM97, Section 6.7] it is shown that three published algorithms for constructing minimal perfect hash functions, claiming (experimentally based) expected polynomial running times, had in fact expected *exponential* running times. But bad behavior was so rare that it had not been encountered during the experiments.

### The Tarjan-Yao displacement scheme

The family of hash functions presented here can be seen as a variation of an early perfect family of hash functions due to Tarjan and Yao [TY79]. Their family requires a universe of size $u = O(n^2)$ (which they improve to $u = O(n^c)$ for constant $c > 2$). The idea is to split the universe into blocks of size $O(n)$, each of which is assigned a "displacement" value. The $i$th element within the $j$th block is mapped to $i + d_j$, where $d_j$ is the displacement value of block $j$. Suitable displacement values can always be found, but in general displacement values (and thus hash table size) larger than $n$ may be required. A "harmonic decay" condition on the distribution of elements within the blocks ensures that suitable displacement values in the range $\{0, \ldots, n\}$ can be found, and that they can in fact be found "greedily" in decreasing order of the number of elements within the blocks. To achieve harmonic decay, Tarjan and Yao first perform a displacement "orthogonal" to the other.

The central observation of this chapter is that a reduction of the universe to size $O(n^2)$, as well as harmonic decay, can be achieved using universal hash functions. Or equivalently, that buckets in a (universal) hash table can be resolved using displacements.

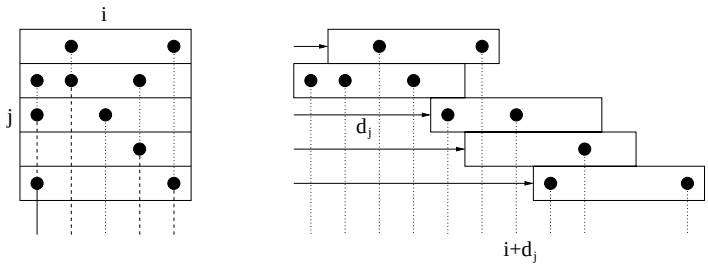


Figure 3.1: Tarjan-Yao displacement scheme

### Overview of chapter

We describe and analyze our perfect hash function construction in Section 3.1.1 and 3.1.2. Several choices of the construction are left open, and Section 3.1.3 gives some concrete instances of perfect families. In Section 3.2 we describe some variants of the basic construction, in particular arguing that a single multiplication suffices for the evaluation of functions in (a certain instance of) our family. Finally, in Section 3.3 it is shown that the evaluation complexity of our perfect hash functions, with respect to "expensive" instructions and number of memory probes, is optimal.

## 3.1 A perfect family of hash functions

### 3.1.1 Definition of family

The concept of universality [CW79] plays an important role in the analysis of our family. We use the following notation.

**Definition 3.1** *A family of functions* $H_r = \{h_1, \ldots, h_k\}$, $h_i : U \to \{0, \ldots, r-1\}$, *is* $c$-universal *if for any* $x, y \in U$, $x \neq y$, $\Pr_i[h_i(x) = h_i(y)] \leq c/r$. *It is* $(c, 2)$-universal *if for any* $x, y \in U$, $x \neq y$, *and* $p, q \in \{0, \ldots, r-1\}$, $\Pr_i[h_i(x) = p \text{ and } h_i(y) = q] \leq c/r^2$.

Many such families with constant $c$ are known, see, e.g., [Df96]. For our application the important thing to note is that there are universal families that allow efficient storage and evaluation of their functions. More specifically, $O(\log u)$ (and even $O(\log n + \log\log u)$) bits of storage suffice, and a constant number of simple arithmetic and bit operations are enough to evaluate the functions. Furthermore, $c$ is typically in the range $1 \leq c \leq 2$. Henceforth we will refer to such efficient universal families only.

The second ingredient in the family definition is a *displacement* function. Generalizing Tarjan and Yao's use of integer addition, we allow the use of any group structure on the blocks. It is assumed that the elements $\{0, \ldots, a-1\}$ of a block correspond to (distinct) elements in a group $(G, \boxplus, e)$, such that group operations may be performed on them. We assume the group operation and element inversion to be computable in constant time.

**Definition 3.2** *A* displacement function *for a group $(G, \boxplus, e)$ is a function of the form $x \mapsto x \boxplus d$ for $d \in G$. The element $d$ is called the* displacement value *of the function.*

In the following it may be helpful to simply think of the displacement functions as addition modulo $a$. We are ready to define our perfect family of hash functions.

**Definition 3.3** *Let $(G, \boxplus, e)$ be a group, $D \subseteq G$ a set of displacement values, and let $H_a$ and $H_b$ be $c_f$- and $c_g$-universal, respectively. We define the following family of functions from $U$ to $G$:*

$$\mathcal{H}(\boxplus, D, a, b) = \{x \mapsto f(x) \boxplus d_{g(x)} \mid f \in H_a, g \in H_b, d_i \in D \text{ for } i = 0, \ldots, b-1\} \ .$$

Evaluation of a function in $\mathcal{H}(\boxplus, D, a, b)$ consists of using a displacement function, determined by $g(x)$, on the value $f(x)$. In terms of the Tarjan-Yao scheme, $g$ assigns a block number to each element of $U$, and $f$ determines its number within the block. In the next section we will show that when $a > c_f n/4$ and $b \geq 2c_g n$, it is possible for any set $S$ of size $n$ to find $f$, $g$ and displacement values such that the resulting function is perfect for $S$. The family requires storage of $b$ displacement values. Focusing on instances with a reasonable memory usage, we from now on assume that $b = O(n)$ and that elements of $D$ can be stored in one machine word.

The range of functions in $\mathcal{H}(\boxplus, D, a, b)$ is $\{x \boxplus d \mid x \in \{0, \ldots, a-1\}, d \in D\}$. The size of this set depends on the group in question and the set $D$. Since our interest is in functions with range $n$ (or at most $O(n)$), we assume $a \leq n$. In Section 3.2.2 the issue of families with larger range is briefly discussed.

Choosing group operator addition modulo $n$ and $D = \{0, \ldots, n-1\}$, our family is a special case of a hash function family defined in [DfMadH90, Section 4]. There, several interesting properties of randomly chosen functions from the family are exhibited, but the issue of finding perfect functions in the family is not addressed.

### 3.1.2 Analysis

This section gives a constructive randomized method for finding perfect hash functions in the family $\mathcal{H}(\boxplus, D, a, b)$ for suitable $D$, $a$ and $b$. The key to the existence of proper displacement values is a certain "goodness" condition on $f$ and $g$. Let $B(g, i) = \{x \in S \mid g(x) = i\}$ denote the elements in the $i$th block given by $g$.

**Definition 3.4** *Let $r \geq 1$. A pair $(f, g) \in H_a \times H_b$, is* $r$-good *(for $S$) if*

1. *The function $x \mapsto (f(x), g(x))$ is 1-1 on $S$, and*
2. $\sum_{i, |B(g,i)|>1} |B(g, i)|^2 \leq n/r$ .

The first condition says that $f$ and $g$ successfully reduce the universe to size $ab$ (clearly, if $(f(x_1), g(x_1)) = (f(x_2), g(x_2))$ then regardless of displacement values, $x_1$ and $x_2$ collide). The second condition implies the harmonic decay condition of Tarjan and Yao (however, it is not necessary to know their condition to understand what follows). We show a technical lemma, estimating the probability of randomly finding an $r$-good pair of hash functions. We denote by $\binom{U}{n}$ the set of all subsets of $U$ of size $n$.

**Lemma 3.1** *Assume $a \geq c_f n/4r$ and $b \geq 2c_g rn$. For any $S \in \binom{U}{n}$, a randomly chosen pair $(f, g) \in H_a \times H_b$ is $r$-good for $S$ with positive probability, namely more than $(1 - \frac{2c_g rn}{b})(1 - \frac{c_f n}{4ra})$.*

*Proof.* By $c_g$-universality, the expected value for the sum

$$\sum_i \binom{|B(g,i)|}{2} = \sum_{\substack{\{u,v\} \in \binom{S}{2} \\ g(u)=g(v)}} 1$$

is bounded by $\binom{n}{2}\frac{c_g}{b} < \frac{c_g n^2}{2b}$, so applying Markov's inequality, the sum has value less than $n/4r$ with probability $> 1 - \frac{2c_g rn}{b}$. Since $|B(g,i)|^2 \leq 4\binom{|B(g,i)|}{2}$ for $|B(g,i)| > 1$, we then have that $\sum_{i,|B(g,i)|>1} |B(g,i)|^2 \leq n/r$. Given a function $g$ such that these inequalities hold, we would like to bound the probability that $x \mapsto (f(x), g(x))$ is 1-1 on $S$. By reasoning similar to before, we get that for randomly chosen $f$, the expected number of collisions among elements of $B(g,i)$ is at most $\binom{|B(g,i)|}{2} c_f/a$. Summing over $i$ we get the expected total number of collisions to be less than $c_f n/4ra$. Hence, using Markov's inequality, with probability more than $1 - \frac{c_f n}{4ra}$ there is no collision. By the law of conditional probabilities, the probability of fulfilling both $r$-goodness conditions can be found by multiplying the probabilities found. □

We now show that $r$-goodness is sufficient for displacement values to exist, by means of a constructive argument in the style of [TY79, Theorem 1].

**Theorem 3.1** *Let $(f, g) \in H_a \times H_b$ be $r$-good for $S \in \binom{U}{n}$, $r \geq 1$, and let $|D| \geq n$. Then there exist displacement values $d_0, \ldots, d_{b-1} \in D$, such that $x \mapsto f(x) \boxplus d_{g(x)}$ is 1-1 on $S$.*

*Proof.* Note that displacement $d_i$ is used on the elements $\{f(x) \mid x \in B(g,i)\}$, and that any $h \in \mathcal{H}(\boxplus, D, a, b)$ is 1-1 on each $B(g,i)$. We will assign the displacement values one by one, in non-increasing order of $|B(g,i)|$. At the $k$th step, we will have displaced the $k-1$ largest sets, $\{B(g,i) \mid i \in I\}$, $|I| = k-1$, and want to displace the set $B(g,j)$ such that no collision with previously displaced elements occurs. If $|B(g,j)| \leq 1$ this is trivial since $|D| \geq n$, so we assume $|B(g,j)| > 1$. The following claim finishes the proof.

**Claim 3.1** *If $|B(g,k)| > 1$, then with positive probability, namely more than $1 - \frac{1}{r}$, a randomly chosen $d \in D$ displaces $B(g,k)$ with no collision.*

The proof of the claim goes as follows. The displacement values that are *not* available are those in the set

$$\{f(x)^{-1} \boxplus f(y) \boxplus d_i \mid x \in B(g,k), y \in B(g,i), i \in I\} \ .$$

It has size at most $|B(g,k)| \sum_{i \in I} |B(g,i)| < \sum_{i,|B(g,i)|>1} |B(g,i)|^2 \leq n/r$, using first non-increasing order, $|B(g,k)| > 1$, and then $r$-goodness. Hence there must be more than $(1 - \frac{1}{r})|D|$ good displacement values in $D$. □

Lemma 3.1 implies that, for constant $\epsilon > 0$, when $a \geq (\frac{c_f}{4r} + \epsilon)n$ and $b \geq (2c_g r + \epsilon)n$, an $r$-good pair of hash functions can be found (and verified to be $r$-good) in expected time $O(n)$. The proof of Theorem 3.1, and in particular Claim 3.1, shows that for a $(1+\epsilon)$-good pair $(f, g)$, the strategy of trying random displacement values in $D$ successfully displaces all blocks with more than one element in expected $1/\epsilon$ attempts per block. When $O(n)$ random elements of $D$ can be picked in $O(n)$ time, this runs in expected time $O(n/\epsilon) = O(n)$. Finally, if displacing all blocks with only one element is easy (as is the case in the examples we look at in Section 3.1.3), the whole construction runs in expected time $O(n)$.

For a 1-good pair $(f, g)$, Theorem 3.1 gives the *existence* of proper displacement values, but no linear time algorithm for finding them.

### 3.1.3 Instances of the family

In Section 3.1 we specified at a rather abstract level how to construct a perfect family. We now proceed to look at some specific instances. In particular, the choice of group and set of displacement values is discussed. For simplicity, we use universal families with constant 1. The number of displacement values, $b$, must be at least $2n$ to ensure existence of perfect functions, and at least $(2 + \epsilon)n$, for constant $\epsilon > 0$, to ensure expected linear time construction. For convenience, self-contained definitions of the families are given.

#### Addition

The set of integers with addition is a very natural choice of group. For $D = \{0, \ldots, n-1\}$, we get the family:

$$\mathcal{H}_{\mathbb{Z}} = \{x \mapsto f(x) + d_{g(x)} \mid f \in H_{\lceil n/4 \rceil},\ g \in H_b,\ 0 \leq d_i < n \text{ for } 0 \leq i < b\} \ .$$

The range of hash functions in the family is $\{0, \ldots, \lceil \frac{5}{4}n \rceil - 2\}$, so it is not minimal.

#### Addition modulo $n$

The previous family can be made minimal at the expense of a computationally more expensive group operator, addition modulo $n$:

$$\mathcal{H}_{\mathbb{Z}_n} = \{x \mapsto (f(x)+d_{g(x)}) \bmod n \mid f \in H_n,\ g \in H_b,\ 0 \leq d_i < n \text{ for } 0 \leq i < b\} \ .$$

Note that since the argument to the modulo $n$ operation is less than $2n$, it can be implemented using one comparison and one subtraction.

**Bitwise exclusive or**

The set of bit strings of length $\ell = \lceil \log n \rceil$ form the group $\mathbb{Z}_2^\ell$ under the operation of bitwise exclusive or, denoted by $\oplus$. We let $\{0, \ldots, n-1\}$ correspond to $\ell$-bit strings of their binary representation, and get

$$\mathcal{H}_{\mathbb{Z}_2^\ell} = \{x \mapsto f(x) \oplus d_{g(x)} \mid f \in H_{2^{\ell-1}},\ g \in H_b,\ d_i \in \{0,1\}^\ell \text{ for } 0 \leq i < b\} \ .$$

The range of functions in this family is (or corresponds to) the numbers $\{0, \ldots, 2^\ell - 1\}$. It is thus only minimal when $n$ is a power of two. However, for $b \geq 4n^2/2^\ell$, displacement values can be chosen such that elements of the set are mapped to $\{0, \ldots, n-1\}$: All displacement values for sets with more than one element can be chosen from $0\{0,1\}^{\ell-1}$, and single elements can be displaced to any value desired. Since some elements not in the set might map outside $\{0, \ldots, n-1\}$, a check for values larger than $n-1$ must be inserted.

**Construction time**

The discussion in Section 3.1.2 implies that perfect functions in the above families can be found efficiently. Also note that it is possible to "pack" all displacement values into $b\lceil \log n \rceil$ bits.

**Theorem 3.2** *Let $S \in \binom{U}{n}$ and let $\epsilon > 0$ be a constant. A perfect hash function for $S$ in the families $\mathcal{H}_{\mathbb{Z}}$, $\mathcal{H}_{\mathbb{Z}_n}$ and $\mathcal{H}_{\mathbb{Z}_2^\ell}$, with storage requirement $(2+\epsilon)n$ words (or $(2+\epsilon)n\lceil \log n \rceil$ bits), can be found in expected time $O(n)$.*

## 3.2 Variants

This section describes some variants of the basic scheme presented in the previous section.

### 3.2.1 Using a single multiplication

In the introduction we promised a family that was not only efficient with respect to the number of probes into the description, but also with respect to the amount of computation involved. It would seem that the evaluation of hash functions $f$ and $g$ makes the computational cost of the scheme presented here comparable to that of, e.g., the FKS scheme. However, we have one advantage that can be exploited: The hash functions used are *fixed* in advance, as opposed to other schemes where the choice of second hash function depends on the value of the first. We will show how to "simulate" our two universal hash functions with a single $(c, 2)$-universal hash function.

**Definition 3.5** *Let $V = \{0, \ldots, ab-1\}$. Functions $p_f : V \to \{0, \ldots, a-1\}$ and $p_g : V \to \{0, \ldots, b-1\}$ are said to* decompose *$V$ if the map $x \mapsto (p_f(x), p_g(x))$ is 1-1 on $V$.*

Let $H_{ab}$ be a family of $(c,2)$-universal hash functions with range $\{0,\ldots,ab-1\}$, and let $p_f$ and $p_g$ decompose $\{0,\ldots,ab-1\}$. It is not hard to show that $\{p_f \circ h \mid h \in H_{ab}\}$ and $\{p_g \circ h \mid h \in H_{ab}\}$ are $(c,2)$-universal, and hence also $c$-universal. However, because of possible dependencies, this will not directly imply that $(p_f \circ h, p_g \circ h)$ is $r$-good with positive probability, for random $h \in H_{ab}$. But we now show that, with a small penalty in the number of displacement values, this is the case.

**Lemma 3.1b** *Let $H_{ab}$ be a $(c,2)$-universal family, and let $p_f$ and $p_g$ decompose $\{0,\ldots,ab-1\}$. Assume $ab \geq cn(n+4ra)/2$. For any $S \in \binom{U}{n}$, a pair $(p_f \circ h, p_g \circ h)$ with randomly chosen $h \in H_{ab}$, is $r$-good for $S$ with positive probability, namely more than $1 - \frac{cn(n+4ra)}{2ab}$.*

*Proof.* When choosing $h \in H$ uniformly at random, the expected number of collisions between pairs of elements in $S$ is at most $\binom{n}{2}c/ab < \frac{cn^2}{2ab}$. Hence a collision occurs with probability less than $\frac{cn^2}{2ab}$. Using injectivity of the decomposition functions, the same holds for $x \mapsto ((p_f \circ h)(x), (p_g \circ h)(x))$. Since $\{p_g \circ h \mid h \in H_{ab}\}$ is $c$-universal, the argument in the proof of Lemma 3.1 shows that $\sum_{i,|B(p_g \circ h,i)|>1} |B(p_g \circ h, i)|^2 > n/r$ occurs with probability less than $\frac{2crn}{b}$. Since $\frac{cn^2}{2ab} + \frac{2crn}{b} = \frac{cn(n+4ra)}{2ab}$, the stated probability of $r$-goodness follows. □

For $a = n$ this means that $b = \lceil \frac{5c}{2} n \rceil$ is enough to ensure the existence of a 1-good pair $(p_f \circ h, p_g \circ h)$. For constant $\epsilon > 0$ and $b = \lceil (\frac{5c}{2} + \epsilon)n \rceil$ a $(1+\epsilon/4c)$-good pair can be found in an expected constant number of attempts.

Decomposition of $\{0,\ldots,ab-1\}$ can be done efficiently when $a$ or $b$ is a power of two. Then $p_f$ and $p_g$ simply pick out appropriate bits. More generally, $p_f(u) = u \text{ div } b$, $p_g(u) = u \bmod b$ (or $p_f(u) = u \bmod a$, $p_g(u) = u \text{ div } a$) is a natural choice for decomposing the range of $h$.

Returning to the claim in the introduction, we use the $(1,2)$-universal family of Dietzfelbinger [Df96], which requires just one multiplication, one addition and some simple bit operations when the range has size a power of two. Choose $a$ and $b$ as powers of two satisfying $ab \geq (n(n+4a)+\epsilon)/2$, and it is easy to compute $p_f$ and $p_g$. By Section 3.1.3, a displacement function can be evaluated using only additions and bit operations, so this means that that (apart from a few less expensive operations) one multiplication suffices.

### 3.2.2 Larger range

We have focused on perfect hash functions with range $O(n)$, since these have the most applications. It is, however, straightforward to generalize the families we have seen to ones with range $a = \omega(n)$. The number of displacement values necessary is in general $b = O(n^2/a)$. (Note that for $a > \binom{n}{2}$ a universal family in itself is perfect.) The differences to the proofs seen so far are:

- $r$-goodness is generalized so that the second requirement is
$$\sum_{i,|B(g,i)|>1} |B(g,i)|^2 \leq a/r \ .$$
- The set of displacement values must have size at least $a$.

## 3.3 Optimality

### 3.3.1 One multiplication is necessary

The unbounded fan-in circuit depth of multiplication of two $\log u$ bit numbers is $O(\log\log u/\log\log\log u)$ by [BCH86] and [CSV84]. All other instructions of our RAM model have constant depth circuits. It was shown in [AMRT96] that any dictionary for subsets of $\{0,\ldots,u-1\}$ of size $n$, using space $2^{(\log n)^{O(1)}}$ and a constant number of instructions for lookups, must employ an instruction of circuit depth $\Omega(\log\log u/\log\log\log u)$. Hence, our use of a single instruction with this circuit depth is optimal.

### 3.3.2 An adaptive probe is necessary

We now show that one cannot in general evaluate a perfect hash function with range size $a = O(n)$ on input $x$ using a constant number of word probes that depend only on $x$ (and not the results of other probes), in a data structure of $b = O(n)$ words. That is, an adaptive word probe is necessary in general to achieve constant evaluation time. A similar lower bound for the *bit probe* complexity of perfect hashing is shown in Chapter 9.

Of course, for large word length $w$, adaptivity is not necessary: If $w \geq \lceil\log n\rceil + \lceil u/b\rceil$ then a bitmap of the whole set, as well as rank information in each word, can be put into $b$ words, and nonadaptive minimal perfect hashing is easy. Also, if a perfect hash function can be described in $O(w)$ bits, a constant number of fixed probes obviously suffice.

The following theorem (with $a = O(n)$ and $b = O(n)$) shows that if $u \geq nw^{\omega(1)}$, nonadaptive schemes need to probe a nonconstant number of words to evaluate a perfect hash function. We conclude that a single adaptive word probe, as used by the family presented here, cannot be improved in general.

**Theorem 3.3** *For positive integers $a$, $b$, $t$, $w$, $n \geq 2$ and $u \geq 4a(\frac{4abw}{n^2})^t$, let $\mathcal{H}$ be a family of hash functions from $\{0,\ldots,u-1\}$ to $\{0,\ldots,a-1\}$ that is perfect for sets of size $n$. If functions from $\mathcal{H}$ can be stored in a data structure of $b$ words of $w$ bits and evaluated using $t$ nonadaptive word probes, then $t = \Omega(n^2/aw)$.*

*Proof.* Let $C_x$ be the set of $t$ indices probed when evaluating a hash function on input $x$. For any set $B \subseteq \{0,\ldots,b-1\}$ of indices in the data structure, let $U_B = \{x \in U \mid C_x \subseteq B\}$.

Let $s'$ be the minimum number of words needed to represent a perfect hash function for a set of $n$ elements from a universe of size $u' = \lceil(\frac{n^2}{2abw})^t u\rceil$. By [Meh84, Theorem III.2.3.6] we have $s' \geq \binom{n}{2}\log e(1/a - 1/u') \geq \frac{n^2}{2aw}$. We show that $t \geq s'/2$. Suppose for the sake of contradiction that $t < s'/2$. Each set $C_x$ is contained in $\binom{b-t}{s'-1-t}$ subsets of $\{0,\ldots,b-1\}$ of size $s'-1$. This means that there exists some set $B \subseteq \{0,\ldots,b-1\}$ of $s'-1$ elements such that

$$|U_B| \geq u\tbinom{b-t}{s'-1-t}/\tbinom{b}{s'-1} \geq (\tfrac{s'-t}{b})^t u > (\tfrac{n^2}{4abw})^t u \ .$$

But the words with indices in $B$ encode a perfect hash function for any size $n$ subset of $U_B$, contradicting the definition of $s'$. □

## 3.4 Conclusion

We have seen that displacements, together with universal hash functions, form the basis of very efficient (minimal) perfect hashing schemes. The efficiency of evaluation is optimal in the sense that only one adaptive probe into the data structure is needed, and only one "expensive" instruction is used.

The space consumption in bits, although quite competitive with that of other evaluation-efficient schemes, is a factor of $\Theta(\log n)$ from the theoretical lower bound. As mentioned in the introduction, some experiments suggest that the space consumption for this kind of scheme may be brought close to the optimum by simply having fewer displacement values. It would be interesting to extend the results of this chapter in that direction. This might help to further bring together the theory and practice of perfect hashing.

One result in this direction has been shown by Dietzfelbinger and Hagerup in [DH01]. They designed an improved algorithm for choosing displacement values. It allows the number of displacement values to be reduced from $(2+\epsilon)n$ to $(1+\epsilon)n$, at the cost of using slightly stronger hash functions than the universal ones used here.

# Chapter 4

## Simulating uniform hashing in constant time and optimal space

Hashing is an important tool for designing and implementing randomized algorithms and data structures. The basic idea is to use a function $h : U \to V$, called a *hash function*, that "mimics" a random function. In this way a "random" value $h(x)$ can be associated with each element from the domain $U$. This has been useful in many applications in, for example, information retrieval, data mining, cryptology, and parallel algorithms.

Many algorithms have been carefully analyzed under the assumption of *uniform hashing*, i.e., assuming that the hash function employed is a truly random function. As the representation of a random function requires $|U| \log |V|$ bits, it is usually not feasible to actually store a randomly chosen function. For many years hashing was largely a heuristic, and one used fixed functions that were empirically found to work well in cases where truly random functions could be shown to work well.

The gap between hashing algorithms and their analysis narrowed with the advent of *universal hashing* [CW79]. The key insight was that it is often possible to get provable performance guarantees by choosing hash functions at random from a small family of functions (rather than from the family of all functions). The importance of the family being small is, of course, that a function from the family can be stored succinctly. Following universal hashing, many hash function families have been proposed (e.g., [ADfM$^+$97, BCFM00, Chi94, Df96, DfGMP92, DfMadH90, GW97, IMRV99, LS98, Sie89]), and their performance analyzed in various settings.

One property of the choice of hash function that often suffices to give performance guarantees is that it maps each set of $k$ elements in $U$ to uniformly random and independent values, where $k$ is some parameter that depends on the application. If this holds for a random function from a family $\mathcal{H}$ of functions, $\mathcal{H}$ is called *$k$-wise independent*. The hash functions described by Carter and Wegman in [CW79], for example, were 2-wise independent. The first constructions of $k$-wise independent families required time $\Omega(k)$ for evaluating a hash function. (Here, and in the rest of the chapter, we will consider complexity on a RAM with word size $\Theta(\log |U| + \log |V|)$.) A breakthrough was made by Siegel [Sie89], who showed that high independence is achievable with relatively

small families of hash functions that can be evaluated in *constant* time.

The two main performance parameters of a hash function family is the space needed to represent a function and the time necessary to compute a given function value from a representation. A lower bound on the number of bits needed to achieve $k$-wise independence is $\Omega(k)$ words [ABI86, CGH+85], and there are constructions using $O(k)$ words of space in the case where $|U|$ and $|V|$ are powers of the same prime. Sometimes there is a trade-off between the space used to represent a function and its evaluation time. For example, Siegel's $k$-wise independent family requires $k^{1+\Omega(1)}$ words of space to achieve constant evaluation time, for $|U| = k^{O(1)}$.

If one applies Siegel's family with $k = n$ to a set $S$ of $n$ elements, it will map these to independent and uniformly random values. We say that it is *uniform on $S$*. However, the superlinear space usage means that, in many possible applications, the hash function description itself becomes asymptotically larger than all other parts of the data structure. In this chapter we present a family of hash functions that has the same performance as Siegel's family on any particular set of $n$ elements, and improves space to the optimal bound of $O(n)$ words.

**Theorem 4.1** *Let $S \subseteq U$ be a set of $n$ elements. For any constant $c > 0$ there is an algorithm constructing a random family of functions from $U$ to $V$ in $o(n)$ time and space, such that:*

- *With probability $1 - O(n^{-c})$ the family is uniform on $S$.*
- *There is a data structure of $O(n)$ words words representing its functions such that function values can be computed in constant time. The data structure can be initialized to a random function in $O(n)$ time.*

## Implications

The fact that the space usage is linear in $n$ means that a large class of hashing schemes can be implemented to perform, with high probability, *exactly as if uniform hashing was used*, increasing space by at most a constant factor. This means that our family makes a large number of analyses performed under the uniform hashing assumption "come true" with high probability.

Two comprehensive surveys of early data structures analyzed under the uniform hashing assumption can be found in the monographs of Gonnet [Gon84] and Knuth [Knu98]. Gonnet provides more than 100 references to books, surveys and papers dealing with the analysis of classic hashing algorithms. This large body of work has made the characteristics of these schemes very well understood, under the uniform hashing assumption. As the classic hashing algorithms are often very simple to implement, and efficient in practice, they seem to be more commonly used in practice than newer schemes with provably good behavior[1]. While our family is not likely to be of practical importance

[1] One could argue that hashing will always be a heuristic on real, deterministic machines. However, cryptographic applications have made it increasingly common to equip computers with a hardware random number generator, such as in Intel's 8xx chipsets.

for these hashing schemes, it does provide a theoretical "bridge" justifying the uniform hashing assumption for a large class of them. Previously, such justifications have been made for much more narrow classes of hashing schemes, and have only dealt with certain performance parameters (see, e.g., [SS89, SS90a]).

In addition to the classic hashing schemes, our hash functions provide a provably efficient implementation of a recent load balancing scheme of Azar et al. [ABKU99].

### Overview of the chapter

The organization of the chapter is as follows. In section 4.1 we provide the background information necessary to understand our construction. Specifically, we survey Siegel's construction, which will play an important role. Section 4.2 presents our construction and its analysis. Finally, section 4.3 gives a number of applications of our result.

## 4.1 Background

The main result of this chapter can be seen as an extension of Siegel's family of high performance hash functions [Sie89, Sie95]. The motivation for Siegel's work was that many algorithms employing hashing can be shown to work well if the hash functions are chosen at random from a *k-wise independent* family of functions, for suitably large $k$.

**Definition 4.1** *A family $\mathcal{H}$ of functions from $U$ to $V$ is* $k$-wise independent *if, for any set of distinct elements $x_1, \ldots, x_k \in U$, and any $y_1, \ldots, y_k \in V$, when $h \in \mathcal{H}$ is chosen uniformly at random,*

$$\Pr[h(x_1) = y_1, \ldots, h(x_k) = y_k] = |V|^{-k} \ .$$

In other words, a random function from a $k$-wise independent family acts like a truly random function on any set of $k$ elements of $U$. In this chapter we will assume that the range $V$ of hash functions is the set of elements in some group $R = (V, \oplus)$, where the group operation $\oplus$ can be performed in constant time on a RAM. There are many such examples of groups, for example those with group operations addition modulo $|V|$ and bitwise exclusive or.

Siegel primarily considered the case in which $|U| = k^{O(1)}$. He showed that in this case one can, for arbitrary constants $c, \epsilon > 0$, construct, in $o(k)$ time and space, a family of functions from $U$ to $V$ such that:

- The family is $k$-wise independent with probability $1 - k^{-c}$.
- There is a data structure of $k^{1+\epsilon}$ words words representing its functions such that function values can be computed in constant time. The data structure can be initialized to a random function in $k^{1+\epsilon}$ time.

The positive probability that the family is not $k$-wise independent is due to the fact that Siegel's construction relies on a certain type of expander graph that,

in lack of an explicit construction, is found by selecting a graph at random (and storing it). However, there is a small chance that the randomly chosen graph is not the desired expander, in which case there is no guarantee on the performance of the family. Also, there seems to be no known efficient way of generating a graph at random and verifying that it is the desired expander. (However, a slightly different class of expanders can be efficiently generated in this way [Alo86].)

It is no coincidence that Siegel achieves constant evaluation time only for $|U| = k^{O(1)}$. He shows the following trade-off between evaluation time and the size of the data structure:

**Theorem 4.2** (Siegel [Sie89]) *For any $k$-wise independent family $\mathcal{H}$ of functions from $U$ to $V$, any data structure using $m$ words of $O(\log |V|)$ bits to represent a function from $\mathcal{H}$ requires worst case time $\Omega(\min(\log_{m/k}(|U|/k), k))$ to evaluate a function.*

Note that when using optimal space, i.e., $m = O(k)$, one must use time $\Omega(\min(\log(|U|/k), k))$ to evaluate a function. Siegel's upper bound extends to the case where $|U|$ is not bounded in terms of $k$. However, in this case the lack of an explicit expander construction results in an exponentially larger evaluation time than in the first term of the lower bound.

Theorem 4.2 establishes that high independence requires either high evaluation time or high space usage when $|U|$ is large. A standard way of getting around problems with hashing from a large domain is to first perform a *domain reduction*, where elements of $U$ are mapped to elements of a smaller domain $U'$ using, say, universal hashing. As this mapping cannot be 1-1, the domain reduction forces some hash function values to be identical. However, for any particular set $S$ of $n$ elements, the probability of two elements in $S$ mapping to the same element of $U'$ can be made low by choosing $|U'| = n^{O(1)}$ sufficiently large.

**Definition 4.2** *A family of functions defined on $U$ is* uniform *on the set $S \subseteq U$ if its restriction to $S$ is $|S|$-wise independent.*

Using domain reduction with Siegel's family described above, one gets the following result. For $k = n$ it is similar to our main theorem, except that the space usage is superlinear.

**Theorem 4.3** (Siegel [Sie89, Sie95]) *Let $S \subseteq U$ be a set of $n = k^{O(1)}$ elements. For any constants $\epsilon, c > 0$ there is an algorithm constructing a random family $\mathcal{SI}(U, V, k, n, c, \epsilon)$ of functions from $U$ to $V$ in $o(k)$ time and space, such that:*

- *With probability $1 - n^{-c}$ the family is $k$-wise independent on $S$.*
- *There is a data structure of $O(k^{1+\epsilon})$ words words representing its functions such that function values can be computed in constant time. The data structure can be initialized to a random function in $O(k^{1+\epsilon})$ time.*

With current expander "technology", Siegel's construction exhibits high constant factors. Other proposals for high performance hash functions, due to Dietzfelbinger and Meyer auf der Heide [DfMadH90, DMadH92], appear more practical. However, these families only exhibit $O(1)$-wise independence and appear to be difficult to analyze in general.

## 4.2 Hash function construction

In this section we describe our hash function family and show Theorem 4.1. We use the notation $T[i]$ to denote the $i$th entry in an array (or vector) $T$. By $[m]$ we denote the set $\{1, \ldots, m\}$.

### 4.2.1 The hash function family

**Definition 4.3** *Let $R = (V, \oplus)$ be a group, let $\mathcal{G}$ be a family of functions from $U$ to $V$, and let $f_1, f_2 : U \to [m]$. We define the family of functions*

$$\mathcal{H}(f_1, f_2, \mathcal{G}) = \{x \mapsto T_1[f_1(x)] \oplus T_2[f_2(x)] \oplus g(x) \mid T_1, T_2 \in V^m \text{ and } g \in \mathcal{G}\}.$$

A similar way of constructing a function family was presented in [DfMadH90]. The novel feature of the above definition is the use of *two* values looked up in tables, rather than just one. The hash function family used to prove Theorem 4.1 uses Siegel's construction of function families to get the functions $f_1$ and $f_2$ and the family $\mathcal{G}$ in the above definition.

**Definition 4.4** *For $n \leq |U|$ and any constant $c > 0$ we define the random family $\mathcal{H}_{n,c} = \mathcal{H}(f_1, f_2, \mathcal{G})$ of functions as follows: Construct the random families $\mathcal{G} = \mathcal{SI}(U, V, \sqrt{n}, n, c, 1/2)$ and $\mathcal{F} = \mathcal{SI}(U, [4n], \sqrt{n}, n, c, 1/2)$ according to Theorem 4.3, and pick $f_1$ and $f_2$ independently at random from $\mathcal{F}$.*

### 4.2.2 Properties of the family

For a set $S \subseteq U$ and two functions $f_1, f_2 : U \to [m]$, let $G(f_1, f_2, S) = (A, B, E)$ be the bipartite graph with vertex sets $A = \{a_1, \ldots, a_m\}$ and $B = \{b_1, \ldots, b_m\}$, and edge set $E = \{e_x = (a_{f_1(x)}, b_{f_2(x)}) \mid x \in S\}$, where $e_x$ is labeled by $x$. Note that there may be parallel edges.

We define a *cyclic subgraph* $E' \subseteq E$ of a graph as a subset of the edges such that there is no vertex incident to exactly one edge in $E'$. A graph's *cyclic part* $C \subseteq E$ is the maximal cyclic subgraph in the graph, i.e., the edges in cycles and edges in paths connecting cycles.

**Lemma 4.1** *Let $S \subseteq U$ be a set of $n$ elements and let $\mathcal{G}$ be a family of functions from $U$ to $V$ that is $k$-wise independent on $S$. If the total number of edges in the cyclic part of $G(f_1, f_2, S) = (A, B, E)$ is at most $k$, then $\mathcal{H}(f_1, f_2, \mathcal{G})$ is uniform on $S$.*

*Proof.* Let $S'$ be the set of all elements $x \in S$ where the corresponding edge $e_x$ is in the cyclic part $C$ of $G(f_1, f_2, S)$.

The proof is by induction. First, assume that $|E \setminus C| = 0$. Since $g$ is chosen from a $k$-wise independent family, $S = S'$, and $|S'| \leq k$ we can conclude that $\mathcal{H}(f_1, f_2, \mathcal{G})$ is uniform on $S$.

It remains to show that $\mathcal{H}(f_1, f_2, \mathcal{G})$ is uniform on $S$ when $|E \setminus C| \geq 1$. Among the edges in $E \setminus C$ there has to be at least one edge with one unique endpoint. Let $e_{x^*} = (a_{f_1(x^*)}, b_{f_2(x^*)})$ be such an edge, $x^* \in S \setminus S'$. W.l.o.g. assume that $a_{f_1(x^*)}$ is the unique endpoint. By induction it holds that $\mathcal{H}(f_1, f_2, \mathcal{G})$ is uniform on $S \setminus \{x^*\}$. For $h \in \mathcal{H}(f_1, f_2, \mathcal{G})$ chosen at random, all values $h(x)$ for $x \in S \setminus \{x^*\}$ are independent of the value $T_1[f_1(x^*)]$. Additionally, given $g \in \mathcal{G}$ and all entries in vectors $T_1$ and $T_2$ except $T_1[f_1(x^*)]$, $h(x^*)$ is uniformly distributed when choosing $T_1[f_1(x^*)]$ at random. Hence $\mathcal{H}(f_1, f_2, \mathcal{G})$ is uniform on $S$. □

**Lemma 4.2** *For each set $S$ of size $n$, and for $f_1, f_2 : U \to [4n]$ chosen at random from a family that is $k$-wise independent on $S$, $k \geq 32$, the probability that the cyclic part $C$ of the graph $G(f_1, f_2, S)$ has size at least $k$ is $n/2^{\Omega(k)}$.*

*Proof.* Assume that $|C| \geq k$ and that $k$ is even (w.l.o.g.). Either there is a *connected* cyclic subgraph in $G(f_1, f_2, S)$ of size at least $k/2$ or there is a cyclic subgraph of size $k'$, where $k/2 < k' \leq k$. In the first case there is a connected subgraph in $G(f_1, f_2, S)$ with exactly $k/2$ edges and at most $k/2 + 1$ vertices. In the second case there is a subgraph with $k'$ edges and at most $k'$ vertices in $G(f_1, f_2, S)$, where $k/2 < k' \leq k$.

In the following we will count the number of different edge labeled subgraphs with $k'$ edges and at most $k' + 1$ vertices for $k/2 \leq k' \leq k$ to bound the probability of such a subgraph to appear in $G(f_1, f_2, S)$. Hence, we also get an upper bound on the probability that $|C|$ is at least $k$. Note that since $f_1$ and $f_2$ are chosen from a $k$-wise independent family, each subset of at most $k$ edges will be random and independent. We will only consider subgraphs with at most $k$ edges.

To count the number of different subgraphs with $k'$ edges and at most $k' + 1$ vertices, for $k/2 \leq k' \leq k$, in a bipartite graph $G = (A, B, E)$ with $|A| = |B| = 4n$ and $|E| = n$, we count the number of ways to choose the edge labels, the vertices, and the endpoints of the edges such that they are among the chosen vertices. The $k'$ edge labels can be chosen in $\binom{n}{k'} \leq (en/k')^{k'}$ ways. Since the number of vertices in the subgraph is at most $k' + 1$, and they are chosen from $8n$ vertices in $G$, the total number of ways in which they can be chosen is bounded by $\sum_{i=1}^{k'+1} \binom{8n}{i} \leq (8en/(k'+1))^{k'+1}$. Let $k_a$ and $k_b$ be the number of vertices chosen from $A$ and $B$, respectively. The number of ways to choose an edge such that it has both its endpoints among the chosen vertices is $k_a k_b \leq ((k'+1)/2)^{2k'}$. In total, the number of different subgraphs with $k'$ edges and up to $k' + 1$ vertices is at most

$$\begin{aligned}
&(en/k')^{k'} \cdot (8en/(k'+1))^{k'+1} \cdot ((k'+1)/2)^{2k'} \\
&= \tfrac{8en}{k'+1} \cdot (2e^2 \cdot n^2 \cdot \tfrac{k'+1}{k'})^{k'} \\
&\leq \tfrac{8en}{k'+1} \cdot (\tfrac{63}{4} \cdot n^2)^{k'},
\end{aligned}$$

using $k' \geq k/2 \geq 16$.

There are in total $(4n)^{2k'}$ graphs with $k'$ specific edges. In particular, the probability that $k'$ specific edges form a particular graph is $(4n)^{-2k'}$, using $k'$-wise independence. To get an upper bound on the probability that there is some subgraph with $k'$ edges and at most $k'+1$ vertices, where $k/2 \leq k' \leq k$, we sum over all possible values of $k'$:

$$\begin{aligned}\sum_{k/2 \leq k' \leq k} \tfrac{8en}{k'+1} \cdot (\tfrac{63}{4} \cdot n^2)^{k'} \cdot (4n)^{-2k'} &= \sum_{k/2 \leq k' \leq k} \tfrac{8en}{k'+1} \cdot (\tfrac{63}{64})^{k'} \\ &\leq (k/2+1) \cdot \tfrac{8en}{k/2+1} \cdot (\tfrac{63}{64})^{k/2} \\ &= n/2^{\Omega(k)} \ .\end{aligned}$$

□

*Proof of Theorem 4.1.* We will show that the random family $\mathcal{H}_{n,c}$ of Definition 4.4 fulfills the requirements in the theorem. Assume w.l.o.g. that $\sqrt{n}$ is integer. The families $\mathcal{G} = \mathcal{SI}(U, V, \sqrt{n}, n, c, 1/2)$ and $\mathcal{F} = \mathcal{SI}(U, [4n], \sqrt{n}, n, c, 1/2)$ are both $\sqrt{n}$-wise independent with probability $1 - n^{-c}$ for sets of size up to $n$ according to Theorem 4.3. If $\mathcal{F}$ is $\sqrt{n}$-wise independent then by Lemma 4.2 the probability that the cyclic part of graph $G(f_1, f_2, S)$ has size at most $\sqrt{n}$ is at least $1 - n^{-\Omega(\sqrt{n})}$, if $\sqrt{n} \geq 32$. We can assume w.l.o.g. that $\sqrt{n} \geq 32$, since otherwise the theorem follows directly from Theorem 4.3. When the cyclic part of graph $G(f_1, f_2, S)$ has size at most $\sqrt{n}$ then, by Lemma 4.1, $\mathcal{H}_{n,c}$ is uniform on $S$ if $\mathcal{G}$ is $\sqrt{n}$-wise independent. The probability that $\mathcal{G}$ is $\sqrt{n}$-wise independent, $\mathcal{F}$ is $\sqrt{n}$-wise independent, and that the cyclic part of graph $G(f_1, f_2, S)$ has size at most $\sqrt{n}$ is altogether $(1 - n^{-c})^2(1 - n^{-\Omega(\sqrt{n})}) = 1 - O(n^{-c})$.

The construction of $\mathcal{H}_{n.c}$, i.e., constructing $\mathcal{F}$ and $\mathcal{G}$ and choosing $f_1$ and $f_2$, can according to Theorem 4.3 be done in time and space $o(n)$. The space usage of a data structure representing a function from $\mathcal{H}_{n,c}$ is $O(n)$ words for $T_1$ and $T_2$, and $o(n)$ words for storing $g \in \mathcal{G}$. The initialization time is dominated by the time used for initializing $T_1$ and $T_2$ to random arrays. Function values can clearly be computed in constant time. □

## 4.3 Applications

We now characterize a class of data structures that, when used with our hash function construction, behave exactly as if uniform hashing was used, in the sense that at any time it holds (with high probability) that the probability distribution over possible memory configurations is the same. We give a number of examples of data structures falling into this class.

**Definition 4.5** *A data structure with oracle access to a hash function $h : U \to V$ is* $n$-hash-dependent *if there is a function $f$ mapping operation sequences to subsets of $U$ of size at most $n$, such that after any sequence of operations*

*$O_1, \ldots, O_t$, the memory configuration depends only on $O_1, \ldots, O_t$, the random choices made by the data structure, and the function values of $h$ on the set $f(O_1, \ldots, O_t)$.*

The following is an immediate implication of Theorem 4.1.

**Theorem 4.4** *Consider a sequence of $n^{O(1)}$ operations on an $n$-hash-dependent RAM data structure with a random hash function oracle. For any constant $c > 0$, the oracle can be replaced by a random data structure using $O(n)$ words of space and increasing time by at most a constant factor, such that with probability $1 - O(n^{-c})$ the distribution of memory configurations after each operation remains the same.*

At first glance, the theorem concerns only what the data structure will look like, and does not say anything about the behavior of queries. However, in most cases $O(n)$-hash-dependence is maintained if we extend a data structure to write down in memory, say, the memory locations inspected during a query. Using the theorem on this data structure one then obtains that also the distribution of memory accesses during queries is preserved when using our class of hash functions.

The additional space usage of $O(n)$ words can be reduced if $U$ is much larger than $V$ by packing several $\log |V|$ bit entries of the arrays $T_1$ and $T_2$ in each $\Theta(\log |U|)$ bit word. It should be noted that although $O(n)$ words may be of the same order as the space used by the rest of the data structure, there are many cases where it is negligible. For example, if more than a constant number of words of associated information is stored with each key in a hash table, the space usage for our hash function is a vanishing part of the total space.

### 4.3.1 Examples

In the following we describe some $n$-hash-dependent hashing schemes.

**Insertion only hash tables.** One class of hash tables that are clearly $n$-hash-dependent are those that support only insertions of elements, have a bound of $n$ on the number of elements that can be inserted (before a rehash), and use $h$ only on inserted elements. This is the primary kind of scheme considered by Gonnet in [Gon84], and includes linear probing, double hashing, quadratic hashing, ordered hashing, Brent's algorithm, chained hashing, coalesced hashing, and extendible hashing.

Many such schemes are extended to support deletions by employing "deletion markers". However, as noted by Knuth [Knu98], deleting many elements in this way tends to lead to very high cost for unsuccessful searches. It thus makes sense to rebuild such data structures (with a new hash function) when the *total* number of insertions and deletions reaches some number $n$ (around the size of the hash table). If this is done, the hashing scheme remains $n$-hash-dependent.

**Deletion independent hash tables.** Some hash tables have the property that deleting an element $x$ leaves the data structure in exactly the state it would have been in if $x$ had never been inserted. In particular, the state depends exclusively on the current set of elements, the order in which they were inserted, and their hash function values. If the capacity of the hash table is bounded by $n$, such a data structure is $n$-hash-dependent.

An example of the above is a hash table using linear probing, with the deletion algorithm in [Knu98]. Also, chained hashing methods have deletion independent *pointer structure*. In particular, for those methods we get $n$-hash-dependence up to pointer structure equivalence.

**Load balancing.** A load balancing scheme of Azar et al. [ABKU99], further developed and analyzed in [BCSV00,Vöc99], can also be thought of as a hashing data structure. This scheme has been analyzed under the uniform hashing assumption. It has the property that an element in the hash table never needs to be moved once it has been placed, while at the same time, the worst case time for accessing an element remains very low.

Theorem 4.4 implies that, in the insertion only case, this data structure can be efficiently implemented such that the uniform hashing analysis holds with high probability. This means, in turn, that this is also true for the load balancing scheme.

# Chapter 5

## Low redundancy in static membership data structures with constant query time

We consider the problem of storing a subset $S$ of a finite set $U$, such that membership queries, "$u \in S$?", can be answered in worst case constant time on a unit cost RAM. We are interested only in membership queries, so we assume that $U = \{0, \ldots, m-1\}$. Also, we restrict attention to the case where the RAM has word size $\Theta(\log m)$. In particular, elements of $U$ can be represented within a constant number of machine words, and the usual RAM operations (including multiplication) on numbers of size $m^{O(1)}$ can be done in constant time.

Our goal will be to solve this, the *static membership problem*, using little memory, measured in consecutive bits. We express the complexity in terms of $m = |U|$ and $n = |S|$, and often consider the asymptotics when $n$ is a function of $m$. Since the queries can distinguish any two subsets of $U$, we need at least $\binom{m}{n}$ different memory configurations, that is, at least $B = \lceil \log \binom{m}{n} \rceil$ bits (log is base 2 throughout this chapter). We will focus on the case $n \leq m/2$ and leave the symmetry implications to the reader. Using Stirling's approximation to the factorial function, one can derive the following (where $e = 2.718\ldots$ denotes the base of the natural logarithm):

$$B = n \log(e\, m/n) - \Theta(n^2/m) - O(\log n) \ . \tag{5.1}$$

It should be noted that using space very close to $B$ is only possible if elements of $S$ are stored *implicitly*, since explicitly representing all elements requires $n \log m = B + \Omega(n \log n)$ bits.

### Previous work

The static membership data structure is very fundamental, and has been much studied. We focus on the development in space consumption for schemes with worst case constant query time. A bit vector is the simplest possible solution to the problem, but the space complexity of $m$ bits is poor compared to B unless $n \approx m/2$. By the late 70s, known membership data structures with a space complexity of $O(n)$ words (i.e., $O(n \log m)$ bits) either had non-constant query time or worked only for restricted universe sizes [CFG+78, TY79, Yao81].

The breakthrough paper of Fredman, Komlós and Szemerédi [FKS84] described a general constant time hashing scheme, from now on referred to as

the FKS scheme, using $O(n)$ words. A refined solution in the paper uses $B + O(n \log n + \log\log m)$ bits, which is $O(B)$ when $n = m^{1-\Omega(1)}$. Brodnik and Munro [BM94] constructed the first static membership data structure using $O(B)$ bits with no restrictions on $m$ and $n$. They later improved the bound to to $B + O(B/\log\log\log m)$ bits [BM99].

No non-trivial space lower bound is known in a general model of computation. However, various restrictions on the data structure and the query algorithm have been successfully studied. Yao [Yao81] showed that if words of the data structure must contain elements of $S$, the number of words necessary for $o(\log n)$ time queries cannot be bounded by a function of $n$. Fich and Miltersen [FM95] studied a RAM with standard unit cost arithmetic operations but *without division and bit operations*, and showed that query time $o(\log n)$ requires $\Omega(m/n^\epsilon)$ words of memory for any $\epsilon > 0$. Miltersen [Mil96] showed that on a RAM with bit operations but *without multiplication*, one needs $m^\epsilon$ words, for some $\epsilon > 0$, in order to achieve constant query time when $n = m^{o(1)}$.

## This chapter

In this chapter we show that it is possible to achieve space usage very close to the information theoretical minimum of $B$ bits. The additional term of the space complexity, which we will call the *redundancy*, will be $o(n) + O(\log\log m)$ bits. More precisely we show:

**Theorem 5.1** *The static membership problem with worst case constant query time can be solved using* $B + O(n\,(\log\log n)^2/\log n + \log\log m)$ *bits of storage.*

Theorem 5.1 improves the redundancy of $\Omega(\min(n\log\log m, m/(\log n)^{o(1)}))$ obtained by Brodnik and Munro by a factor of $\Omega(\min(n, \log\log m\,(\log n)^{1-o(1)}))$. For example, when $n = \Omega(m)$ we obtain space $B + B/(\log B)^{1-o(1)}$ as compared to $B + B/(\log B)^{o(1)}$. For $n = \Theta(\log m)$ our space usage is $B + n/(\log n)^{1-o(1)}$ rather than $B + \Omega(n \log n)$.

We will also show how to associate *satellite data* from a finite domain to each element of $S$, with nearly the same redundancy as above.

Our main observation is that one can save space by "compressing" the hash table part of data structures based on (perfect) hashing, storing in each cell not an element of $S$, but only a *quotient* — information that distinguishes the element from the part of $U$ that hashes to the cell[1]. This technique, referred to as quotienting, is presented in section 5.1, where a $B + O(n + \log\log m)$ bit membership data structure is exhibited. Section 5.2 outlines how to improve the dependency on $n$ to that of theorem 5.1. The construction uses a data structure supporting *rank* and *predecessor* queries, described in section 5.3. Section 5.4 gives the details of the construction and an analysis of the redundancy. The sizes of the data structures described are not computed explicitly. Rather, indirect means are employed to determine the number of redundant bits. While

[1]The term "quotient" is inspired by the use of modulo functions for hashing, in which case the integer quotient is exactly what we want in the cell.

direct summation and comparison with (5.1) would be possible, it is believed that the proofs given here contain more information about the "nature" of the redundancy.

Without loss of generality, we will assume that $n$ is greater than some sufficiently large constant. This is to avoid worrying about special cases for small values of $n$.

## 5.1 First solution

This section presents a static membership data structure with a space consumption of $B + O(n + \log\log m)$ bits. Consider a *minimal perfect hash function* for $S$, i.e., $h_{\text{perfect}} : U \to \{0, \ldots, n-1\}$ which is 1-1 on $S$. Defining an $n$-cell *hash table* $T$ such that $T[i] = s_i$ for the unique $s_i \in S$ with $h_{\text{perfect}}(s_i) = i$, the following program implements membership queries for $S$:

**function** member($x$)
  **return** ($T[h_{\text{perfect}}(x)] = x$);
**end**;

A more compact data structure results from the observation that $T[i]$ does not need to contain $s_i$ itself ($\lceil \log m \rceil$ bits), but only enough information to identify $s_i$ within $U_i = \{u \in U \mid h_{\text{perfect}}(u) = i\}$ ($\lceil \log |U_i| \rceil$ bits). We will be slightly less ambitious, though, and not necessarily go for the minimal number of bits in each hash table cell. In particular, to allow efficient indexing we want the number of bits to be the same for each table cell. Note that $\log(m/n)$ bits is a lower bound on the size of a cell, since the average size of the $U_i$ is $m/n$.

To compute the information needed in the hash table, we define a *quotient function* (for $h_{\text{perfect}}$) as a function $q : U \to \{0, \ldots, r-1\}$, $r \in \mathbf{N}$, which is 1-1 on each set $U_i$. Given such a function, let $T'[i] = q(T[i])$, and the following program is equivalent to the above:

**function** member'($x$)
  **return** ($T'[h_{\text{perfect}}(x)] = q(x)$);
**end**;

Thus it suffices to use the hash table $T'$ of $\lceil \log r \rceil$-bit entries. By the above discussion, we ideally have that $r$ is close to $m/n$.

Although the FKS scheme [FKS84] is not precisely of the form "minimal perfect hash function + hash table", it is easy to modify it to be of this type. We will thus speak of the FKS minimal perfect hash function, $h_{\text{FKS}}$. It has a quotient function which is evaluable in constant time, and costs no extra space in that its parameters $k$, $p$ and $a$ are part of the data structure already:

$$q : u \mapsto (u \text{ div } p)\, \lceil p/a \rceil + (k\,u \mod p) \text{ div } a \ . \tag{5.2}$$

Intuitively, this function gives the information that is thrown away by the mod-

ulo applications of the scheme's top-level hash function:

$$h : u \mapsto (k\,u \mod p) \mod a \tag{5.3}$$

where $k$ and $a$ are positive integers and $p > a$ is prime. We will not give a full proof that $q$ is a quotient function of $h_{\mathrm{FKS}}$, since our final result does not depend on this. However, the main part of the proof is a lemma that will be used later, showing that $q$ is a quotient function for $h$:

**Lemma 5.1** *For $U_i = \{u \in U \mid h(u) = i\}$ where $i \in \{0, \ldots, a-1\}$, $q$ is 1-1 on $U_i$. Further, $q[U] \subseteq \{0, \ldots, r-1\}$, where $r = \lceil m/p \rceil \lceil p/a \rceil$.*

*Proof.* Let $u_1, u_2 \in U_i$. If $q(u_1) = q(u_2)$ we have that $u_1 \operatorname{div} p = u_2 \operatorname{div} p$ and $(k\,u_1 \mod p) \operatorname{div} a = (k\,u_2 \mod p) \operatorname{div} a$. From the latter equation and $h(u_1) = h(u_2)$, it follows that $k\,u_1 \mod p = k\,u_2 \mod p$. Since $p$ is prime and $k \neq 0$ this implies $u_1 \mod p = u_2 \mod p$. Since also $u_1 \operatorname{div} p = u_2 \operatorname{div} p$ it must be the case that $u_1 = u_2$, so $q$ is indeed 1-1 on $U_i$. The bound on the range of $q$ is straightforward. □

In the FKS scheme, $p = \Theta(m)$ and $a = n$, so the range of $q$ has size $O(m/n)$ and $\log(m/n) + O(1)$ bits suffice to store each hash table element. The space needed to store $h_{\mathrm{FKS}}$, as described in [FKS84], is not good enough to show the result claimed at the beginning of this section. However, Schmidt and Siegel [SS90b] have shown how to implement (essentially) $h_{\mathrm{FKS}}$ using $O(n + \log\log m)$ bits of storage (which is optimal up to a constant factor, see e.g. [Meh84, Theorem III.2.3.6]). The time needed to evaluate the hash function remains constant. Their top-level hash function is not of the form (5.3), but the composition of two functions of this kind, $h_1$ and $h_2$. Call the corresponding quotient functions $q_1$ and $q_2$. A quotient function for $h_2 \circ h_1$ is $u \mapsto (q_1(u), q_2(h_1(u)))$, which has a range of size $O(m/n)$. One can thus get a space usage of $n \log(m/n) + O(n)$ bits for the hash table, and $O(n + \log\log m)$ bits for the hash function, so by (5.1) we have:

**Proposition 5.1** *The static membership problem with worst case constant query time can be solved using $B + O(n + \log\log m)$ bits of storage.*

As a by-product we get:

**Corollary 5.1** *When $n > c \log\log m / \log\log\log m$, for a suitable constant $c > 0$, the static membership problem with worst case constant query time can be solved using $n$ words of $\lceil \log m \rceil$ bits.*

*Proof.* The membership data structure of proposition 5.1 uses $n \log m - n \log n + O(n + \log\log m)$ bits. For suitable constants $c$ and $N$, the $O(n + \log\log m)$ term is less than $n \log n$ when $n > N$. If $n \leq N$ we can simply list the elements of $S$. □

The previously best result of this kind needed $n \geq (\log m)^c$ for some constant $c > 0$ [FN93] (an interesting feature of this non-constructive scheme is that it is *implicit*, i.e., the $n$ words contain a permutation of the elements in $S$). The question whether $n$ words suffice in all cases was posed in [FM95].

## 5.2 Overview of final construction

This section describes the ideas which allow us to improve the $O(n)$ term of proposition 5.1 to $o(n)$. There are two redundancy bottlenecks in the construction of the previous section:

- The Schmidt-Siegel hash function is $\Omega(n)$ bit redundant.
- The hash table is $\Omega(n)$ bit redundant.

The first bottleneck seems inherent to the Schmidt-Siegel scheme: it appears there is no easy way of improving the space usage to $1+o(1)$ times the minimum, at least if constant evaluation time is to be preserved. The second bottleneck is due to the fact that $m/n$ may not be close to a power of two, and hence the space consumption of $n \lceil \log r \rceil$ bits, where $r \geq m/n$, may be $\Omega(n)$ bits larger than the ideal of $n \log(m/n)$ bits. Our way around these bottlenecks starts with the following observations:

- We only need to solve the membership problem for some "large" subset $S_1 \subseteq S$.
- We can look at some universe $U_1 \subseteq U$, where $S_1 \subseteq U_1$ and $|U_1|/|S_1|$ is "close to" a power of 2.

The first observation helps by allowing "less than perfect" hash functions which occupy much less memory. The remaining elements, $S_2 = S \backslash S_1$, can be put in a more redundant membership data structure, namely the refined FKS scheme [FKS84]. The second observation gives a way of minimizing redundancy in the hash table.

We will use a hash function of the form (5.3). The following result from [FKS84] shows that (unless $a$ is not much larger than $n$) it is possible to choose $k$ such that $h$ is 1-1 on a "large" set $S_1$. (This is further studied in Chapter 8.)

**Lemma 5.2** *If $u \mapsto u \mod p$ is 1-1 on $S$, then for at least half the values of $k \in \{1, \ldots, p-1\}$, there exists a set $S_1 \subseteq S$ of size $|S_1| \geq (1 - O(n/a))\,|S|$, on which $h$ is 1-1.*

Without loss of generality we will assume that $|S_1| = n_1$, where $n_1$ only depends on $n$ and $a$, and $n_2 = n - n_1 = O(n/a)$. The hash function $h$ is not immediately useful, since it has a range of size $a \gg n_1$. To obtain a minimal perfect hash function for $S_1$, we compose with a function $g : \{0, \ldots, a-1\} \to \{0, \ldots, n_1 - 1\} \cup \{\perp\}$ which has $g[h[S_1]] = \{0, \ldots, n_1 - 1\}$ (in particular, it is 1-1

on $h[S_1]$). The extra value of $g$ allows us to look at $U_1 = \{u \in U \mid g(h(u)) \neq \perp\}$, which clearly contains $S_1$. We require that $g(v) = \perp$ when $v \notin h[S_1]$, since this makes $q$ a quotient function for $g \circ h$ (restricted to inputs in $U_1$).

The implementation of the function $g$ has the form of a membership data structure for $h[S_1]$ within $\{0, \ldots, a-1\}$, which apart from membership queries answers *rank queries* (the result of a rank query on input $v$ is $|\{w \in h[S_1] \mid w < v\}|$). So we may take $g$ as the function that returns $\perp$ if its input $v$ is not in the set, and otherwise returns the rank of $v$. The details on how to implement the required membership data structure are given in section 5.3.

The data structure of section 5.3 will also be our choice when $m \leq n \log^3 n$ (the "dense" case). Only when $m > n \log^3 n$ do we use the scheme described in this section. This allows us to choose suitable values of hash function parameters $p$ and $a$ (where $p = O(n^2 \log m)$ and $a = \Theta(n (\log n)^2) \leq m$), such that the range of the quotient function, $r = \lceil m/p \rceil \lceil p/a \rceil$, is close to $m/a$. The details of this, along with an analysis of the redundancy of the resulting membership data structure, can be found in section 5.4

## 5.3 Membership data structures for dense subsets

In this section we describe a membership data structure which has the redundancy stated in theorem 5.1 when $m = n (\log n)^{O(1)}$. Apart from membership queries, it will support queries on the ranks of elements (the rank of $u$ is $|\{v \in S | v < u\}|$), as well as queries on predecessors (the predecessor of $u$ is $\max\{v \in S \mid v < u\}$).

A a first step, we describe a data structure which has redundancy dependent on $m$, namely $O(m \log\log m / \log m)$ bits. The final data structure uses the first one as a substructure.

### 5.3.1 Block compression

The initial idea is to split the universe into blocks $U_i = \{b\,i, \ldots, b\,(i+1) - 1\}$ of size $b = \lceil \frac{1}{2} \log m \rceil$, and store each block in a compressed form (this is similar to the ideas of range reduction and "a table of small ranges" used in [BM99]). To simplify things we may assume that $b$ divides $m$ (otherwise consider a universe at most $b-1$ elements larger, increasing the space usage by $O(b)$ bits). If a block contains $j$ elements from $S$, the compressed representation is the number $j$ ($\lfloor \log\log m \rfloor$ bits) followed by a number in $\{1, \ldots, \binom{b}{j}\}$ corresponding to the particular subset with $i$ elements ($\lceil \log \binom{b}{j} \rceil$ bits). Extraction of information from a compressed block is easy, since any function of the block representations can be computed by table lookup (the crucial thing being that, since representations have size at most $\frac{1}{2} \log m + \log\log m$ bits, the number of entries in such a table makes its space consumption negligible compared to $O(m \log\log m / \log m)$ bits).[2]

Let $n_i = |S \cap U_i|$ and $B_i = \lceil \log \binom{b}{n_i} \rceil$. The overall space consumption of the above encoding is $\sum_i B_i + O(m \log\log m / \log m)$. Let $s$ denote the number

[2] Alternatively, assume that the RAM has instructions to extract the desired information.

of blocks, $s = O(m/\log m)$. A lemma from [BM94] bounds the above sum by $B + s$:

**Lemma 5.3** *(Brodnik-Munro) Let $m_0, \ldots, m_{s-1}$ and $n_0, \ldots, n_{s-1}$ be non-negative integers. The following inequality holds:*

$$\sum_{i=0}^{s-1} \lceil \log \tbinom{m_i}{n_i} \rceil < \log \binom{\sum_{i=0}^{s-1} m_i}{\sum_{i=0}^{s-1} n_i} + s \ .$$

*Proof.* We have $\sum_{i=0}^{s-1} \lceil \log \binom{m_i}{n_i} \rceil < \sum_{i=0}^{s-1} \log \binom{m_i}{n_i} + s \leq \log \binom{\sum_{i=0}^{s-1} m_i}{\sum_{i=0}^{s-1} n_i} + s$. The latter inequality follows from the fact that there are at least $\prod_{i=0}^{s-1} \binom{m_i}{n_i}$ ways of picking $\sum_{i=0}^{s-1} n_i$ elements out of $\sum_{i=0}^{s-1} m_i$ elements (namely by picking $n_1$ among the $m_1$ first, $n_2$ among the $m_2$ next, etc.). □

We need an efficient mechanism for extracting rank and predecessor information from the compressed representation. In particular we need a way of finding the start of the compressed representation of the $i$th block. The following result, generalizing a construction in [TY79], is used:

**Proposition 5.2** *(Tarjan-Yao) A sequence of integers $z_1, \ldots, z_k$, where for all $1 < i \leq k$ we have $|z_i| = n^{O(1)}$ and $\max(|z_i|, |z_i - z_{i-1}|) = (\log n)^{O(1)}$, can be stored in a data structure allowing constant time random access, using $O(k \log\log n)$ bits of memory.*

*Proof.* Every $\lceil \log n \rceil$th integer is stored "verbatim", using a total of $O(k)$ bits. All other integers are stored as either an offset relative to the previous of these values, or as an absolute value (one of these has size $(\log n)^{O(1)}$). This uses $O(k \log\log n)$ bits in total. □

Placing the compressed blocks consecutively in numerical order, the sequence of pointers to the compressed blocks can be stored by this method. Also, the rank of the first element in each block can be stored like this. Finally, we may store the distance to the predecessor of the first element in each block (from which the predecessor is simple to compute). All of these data structures use $O(m \log\log m/\log m)$ bits. Ranks and predecessors of elements within a block can be found by table lookup, as sketched above. So we have:

**Proposition 5.3** *A static membership data structure with worst case constant query time, supporting rank and predecessor queries, can be stored in $B + O(m \log\log m/\log m)$ bits.*

### 5.3.2 Interval compression

The membership data structure of Section 5.3.1 has the drawback that the number of compressed blocks, and hence the redundancy, grows almost linearly with

$m$. For $m \leq n\,(\log n)^c$, where $c$ is any integer constant, the number of "compressed units" can be reduced to $O(n \log\log n/\log n)$ by instead compressing *intervals* of varying length. We make sure that the compressed representations have length $(1-\Omega(1))\log n$ (so that information can be extracted by lookup in a table of negligible size) by using intervals of size $O((\log n)^{c+1})$ with at most $\log n/(2c\log\log n)$ elements. We must be able to retrieve the interval number and position within the interval for any element of $U$ in constant time. The block compression scheme of Section 5.3.1 is then trivially modified to work with intervals, and the space for auxiliary data structures becomes $O(n\,(\log\log n)^2/\log n)$ bits.

We now proceed to describe the way in which intervals are formed and represented. Let $d = \lfloor\sqrt{\log n}\rfloor$, and suppose without loss of generality that $d^{2c}$ divides $m$. (Considering a universe at most $d^{2c}$ elements larger costs $O(d^{2c})$ bits, which is negligible). Our first step is to partition $U$ into "small blocks" $U_i$, satisfying $|S \cap U_i| \leq \log n/(2c\log\log n)$. These will later be clustered to form the intervals. The main tool is the membership data structure of Proposition 5.3, which is used to locate areas with a high concentration of elements from $S$. More specifically, split $U$ into at most $n$ blocks of size $d^{2c}$ and store the indices of blocks that are *not* small, i.e., contain more than $\log n/(2c\log\log n)$ elements from $S$. Since at most $2cn\log\log n/\log n$ blocks are not small, the memory for this data structure is $O(n\,(\log\log n)^2/\log n)$ bits. The part of the universe contained in non-small blocks has size at most $2cm\log\log n/\log n \leq n\,d^{2c-1}$. A rank query can be used to map the elements of non-small blocks injectively and in an order preserving way to a sub-universe of this size. The splitting is repeated recursively on this sub-universe, now with at most $n$ blocks of size $d^{2c-1}$. Again, the auxiliary data structure uses $O(n\,(\log\log n)^2/\log n)$ bits. At the bottom of the recursion we arrive at a universe of size at most $nd$, and every block of size $d$ is small. This defines our partition of $U$ into $O(n)$ small blocks, which we number $0, 1, 2, \ldots$ in order of increasing elements. Note that the small block number of any element in $U$ can be computed by a rank query and a predecessor query at each level.

As every small block has size at most $(\log n)^c$, intervals can be formed by up to $\log n$ consecutive small blocks, together containing at most $\log n/(2c\log\log n)$ elements of $S$. The "greedy" way of choosing such compressible intervals from left to right results in $O(n\log\log n/\log n)$ intervals, as no two adjacent intervals can both contain less than $\log n/(4c\log\log n)$ elements and be shorter than $(\log n)^{c+1}$. To map the $O(n)$ block numbers to interval numbers, we use the membership data structure of Proposition 5.3 to store the number of the first small block in each interval, using $O(n\,(\log\log n)^2/\log n)$ bits. A rank query on a small block number then determines the interval number. Finally, the first element of each interval is stored using Proposition 5.2, allowing positions within intervals to be computed, once again using $O(n\,(\log\log n)^2/\log n)$ bits.

**Theorem 5.2** *For $m = n\,(\log n)^{O(1)}$, a static membership data structure with worst case constant query time, supporting rank and predecessor queries, can be stored in $B + O(n\,(\log\log n)^2/\log n)$ bits.*

## 5.4 Membership data structures for sparse subsets

In this section we fill out the remaining details of the construction described in section 5.2, and provide an analysis of the redundancy obtained. By section 5.3 we need only consider the case $m > n\,(\log n)^c$ for some constant $c$ (it will turn out that $c = 3$ suffices).

### 5.4.1 Choice of parameters

We need to specify how hash function parameters $a$ and $p$ are chosen (a choice of $k$ then follows by lemma 5.2). Parameter $a$ will depend on $p$, but is bounded from above by $A(n)$ and from below by $A(n)/3$, where $A$ is a function we specify later. For now, let us just say that $A(n) = n\,(\log n)^{\Theta(1)}$ (our construction requires $A(n) = n\,(\log n)^{O(1)}$, and we want $A(n)$ large in order to make $S\backslash S_1$ small). Parameter $p$ will have size $O(n^2 \log m)$, so it can be stored using $O(\log n + \log\log m)$ bits. It is chosen such that $u \mapsto u \mod p$ is 1-1 on $S$, and such that $r = \lceil m/p \rceil \lceil p/a \rceil$ is not much larger than $m/a$.

**Lemma 5.4** *For $m$ larger than some constant, there exists a prime $p$ in each of the following ranges, such that $u \mapsto u \mod p$ is 1-1 on $S$:*

1. $n^2 \ln m \le p \le 3\,n^2 \ln m$ .
2. $m \le p \le m + m^{2/3}$ .

*Proof.* The existence of a suitable prime between $n^2 \ln m$ and $3\,n^2 \ln m$ is guaranteed by the prime number theorem (in fact, at least half of the primes in the interval will work). See [FKS84, Lemma 2] for details. By [HBI79] the number of primes between $m$ and $m + m^\theta$ is $\Omega(m^\theta / \log m)$ for any $\theta > 11/20$. Take $\theta = 2/3$ and let $p$ be such a prime; naturally the map is then 1-1. □

A prime in the first range will be our choice for $p$ when $m > A(n)\,n^2 \ln m$, otherwise we choose a prime in the second range. In the first case, $r < (m/p + 1)(p/a + 1) = (1 + a/p + p/m + a/m)\,m/a$. In the second case, $r = \lceil p/a \rceil < (m + m^{2/3})/a + 1 \le (1 + a/m + m^{-1/3})\,m/a$. Since we can assume $m > a \log n$, we have in both cases that $r = (1 + O(1/\log n))\,m/a$. We make $r$ close to a power of 2 by suitable choice of parameter $a$.

**Lemma 5.5** *For any $x, y \in \mathbf{R}_+$ and $z \in \mathbf{N}$, with $x/z \ge 3$, there exists $z' \in \{z+1, \ldots, 3z\}$, such that $\lceil \log \lceil x/z' \rceil + y \rceil \le \log(x/z') + y + O(z/x + 1/z)$.*

*Proof.* Since $x/z \ge 3$, it follows that $\log\lceil x/z \rceil + y$ and $\log\lceil x/3z \rceil + y$, have different integer parts. So there exists $z'$, $z < z' \le 3z$, such that $\lceil \log \lceil x/z' \rceil + y \rceil \le \log\lceil x/(z'-1) \rceil + y$. A simple calculation gives $\log\lceil x/(z'-1) \rceil + y = \log(x/(z'-1)) + y + O(z/x) = \log(x/z') + \log(z'/(z'-1)) + y + O(z/x) = \log(x/z') + y + O(z/x + 1/z)$, and the conclusion follows. □

Since $\log r = \log\lceil p/a\rceil + \log\lceil m/p\rceil$ and $p/A(n) \geq 3$ (for $n$ large enough), the lemma gives an $a$ satisfying $A(n)/3 \leq a \leq A(n)$, such that $\lceil \log r\rceil = \log r + O(a/p + 1/a) = \log((1 + O(1/\log n))\, m/a)$.

To conclude, we can choose $p$ and $a$ such that the number of bit patterns in each hash table cell, $2^{\lceil \log r\rceil}$, is $(1 + O(1/\log n))\, m/a$.

### 5.4.2 Storing parameters

A slightly technical point remains, concerning the storage of parameters in the data structure. If the universe size $m$ is supposed to be implicitly known, there is no problem storing the parameters using $O(\log n + \log\log m)$ bits (say, using $O(\log\log m)$ bits to specify the number of bits for each parameter). However, if $m$ is considered a parameter unknown to the query algorithm, it is not clear how to deal with e.g. queries for numbers larger than $m$, without actually using $O(\log m)$ extra bits to store $m$. Our solution is to look at a slightly larger universe $U'$, whose size is specified using $O(\log n + \log\log m)$ bits. Using $O(\log n + \log\log m)$ bits we may store $\lceil \log m\rceil$ (by assumption we know the number of bits needed to store this number within an additive constant) and the $\lceil \log n\rceil$ most significant bits of $m$. This defines $m' = (1 + O(1/n))\, m$, the universe size of $U'$. We need to estimate the information theoretical minimum of the new problem, $B' = \lceil \binom{m'}{n}\rceil$:

**Lemma 5.6** *For $n < m_1 < m_2$ we have $\log \binom{m_2}{n} - \log \binom{m_1}{n} < n \log(\frac{m_2 - n}{m_1 - n})$.*

*Proof.* We have $\binom{m_2}{n}/\binom{m_1}{n} = \frac{m_2\,(m_2-1)\ldots(m_2-n+1)}{m_1\,(m_1-1)\ldots(m_1-n+1)} < (\frac{m_2-n}{m_1-n})^n$. □

Thus, since $n \leq m/2$, $B' = B + O(n \log(m'/m)) = B + O(n/\log n)$. So our slight expansion of the universe is done without affecting the redundancy of theorem 5.1.

### 5.4.3 Redundancy analysis

First note that we can assume all parts of the data structure to have size depending only on $m$ and $n$ (that is, not on the particular set stored). Hence, the entire data structure is a bit pattern of size $B + f(n, m)$, for some function $f$. To show the bound $f(n, m) = O(n\,(\log\log n)^2/\log n + \log\log m)$, we construct a function $\phi$, mapping $n$-element subsets of $U$ to subsets of $\{0,1\}^{B+f(n,m)}$, such that:

- $\log|\phi(S)| = O(n\,(\log\log n)^2/\log n + \log\log m)$.
- $\bigcup_S \phi(S) = \{0,1\}^{B+f(n,m)}$ .

This implies $B + f(n, m) \leq \log(\sum_S |\phi(S)|) = B + O(n\,(\log\log n)^2/\log n + \log\log m)$, as desired. Recall that the data structure consists of:

- Hash function parameters and pointers ($b_1 = O(\log n + \log\log m)$ bits).

- A membership data structure supporting rank, representing the function $g$ via a set of $n_1$ elements in $\{0,\ldots,a{-}1\}$ ($b_2 = \log\binom{a}{n_1} + O(n\,(\log\log n)^2/\log n)$ bits).
- A hash table ($b_3 = n_1\,\lceil\log r\rceil$ bits).
- A membership data structure representing a set of size $n_2$ in $U$ ($b_4 = \log\binom{m}{n_2} + O(n_2 \log n_2 + \log\log m)$ bits).

Since the redundancy of the membership data structure supporting rank is $O(n\,(\log\log n)^2/\log n)$, there exists a function $\phi'$ from the $n_1$-element subsets of $\{0,\ldots,a-1\}$ to $\{0,1\}^{b_2}$, such that $\cup_{\tilde{S}_1}\phi'(\tilde{S}_1) = \{0,1\}^{b_2}$ and $\log|\phi'(\tilde{S}_1)| = O(n\,(\log\log n)^2/\log n)$. Similarly, there exists a function $\phi''$ from the $n_2$-element subsets of $U$ to $\{0,1\}^{b_4}$, such that $\cup_{\tilde{S}_2}\phi''(\tilde{S}_2) = \{0,1\}^{b_4}$ and $\log|\phi''(\tilde{S}_2)| = O(n_2\log n_2 + \log\log m)$. Choosing $A(n) = n\log^2 n$ we have $n_2 = O(n/\log^2 n)$, and hence $\log|\phi''(\tilde{S}_2)| = O(n/\log n + \log\log m)$.

Let $h_1$ denote the hash function $u \mapsto (u \mod p) \mod a$ (any hash function of the form (5.3) would do, we pick this one for simplicity). By lemma 5.6 we can assume that $p$ divides $m$, since either: $m \le p \le m + m^{2/3}$, in which case expanding the universe to $\{0,\ldots,p-1\}$ increases the information theoretical minimum by $O(n/m^{1/3})$; or $p = O(m/n)$, in which case the increase by rounding $m$ to the nearest higher multiple of $p$ is $O(1)$.

When $p$ divides $m$, the number of elements hashed to a cell by $h_1$ is at least $\lfloor m/a\rfloor$. Hence for any function $g$, there is a set $T(g)$ of at least $(m/a-1)^{n_1}$ possible bit patterns in the hash table. The total number of bit patterns is $2^{\lceil\log r\rceil\,n_1} = ((1+O(1/\log n))\,m/a)^{n_1}$, so the ratio between this and the $|T(g)|$ patterns used is:

$$\left(\frac{1+O(1/\log n)}{1-a/m}\right)^{n_1} = (1+O(1/\log n))^{n_1} = 2^{O(n/\log n)}\ .$$

Thus, there exists a function $\phi_g$ from $T(g)$ onto $\{0,1\}^{b_3}$, such that $\log|\phi_g(z)| = O(n/\log n)$. For notational convenience we will from now on denote bit patterns in the hash table simply by the corresponding set of universe elements.

We will take $\phi(S)$ as the union of sets $\phi(\tilde{S}_1,\tilde{S}_2)$, over all $\tilde{S}_1,\tilde{S}_2 \subseteq S$ with $|\tilde{S}_1| = n_1$ and $|\tilde{S}_2| = n_2$. If $|h_1[\tilde{S}_1]| \neq n_1$, we set $\phi(\tilde{S}_1,\tilde{S}_2) = \emptyset$. Otherwise $h_1[\tilde{S}_1]$ defines the function $g$, and we set:

$$\phi(\tilde{S}_1,\tilde{S}_2) = \{s_1\,s_2\,s_3\,s_4 \mid s_1 \in \{0,1\}^{b_1},\ s_2 \in \phi'(h_1[\tilde{S}_1]),\ s_3 \in \phi_g(\tilde{S}_1),\ s_4 \in \phi''(\tilde{S}_2)\}\ .$$

By our bounds on the sizes of $\phi'(h_1[\tilde{S}_1])$, $\phi_{h_1,g}(\tilde{S}_1)$ and $\phi''(\tilde{S}_2)$, we conclude that $\log|\phi(\tilde{S}_1,\tilde{S}_2)| = O(n\,(\log\log n)^2/\log n + \log\log m)$. Since $\phi(S)$ is the union of the $2^{O(n/\log n)}$ sets of the form $\phi(\tilde{S}_1,\tilde{S}_2)$, it follows that the requirement on $|\phi(S)|$ holds.

To see that $\bigcup_S \phi(S) = \{0,1\}^{B+f(n,m)}$, take any $x \in \{0,1\}^{B+f(n,m)}$. Let $x = s_1\,s_2\,s_3\,s_4$, where $s_i \in \{0,1\}^{b_i}$. By definition of $\phi'$ and $\phi''$, there is some set $T \subseteq \{0,\ldots,a-1\}$, $|T| = n_1$, such that $s_2 \in \phi'(T)$, and a set $\tilde{S}_2 \subseteq U$, $|\tilde{S}_2| = n_2$, such that $s_4 \in \phi''(S_2)$. The set $T$ corresponds to a function $g$. From the way we defined $\phi_g$, there exists a bit pattern $z \in T(g)$, such that $s_3 \in \phi_g(z)$. Let $\tilde{S}_1$ be the set of size $n_1$ corresponding to $h_1$, $g$ and $z$. We then have that $x \in \phi(\tilde{S}_1,\tilde{S}_2)$, so if we take $S \supseteq \tilde{S}_1 \cup \tilde{S}_2$, we get $x \in \phi(S)$ as desired.

## 5.5 Dictionaries

We now discuss how to associate information with elements, solving the *dictionary* problem. More specifically, we consider the setting where each element of $S$ has an associated piece of satellite information from some set $V = \{0, \ldots, s-1\}$, where $s = m^{O(1)}$. The information theoretical minimum for this problem is $B_s = \lceil \log \binom{m}{n} + n \log s \rceil$.

The quotienting technique generalizes to this setting. We simply extend the quotient function to take an extra parameter from $V$ as follows: $q'(u, v) = q(u) + r\, v$. Note that from $q'(u, v)$ it is easy to compute $q(u)$ and $v$, and that the range of $q'$ has size $r\, s$. With this new quotient function, the remaining parts of the construction for $m > n\,(\log n)^3$ are unchanged.

In the dense range, the membership data structure supporting rank can be used to index into a table of $V$-values, but in general $\Omega(n)$ bits will be wasted in the table since $|V|$ need not be a power of 2. Thus we have:

**Theorem 5.3** *The static dictionary problem with worst case constant query time can be solved with storage:*

- $B_s + O(n\,(\log\log n)^2/\log n + \log\log m)$ *bits, for* $m > n\,(\log n)^3$ .
- $B_s + O(n)$ *bits, otherwise.*

Using the data structure for the sparse case, it is in fact possible to achieve redundancy $o(n)$ when $n = o(m)$. The dictionary of proposition 5.1 is then used to store $S_2$, and parameter $a$ is chosen around $\sqrt{nm}$.

## 5.6 Construction

We now sketch how to construct the static membership data structures and dictionaries in expected time $O(n + (\log\log m)^{O(1)})$. The data structures of section 5.3 can in fact be constructed in time $O(n)$ when $m = n^{O(1)}$. The construction algorithm is quite straightforward, so we do not describe it here. As for the dictionary described in sections 5.2 and 5.4, the hardest part is finding appropriate parameters for the hash function. Once this is done, the dictionary for $h[S_1]$, the hash table, and the dictionary for $S_2$ can all be constructed in expected time $O(n)$ (see [FKS84] for the latter construction algorithm).

The prime $p$ is found by randomly choosing numbers from the appropriate interval of lemma 5.4. Each number chosen is checked for primality (using a probabilistic check which uses expected time poly-logarithmic in the number checked [AH87], that is, time $(\log n + \log\log m)^{O(1)}$). When a prime is found, it is checked whether $u \mapsto u \mod p$ is 1-1 on $S$ (the element distinctness problem on the residues, taking expected $O(n)$ time using universal hashing). The process is repeated until this is the case. Inspecting the proof of lemma 5.4 it can be seen that the expected number of iterations is $O(1)$, so the expected total time is $O(n + (\log\log m)^{O(1)})$.

Parameter $a$ is simple to compute according to lemma 5.5, for example by binary search on the interval in which it is wanted.

Parameter $k$ is tentatively chosen at random and checked in time $O(n)$ for the inequality of lemma 5.2, with some constant $c$ in the big-oh. For sufficiently large $c$, the expected number of attempts made before finding a suitable $k$ is constant, and thus the expected time for the choice is $O(n)$.

**Theorem 5.4** *The data structure of theorem 5.1 can be constructed in expected time* $O(n + (\log\log m)^{O(1)})$.

## 5.7 Conclusion and final remarks

We have seen that for the static dictionary problem it is possible to come very close to using storage at the information theoretic minimum, while retaining constant query time. From a data compression point of view this means that a sequence of bits can be coded in a number of bits close to the first-order entropy, in a way that allows efficient random access to the original bits.

The important ingredient in the solution is the concept of quotienting. Quotienting was recently applied in a space efficient dictionary supporting rank [RR99]. In general, quotienting can be used to save around $n \log n$ bits in hash tables. Thus, the existence of an efficiently evaluable quotient function is a desirable property for a hash function. For the quotient function to have a small range, it is necessary that the hash function used hashes $U$ quite evenly to the entire range.

Quotienting works equally well in a dynamic setting, where it can be used directly to obtain an $O(B)$ bit scheme, equaling the result of Brodnik and Munro [BM94]. However, lower bounds on the time for maintaining ranks under insertions and deletions (see [FS89]) show that our construction involving the dictionary supporting rank will not dynamize well.

It would be interesting to determine the exact redundancy necessary to allow constant time queries. In particular, it is remarkable that no lower bound is known in the *cell probe* model (where only the number of memory cells accessed is considered). As for upper bounds, a less redundant implementation of the function $g$ would immediately improve the asymptotic redundancy of our scheme. There seems to be no hope of getting rid of the $O(\log\log m)$ term using our basic approach, since any hash function family ensuring that some function is 1-1 on a "large" subset of $S$, has size $\Omega(\log m)$, see Chapter 8.

# Chapter 6

## Deterministic dictionaries

*Dictionaries* are among the most fundamental data structures. A dictionary stores a subset $S$ of a universe $U$, offering membership queries of the form "Is $x \in S$?" for $x \in U$. It also supports the retrieval of *satellite data* associated with the elements of $S$, which are called *keys*. One distinguishes between the *dynamic* case, where the dictionary supports insertion and deletion of keys (with their satellite data), and the *static* case, where $S$ does not change over time.

Several performance measures are of interest for dictionaries: the amount of space occupied by a dictionary, the time needed to construct or update it, and the time needed to answer a query. In this chapter, our primary interest lies in obtaining static dictionaries with optimal query time and minimal space consumption that can be constructed rapidly by deterministic algorithms. By general dynamization results, this also has implications for deterministic dynamic dictionaries.

Our model of computation is the unit-cost *word RAM*. This natural and realistic model of computation has been the object of much recent research, surveyed in [Hag98b], which also offers a detailed definition. For a positive integer parameter $w$, called the *word length*, the memory cells of a word RAM store $w$-bit *words*, variously viewed as integers in $\{0, \ldots, 2^w - 1\}$ or as bit vectors in $\{0,1\}^w$, and standard operations can be carried out on words in constant time. We adopt the *multiplication model*, whose instruction set includes addition, bitwise boolean operations, shifts, and multiplication and measure the space requirements of a word-RAM algorithm in units of $w$-bit words. Word-RAM algorithms can be *weakly nonuniform*, that is, access a fixed number of word-size constants that depend (only) on $w$. These constants, which we call *native constants*, may be thought of as computed at "compile time".

The keys to be stored in a dictionary are assumed to be representable in single words, i.e., to come from the universe $U = \{0,1\}^w$. For simplicity, we assume that each piece of satellite data occupies a single word of memory (if necessary, it can be a pointer to more bulky data).

Denoting the number of keys by $n$, we see that constant query time and $O(n)$ space is the best for which one can hope. A seminal result of Fredman et al. [FKS84] states that in the static case, a dictionary with these properties, henceforth referred to as *efficient*, is indeed possible. To achieve fast construction of the dictionary, Fredman et al. augment their RAM model with

an additional resource: a source of random bits. In this setting, there is a construction algorithm with expected running time $O(n)$. This efficient dictionary and its construction algorithm are known as the *FKS scheme*.

The main result of this chapter is an alternative efficient dictionary that can be constructed deterministically in $O(n \log n)$ time. A standard dynamization technique yields a range of combinations of lookup time and update time. For example, we achieve constant lookup time with update time $O(n^\epsilon)$, for arbitrary constant $\epsilon > 0$.

## Related work

Efficient dictionaries have been known for a long time for certain combinations of $n$ and $w$. For $n = \Omega(2^w)$, e.g., a bit vector does the job. Tarjan and Yao [TY79] showed how to construct efficient static dictionaries when the number of keys and the size of the universe are polynomially related, i.e., when $w = O(\log n)$. As already mentioned, Fredman et al. demonstrated how to build efficient static dictionaries for arbitrary word sizes. Besides the randomized construction running in expected time $O(n)$, they gave a deterministic one with a running time of $O(n^3 w)$. A bottleneck in the deterministic algorithm is the choice of appropriate hash functions. It can be shown that exhaustive search in any *universal* class of hash functions [CW79] yields suitable functions. A more efficient way of conducting the search was devised by Raman [Ram96], who lowered the deterministic construction time to $O(n^2 w)$. For $w = n^{\Omega(1)}$, a static dictionary for $n$ keys can be constructed deterministically in $O(n)$ time plus the time needed to sort the keys. This follows from a straightforward generalization of the fusion trees of Fredman and Willard [FW93] and was stated explicitly by Hagerup [Hag98b, Corollary 8]. Alon and Naor [AN96] used small-bias probability spaces to derandomize a variant of the FKS scheme, achieving construction time $O(nw(\log n)^4)$. However, their lookup operation requires evaluation of a linear function in time $\Theta(w/\log n)$, so the dictionary is not efficient unless $w = O(\log n)$. Another variant of the FKS scheme reduces the number of random bits to $O(\log n + \log w)$, while achieving $O(n)$-time construction with high probability [DfGMP92].

Allowing randomization, the FKS scheme can be dynamized to support insertions and deletions in amortized expected constant time [DfKM$^+$94]. Without a source of random bits, the task of simultaneously achieving fast updates and constant query time seems considerably harder, and no solution with nontrivial performance bounds was previously known. Also, it is shown in [DfKM$^+$94] that approaches similar to the double hashing of the FKS scheme are destined to perform poorly in a deterministic setting. The best result when the update and query times are considered equally important is time $O(\sqrt{\log n / \log\log n})$ per dictionary operation; it uses the data structure of Beame and Fich [BF99a] with the dynamization result of Andersson and Thorup [AT00]. A different trade-off, lookup time $O((\log\log n)^2/\log\log\log n)$ and update time $O((\log n \log\log n)^2)$, is obtained in Chapter 7. For $w = n^{\Omega(1)}$, a standard dynamization of the fusion-tree-like data structure mentioned above provides constant-time lookups with update time $O(n^\epsilon)$, for arbitrary fixed $\epsilon > 0$.

An unpublished manuscript by Sundar [Sun93] states an amortized lower bound of $\Omega(\log\log_w n/\log\log\log_w n)$ per operation for a dynamic dictionary in the cell-probe model for $w = \Omega(\log n \log\log n)$; this bound, in particular, implies the same lower bound on the word RAM.

Andersson et al. [AMRT96] have shown that a unit-cost RAM that allows efficient dictionaries must have an instruction of circuit depth $\Omega(\log w/\log\log w)$. Since this matches the circuit depth of multiplication, we see that efficient dictionaries are not possible with weaker instruction sets (in the circuit-depth sense). However, some work has been done on minimizing the query time in weaker models. In a word-RAM model providing only $AC^0$ instructions, there is a tight bound of $\Theta(\sqrt{\log n/\log\log n})$ on the query time [AMRT96] (a simpler proof of the upper bound appears in [Hag98a]). The algorithm of the upper bound uses nonstandard instructions. In a restricted word-RAM model that lacks multiplication, the best upper bound is $\sqrt{\log n(\log\log n)^{1+o(1)}}$, due to Brodnik et al. [BMM97].

## Our contributions

In this chapter we sum up results contained in three consecutive conference publications. In [Mil98], Miltersen showed how error-correcting codes can be used to construct an efficient dictionary in time $O(n^{1+\epsilon})$, for arbitrary constant $\epsilon > 0$. Combining this approach with the use of word parallelism, Hagerup [Hag99] exhibited a dictionary with $O(\log\log n)$ lookup time and $O(n\log n)$ construction time. Our main theorem was derived in [Pag00a], which added as a new ingredient an improved construction algorithm for a class of perfect hash functions introduced by Tarjan and Yao.

**Theorem 6.1** *A static dictionary for $n$ $w$-bit keys and their satellite data with constant lookup time and a space consumption of $O(n)$ memory words can be constructed in $O(n\log n)$ time on a word RAM with word length $w$ by a weakly nonuniform deterministic algorithm that uses $O(n)$ words of memory.*

The static dictionary can be turned into a dynamic one, supporting insertions and deletions, by a standard dynamization result [OvL81b, Theorem A].

**Theorem 6.2** *Let $t(n) = O(\sqrt{\log n})$ be a nondecreasing function from $\mathbb{N}$ to $\mathbb{N}$ such that $t(n)$ is computable in time and space $O(n)$ and $t(2n) = O(t(n))$. Then there is a weakly nonuniform deterministic dynamic dictionary that runs on a word RAM with word length $w$ and, when $n$ elements are stored, uses $O(n)$ words of memory and supports lookups in time $O(t(n))$, insertions in time $O(n^{1/t(n)})$, and deletions in time $O(\log n)$.*

The theorem is most interesting when $t$ grows slowly. In particular, no previous deterministic linear-space dictionary combined lookup time $O(\log\log n)$ with update time $o(n)$.

It should be noted that we make heavy use of weak nonuniformity. Whereas it is common to employ native constants that can be computed in $O(w)$ or even $O(\log w)$ time, our data structures depend on native constants that are not known to be computable in $w^{O(1)}$ time.

## 6.1 Technical overview

Let $S \subseteq U$ denote the set of keys to be stored and take $n = |S|$. In order to prove Theorem 6.1, we show how to construct a function $h : U \to \{0, \ldots, m-1\}$, where $m = O(n)$, that is 1-1 on $S$ and can be stored in constant space and evaluated in constant time. Informally, such a function will be called an *efficient perfect hash function* for $S$. The desired efficient static dictionary consists of the description of $h$ together with a hash table of size $m$.

Our approach is to first perform a *universe reduction* by finding a function $\rho : U \to \{0,1\}^r$, with $r = O(\log n)$, that is 1-1 on $S$ and can be stored in constant space and evaluated in constant time. Then an efficient perfect hash function $h'$ is found for $\rho(S) \subseteq \{0,1\}^r$. The desired function is $h = h' \circ \rho$.

The universe reduction is based on error-correcting codes, whose use in the context of hashing is introduced in Section 6.2. By applying an error-correcting code $\psi$, replacing each element $x \in U$ by $\psi(x) \in \{0,1\}^{O(w)}$, the Hamming distance (the number of differing bit positions) between any two elements of $U$ can be made $\Omega(w)$. It is then possible to find a set $D$ of $O(\log n)$ *distinguishing bit positions* such that for every pair $\{x, y\}$ of distinct keys in $S$, $\psi(x)$ and $\psi(y)$ differ on $D$. Exploiting word parallelism, we show how to find such distinguishing bit positions in $O(n \log n)$ time. Given these, Raman's deterministic selection of perfect hash functions [Ram96] can be used to construct a function $\rho$ mapping to the desired range $\{0,1\}^r$. We further show that a good error-correcting code can be picked from a universal family of functions from $U$ to $\{0,1\}^{O(w)}$. Since there are such families whose functions can be evaluated in constant time, we obtain an error-correcting code with the same property. The choice of an appropriate function is a source of weak nonuniformity.

In Section 6.3 we show how to find an efficient perfect hash function for $\rho(S) \subseteq \{0,1\}^r$. We first develop a randomized variant of the efficient perfect hash function of Tarjan and Yao. The construction algorithm is then derandomized using conditional expectations, yielding an $O(n \log n)$-time deterministic algorithm. Our algorithm is quite simple compared with the $O(n(\log n)^5)$-time algorithm described in [AN96].

## 6.2 Universe reduction

In this section we describe the construction of a universe-reduction function $\rho : U \to \{0,1\}^r$, where $r = O(\log n)$. The prime feature of $\rho$ is that it is 1-1 on $S$. Because of this, it may be used to "translate" a search for $x \in U$ into a search for $\rho(x)$ within the smaller universe $\{0,1\}^r$. Since we are interested in constant-time queries, $\rho$ should be evaluable in constant time. Constant space will suffice to store the description of $\rho$.

### 6.2.1 Distinguishing bit positions

Let $\psi : U \to \{0,1\}^{4w}$ be an error-correcting code of relative minimum distance $\delta > 0$. This means that for every pair $\{x, y\}$ of distinct elements of $U$, the Hamming distance between $\psi(x)$ and $\psi(y)$ is at least $4\delta w$. We assume that

$\delta \leq 1/2$ (in fact, by the Plotkin bound for error-correcting codes [MS77, p. 41], this is always the case for $w > 2$). Denote the $i$th bit of a bit string $v$ (counted from the right, say) by $v_i$. We have the following:

**Lemma 6.1** *For every subset $S$ of $U$ of size $n$, there is a set $D \subseteq \{1, \ldots, 4w\}$ with $|D| \leq 2\log(n)/\log\frac{1}{1-\delta}$ such that for every pair $\{x, y\}$ of distinct elements of $S$, $\psi(x)_d \neq \psi(y)_d$ for some $d \in D$.*

*Proof.* A simple proof of the lemma proceeds by showing that picking the bit positions independently at random satisfies the condition of the lemma with positive probability. We provide a slightly different argument that is more easily turned into an efficient algorithm.

We will construct a sequence of sets $D_0 = \emptyset \subseteq D_1 \subseteq \cdots \subseteq D_k \subseteq \{1, \ldots, 4w\}$ that are increasingly better at distinguishing elements of $S$. For a set $D \subseteq \{1, \ldots, 4w\}$, we split $S$ into $2^{|D|}$ disjoint *clusters* $C(S, D, 0^{|D|}), \ldots, C(S, D, 1^{|D|})$, one for each possible vector of bit values at the positions given by $D$. Define the *badness* of $D$ as $B(S, D) = \sum_{v \in \{0,1\}^{|D|}} \binom{|C(S,D,v)|}{2}$, which is the number of pairs within the clusters. We will determine our sets such that $|D_i| \leq i$ and $B(S, D_i) < (1 - \delta)^i n^2/2$ for $i = 0, \ldots, k$, a condition that clearly holds for $i = 0$ with $D_0 = \emptyset$. Assume that $D_i$ has been found for some $i$ with $0 \leq i < k$. A pair of distinct elements in some cluster $C(S, D_i, v)$ is also in $C(S, D_i \cup \{d\}, v')$, for some $v'$, for at most a fraction of $1 - \delta$ of the possible choices of $d \in \{1, \ldots, 4w\}$. By an averaging argument, it is possible to choose $d$ such that $B(S, D_i \cup \{d\}) \leq (1 - \delta)B(S, D_i)$, and we let $D_{i+1} = D_i \cup \{d\}$ for such a $d$. Setting $k = \lfloor 2\log_{1/(1-\delta)} n \rfloor$, we achieve $B(S, D_k) < 1$, so we can take $D_k$ as the desired set of distinguishing bits. □

The lemma shows that a very simple hash function with polynomial-sized range can be found that it is 1-1 on the error-corrected representations of the keys: Simply use the projection on $O(\log n)$ suitable bit positions.

We next make the proof of the lemma constructive by giving an algorithm for actually finding a small set $D$ of distinguishing bit positions. When choosing a position, we need only care about the *nontrivial* clusters, those of size at least 2, since smaller clusters do not contribute to the badness. Maintaining the nontrivial clusters under addition of new bit positions is easy: Simply keep a linked list for each cluster; the lists can be split with respect to the bit value at a new position in a linear pass. Therefore the task of finding distinguishing positions boils down to that of finding a single *good* bit position, one that decreases the badness by a factor of at least $1 - \delta$.

**Lemma 6.2** *Let $S$ be a subset of $U$ of size $n$. Given linked lists of the nontrivial clusters of $S$ corresponding to distinguishing positions $D$, a bit position $d$ with*

$$B(S, D \cup \{d\}) \leq (1 - \delta)\, B(S, D)$$

*can be found deterministically in time and space $O(n)$.*

*Proof.* We show how to efficiently compute the badness $B(S, D \cup \{d\})$ for each $d \in \{1, \ldots, 4w\}$. The bit position $d$ with the smallest badness must, by the proof

of Lemma 6.1, satisfy $B(S, D \cup \{d\}) \leq (1-\delta)B(S, D)$. For each $d \in \{1, \ldots, 4w\}$, we perform the following steps:

1. For each nontrivial cluster $C$, compute $s_C = \sum_{x \in C} x_d$, the number of 1s in position $d$.

2. For each nontrivial cluster $C$, compute $z_C = \binom{s_C}{2} + \binom{|C|-s_C}{2}$, the combined badness of the two clusters resulting from $C$ if $d$ is included in $D$.

3. Compute the total badness $\sum_C z_C$ over all (nontrivial) clusters $C$.

It is an easy matter to execute steps 1–3 in $O(n)$ time for a single value of $d$. In order to execute steps 1–3 for all $d \in \{1, \ldots, 4w\}$ within the same time bound, we resort to word-level parallelism, viewing each string of $4w$ bits as a bit vector. We first describe the algorithm under the following (unrealistic) assumptions:

A. Prior to the execution, each bit vector representing an element of $S$ is "stretched" by a sufficiently large factor $f$ through the introduction of $f - 1$ zeros to the left of each original bit. This turns each original bit position into a *field* of $f$ consecutive bit positions.

B. The machine instructions applicable to words can also be applied to vectors of $4w$ fields and still take constant time.

Under these assumptions, it is trivial to execute steps 1 and 3 in $O(n)$ time for all $d \in \{1, \ldots, 4w\}$: The only operation needed is field-wise addition, which can be realized through ordinary word-level addition. The field width $f$ is assumed to be sufficiently large to prevent overflows between fields. Step 2 needs the following additional operations:

- Replication of a value $m$, stored in the rightmost field, to all other fields (used with $m = |C|$ and $m = 1$). This can be done by multiplying $m$ by the constant $1_f$ that contains 1 in every field. For the time being, we assume $1_f$ to be a native constant.

- Field-wise subtraction with a nonnegative result, which can be realized through word-level subtraction.

- Field-wise multiplication. This can be realized through the usual shift-and-add algorithm that successively tests each bit of one factor and, if it is 1, adds an appropriately shifted copy of the other factor to an accumulated sum. We refer to [AHNR98, Sect. 3] for a description of the low-level details needed to carry out such steps as field-wise conditional addition based on a comparison with zero, noting only that the constant $1_f$ comes in handy here as well. The time needed is $O(f)$.

- Field-wise division of even integers by 2, which can be realized through a right shift.

Since the field values manipulated by the algorithm are polynomial in $n$, a field width $f$ of $O(\log n)$ clearly suffices. However, assumptions A and B are not realistic even for this value of $f$. In particular, vectors of $4w$ fields of $f$ bits each occupy $\Theta(f)$ $w$-bit words, and operations such as adding two vectors take $\Theta(f)$ time. We counter this problem by using a variable field width, storing small numbers in small fields and large numbers, of which there are few, in large fields. We begin by describing how to double and halve the field width.

In order to double the field width of a vector from $f$ to $2f$, we use the constant $1_{2f}$ to create a mask whose $f$-bit fields contain alternately only 0s and only 1s — again, the reader is referred to [AHNR98] for programming details. Using this mask, it is easy to separate the odd- and the even-numbered fields, storing each group of fields in a separate vector. This spreads the original vector over twice as many words and (implicitly) changes the field width from $f$ to $2f$. Halving the field width of a vector from $2f$ to $f$ can be done very easily by breaking the vector into halves and forming the disjunction of the halves after shifting one half by $f$ bits. Doubling the field width scrambles the order of the fields, but halving the field width returns the fields to their original order.

In order to carry out step 1, we sum the vectors of each nontrivial cluster $C$ according to a minimum-height binary tree, with leaves corresponding to the vectors of elements in $C$ and internal nodes corresponding to sums of leaves in subtrees. We use a constant field width at the leaves and larger field widths at inner nodes of the tree. The time needed at an inner node of field width $f$, including any necessary field doubling for the vectors produced at the children of the node, is $O(f)$. Since it is easy to see that a field width of $O(i)$ suffices for an inner node at height $i$, for all $i \geq 1$, while there are only $O(|C|/2^i)$ such nodes, the total time needed is $O(|C| \sum_{i=1}^{\infty} i/2^i) = O(|C|)$. Thus step 1 takes $O(n)$ time.

For each nontrivial cluster $C$, since a field width of $O(\log |C|)$ suffices, the computation of step 2 can be carried out in $O((\log |C|)^2)$ time. Over all nontrivial clusters, this sums to $O(n)$ time.

For step 3, we divide the nontrivial clusters into *size groups*: If a cluster contains between $2^j$ and $2^{j+1} - 1$ elements, for some integer $j \geq 1$, it is put in size group $j$. Separately for each value of $j$, we then sum the vectors computed in step 2 for all clusters in size group $j$ in a minimum-height binary tree. If $W_j$ is the total size of all clusters in size group $j$, the number of nodes at height $i$ in the tree is $O(W_j/2^{i+j})$, for all $i \geq 1$, and a field width of $O(i + j)$ suffices at each such node. As in the analysis of step 1, the summation within size group $j$ therefore takes $O(W_j)$ time, which sums to $O(n)$ over all values of $j$. What remains is to add $O(\log n)$ vectors, one for each size group. Since the maximum field width is $O(\log n)$, this can be done in $O((\log n)^2)$ time. Thus step 3 also takes $O(n)$ time.

The output of steps 1–3 is a vector of $4w$ fields, each of which specifies the badness associated with the corresponding bit position. The field width $f$ is $O(\log n)$, for which reason the set of fields containing the minimum badness can be computed in $O((\log n)^2)$ time by binary search over the range of possible minima. Specifically, we can assume the output of the binary search to be a vector, each field of which contains 1 if the corresponding badness is minimum,

and 0 otherwise. We now reduce the field width back to 1 by $\log f$ halvings, which also restores the original order of the fields. The result is a nonzero $4w$-bit vector, each 1 of which indicates a good bit position. To get hold of a single good position, we compute the position of the most significant bit set to 1. This can be done in constant time, employing weak nonuniformity [FW93, p. 431–432].

A final issue is the dependence on the constants $1_{2^i}$ for $i = 1, \ldots, \log f_{\max}$, where $f_{\max} = O(\log n)$. If $w < n$, we can compute $1_{2^i}$ in $O(\log n)$ time by $O(\log w)$ shift-and-or steps, each of which doubles the number of fields containing a 1. Otherwise we use native constants $l = \Theta(\sqrt{w})$, chosen as a power of 2, and $1_l$ together with the number $Q$, composed of segments of $l$ bits each, the $i$th of which is a "piece" of $1_{2^i}$. We can easily pick out the $i$th segment from $Q$. Multiplying the segment with $1_l$ yields the required constant $1_{2^i}$. □

**Theorem 6.3** *Let $S$ be a subset of $U$ of size $n$ and suppose that $\psi : U \to \{0,1\}^{4w}$ is an error-correcting code with minimum relative distance $\delta$ for some constant $\delta > 0$. Then there is a bit vector $D \in \{0,1\}^{4w}$ containing $O(\log n)$ 1s for which $\rho_D : x \mapsto (\psi(x)$ AND $D)$ is 1-1 on $S$, and such a bit vector can be computed deterministically from $\{\psi(x) \mid x \in S\}$ in $O(n \log n)$ time and $O(n)$ space.*

### 6.2.2 Unit-cost error-correcting codes

In order to evaluate the function $\rho_D$ of Theorem 6.3 in constant time, we need an error-correcting code that can be evaluated in constant time. Our construction is based on universal families of hash functions.

**Definition 6.1** *[WC81, MV84] For $c > 0$, a family $\mathcal{H}$ of functions from $U$ to $V$ is* $(c,2)$-universal *if, for all $x_1, x_2 \in U$ and all $y_1, y_2 \in V$, the probability that $h(x_1) = y_1$ and $h(x_2) = y_2$ is at most $c/|V|^2$ when $h \in \mathcal{H}$ is chosen uniformly at random.*

**Proposition 6.1** *Let $\mathcal{H}$ be a $(2,2)$-universal family of functions from $\{0,1\}^w$ to $\{0,1\}^{4w}$. For all $\delta$ with $0 < \delta \le \frac{1}{2}$, a random member of $\mathcal{H}$ is an error-correcting code of relative minimum distance $\delta$ with probability at least $1 - ((\frac{e}{\delta})^{4\delta}/4)^w$.*

*Proof.* Assume first that $\delta \ge \frac{1}{4w}$. The number of vectors in $\{0,1\}^{4w}$ within Hamming distance $k \ge 1$ of a fixed vector is

$$\begin{aligned}
\sum_{i=0}^{k} \binom{4w}{i} &\le \left(\frac{4w}{k}\right)^k \sum_{i=0}^{k} \binom{4w}{i} \left(\frac{k}{4w}\right)^i \\
&\le \left(\frac{4w}{k}\right)^k \left(1 + \frac{k}{4w}\right)^{4w} \qquad \text{(by the binomial theorem)} \\
&\le \left(\frac{4w}{k}\right)^k e^k = \left(\frac{4e\,w}{k}\right)^k .
\end{aligned}$$

This means that for all $x_1, x_2 \in \{0,1\}^w$ with $x_1 \neq x_2$ and for a random function $h \in \mathcal{H}$, the probability that the Hamming distance between $h(x_1)$ and $h(x_2)$ is no larger than $k$ is at most $2^{1-4w}\,(\frac{4ew}{k})^k$ (where we used $(2,2)$-universality). The probability that this happens for *any* of the $\binom{2^w}{2} < 2^{2w}/2$ such pairs is bounded by $2^{-2w}(\frac{4ew}{k})^k$. Setting $k = \lfloor 4\delta w \rfloor$, we see that $h$ fails to have minimum relative distance $\delta$ with probability at most $2^{-2w}(\frac{4ew}{\lfloor 4\delta w \rfloor})^{\lfloor 4\delta w \rfloor} \leq 2^{-2w}(\frac{4ew}{4\delta w})^{4\delta w} = ((\frac{e}{\delta})^{4\delta}/4)^w$.

If $\delta < \frac{1}{4w}$, the desired property of $h$ is simply that it should be injective. The probability that this is not the case is bounded by $2^{-2w} \leq 2^{-2w}(\frac{4ew}{4\delta w})^{4\delta w} = ((\frac{e}{\delta})^{4\delta}/4)^w$. □

The quantity $(\frac{e}{\delta})^{4\delta}/4$ converges to $1/4$ as $\delta$ approaches 0, so the success probability of Proposition 6.1 is positive for sufficiently small values of $\delta$ for all $w$. Thus we can indeed find an error-correcting code with relative minimum distance $\delta$ for some constant $\delta > 0$. As a concrete example, assume that $w > 10$ and let $\mathcal{H}$ be a $(2,2)$-universal family of functions from $\{0,1\}^w$ to $\{0,1\}^{4w}$. The proposition shows that more than half the functions in $\mathcal{H}$ are error-correcting codes with relative minimum distance $1/10$.

Many $(2,2)$-universal families are known. Moreover, when the range is $\{0,1\}^{O(w)}$, there are such families whose functions can be stored in constant space and evaluated in constant time. One example is $\{x \mapsto ((ax+b) \bmod p) \bmod 2^{4w} \mid a, b \in \{0,\ldots,p-1\}\}$, where $p$ is a fixed prime between $2^{4w}$ and $2^{4w+1}$. Multiplication of $O(w)$-bit numbers can be done using a constant number of single-word multiplications and additions. Also, as noted by Knuth [Knu73, p. 509], forming the remainder modulo a constant $p$ can be carried out in constant time with multiplications and shifts, so a division instruction is not needed to evaluate the functions in constant time. Another, very appealing, such family is $\{x \mapsto ((ax+b) \bmod 2^{5w}) \operatorname{div} 2^w \mid a, b \in \{0,\ldots,2^{5w}-1\}\}$, which has parameter 1 [Df96, Theorem 3(b)]. This family can be simplified to $\{x \mapsto ax \mid a \in \{0,\ldots,2^{5w}-1\}\}$ without decreasing the relative minimum distance of the corresponding error-correcting code by more than a constant factor. A direct proof of the error-correction property of this family, along with some results more general than those needed here, can be found in [Mil98].

### 6.2.3 Finishing the construction

We still need to address the issue of mapping injectively to $O(\log n)$ *consecutive* bits. We must "gather" the distinguishing bits in an interval of $O(\log n)$ positions. For every $D \subseteq \{1,\ldots,r\}$, define $Z_D = \{x \in \{0,1\}^r \mid x_i = 1 \Rightarrow i \in D\}$, the set of $r$-bit vectors that have only zeros outside the positions given by $D$.

**Lemma 6.3** *Let $D$ be a subset of $\{1,\ldots,4w\}$ of size $O(\log n)$. Then there is a function $\rho' : \{0,1\}^{4w} \to \{0,1\}^r$, where $r = O(\log n)$, that is 1-1 on $Z_D$ and can be evaluated in constant time, and a constant-size description of such a function can be computed deterministically in $o(n)$ time and space.*

*Proof.* Without loss of generality we can assume that $w = O(\sqrt[4]{n})$: If this is not the case, begin by using a method of Fredman and Willard [FW93, p. 428–429]

to gather the bits with positions in $D$ within $O((\log n)^4)$ consecutive positions by multiplying with a suitable integer $M_D$. The procedure for finding $M_D$ runs in time $(\log n)^{O(1)}$.

Partition $D$ into a constant number of sets $D_1, \ldots, D_k$ of size at most $\frac{1}{4}\log n$. Using the algorithm of Raman [Ram96], we can find hash functions $\rho_1, \ldots, \rho_k$ with range $\{0,1\}^{\lceil \frac{1}{2}\log n \rceil}$, perfect for $Z_{D_1}, \ldots, Z_{D_k}$, respectively, in time and space $O((\sqrt[4]{n})^2 w) = o(n)$. Given an argument value, masking out the bit positions not in $D_i$ and evaluating $\rho_i$, for $i = 1, \ldots, k$, and concatenating the resulting values can be done in constant time. This defines the function $\rho'$. □

Combining Theorem 6.3 and Lemma 6.3, we have the desired result on universe reduction:

**Lemma 6.4** *Let $S$ be a subset of $U$ of size $n$. Then there is a function from $U$ to $\{0,1\}^r$, with $r = O(\log n)$, that is 1-1 on $S$ and can be evaluated in constant time, and a constant-size description of such a function can be computed deterministically in time $O(n \log n)$ and space $O(n)$.*

## 6.3 Universes of polynomial size

In this section we develop a variant of the double-displacement scheme of Tarjan and Yao [TY79], which computes an efficient perfect hash function for $w = O(\log n)$. The construction algorithm of Tarjan and Yao is deterministic and has worst-case complexity $\Theta(n^2)$. We first show how to achieve expected construction time $O(n)$ with a randomized algorithm. This algorithm is then derandomized using the method of conditional expectations, which yields a deterministic $O(n \log n)$-time algorithm.

### 6.3.1 Reduction to universes of quadratic size

Following Tarjan and Yao, we observe that bit vectors of length $O(\log n)$ can be regarded as constant-length strings over an alphabet of size $n$. The trie (with $n$-way branching) of such strings permits lookup of elements (and associated values) in constant time. Although each of the $O(n)$ nodes of the trie uses a table of size $n$, only $O(n)$ entries of all tables contain important information (an element or a pointer). That is, to store all tables in space $O(n)$, it suffices to construct an efficient perfect hash function for the set of important entries within the universe of all $O(n^2)$ table entries. For this reason Tarjan and Yao proceed to study the case $w \leq 2\log n + O(1)$.

### 6.3.2 Randomized double displacement

Our aim is to find a function $h : U \to \{0,1\}^r$, with $r = \log n + O(1)$, such that there are no *collisions* under $h$, i.e., pairs $\{x, y\}$ of distinct keys in $S$ with $h(x) = h(y)$. The *displacement* method of Tarjan and Yao can be viewed as a way of taking an "imperfect" function $f : U \to \{0,1\}^r$ and generating a new function with fewer collisions between the keys. The method needs "advice" in the form of a function $g : U \to \{0,1\}^r$ such that $x \mapsto (f(x), g(x))$ is 1-1 on $S$.

The idea of Tarjan and Yao is to use $f(x)$ as an index into a table of suitably chosen *displacement values* $a_v \in \{0,1\}^r$, with $v \in \{0,1\}^r$. Then the function $x \mapsto g(x) \oplus a_{f(x)}$, where $\oplus$ denotes bitwise exclusive-or, may have far fewer collisions than $f$. Also, $x \mapsto (g(x) \oplus a_{f(x)}, f(x))$ is 1-1 on $S$, so the procedure can be repeated. Two repetitions will suffice; hence the term "double displacement". Our contribution is an efficient procedure for finding suitable displacement values.

**Definition 6.2** *For $q \geq 0$, a pair of functions $(f,g)$, both mapping from $U$ to $\{0,1\}^r$, is* $q$-good *if $f$ has at most $q$ collisions and $x \mapsto (f(x),g(x))$ is 1-1 on $S$.*

**Lemma 6.5** *Suppose that $(f,g)$ is $q$-good and that $r \geq \log n + 1$. Then there exist $a_v \in \{0,1\}^r$, for $v \in \{0,1\}^r$, such that $(x \mapsto g(x) \oplus a_{f(x)}, f)$ is $q'$-good, where $q' = \min\{n, \lfloor 2^{3-r}\, q \rfloor\, n\}$. On input $\{(f(x),g(x)) \mid x \in S\}$, such values $a_v$ can be computed by a randomized algorithm in expected time $O(n)$ and space $O(n)$.*

Let us first see how to use the lemma to obtain an efficient perfect hash function for $S$. We will need $r > \max\{w/2, \log n + 3\}$. Since $r \geq w/2$, it is trivial to find a pair $(f,g)$ of constant-time-evaluable functions that is $\binom{n}{2}$-good (e.g., let $f(x)$ and $g(x)$ be the first and the last $r$ bits of $x$, respectively). By Lemma 6.5, we can find $a_v \in \{0,1\}^r$, for $v \in \{0,1\}^r$, such that $(x \mapsto g(x) \oplus a_{f(x)},\ f)$ is $n$-good. Applying Lemma 6.5 to the pair $(x \mapsto g(x) \oplus a_{f(x)},\ f)$, we obtain values $b_v \in \{0,1\}^r$, for $v \in \{0,1\}^r$, such that $(x \mapsto f(x) \oplus b_{g(x)\oplus a_{f(x)}},\ x \mapsto g(x) \oplus a_{f(x)})$ is $(\lfloor 2^{3-r} n \rfloor\, n)$-good. By the choice of $r$, $\lfloor 2^{3-r} n \rfloor = 0$, so the function $x \mapsto f(x) \oplus b_{g(x)\oplus a_{f(x)}}$ is 1-1 on $S$. When $w \leq 2\log n + O(1)$ we can choose $r = \log n + O(1)$, so the hash-function parameters use space $O(n)$, and the range $\{0,1\}^r$ has size $O(n)$. Thus $x \mapsto f(x) \oplus b_{g(x)\oplus a_{f(x)}}$ is the desired efficient perfect hash function.

Summing up Sections 6.3.1 and 6.3.2, we have, for $w = O(\log n)$, a randomized algorithm constructing an efficient perfect hash function in expected time $O(n)$ and space $O(n)$.

*Proof of lemma 6.5.* For $v \in \{0,1\}^r$, let $S_v = \{x \in S \mid f(x) = v\}$. Our algorithm starts by bucket-sorting the $(f(x),g(x))$-pairs, for $x \in S$, by their first coordinates. It is then easy to compute a permutation $v_1, \ldots, v_{2^r}$ of $\{0,1\}^r$ with $|S_{v_1}| \geq |S_{v_2}| \geq \cdots \geq |S_{v_{2^r}}|$. We now successively compute $a_{v_1}, \ldots, a_{v_{2^r}}$; i.e., the sets $S_v$ are processed in some order of nonincreasing size.

Before the $j$th step of the computation, for $1 \leq j \leq 2^r$, the algorithm will have determined $a_{v_1}, \ldots, a_{v_{j-1}}$, and thus also the value of $g(x) \oplus a_{f(x)}$ for all $x \in S_{v_{<j}}$, where $S_{v_{<j}} = \bigcup_{i=1}^{j-1} S_{v_i}$. We maintain counts of the values determined so far, $m_v = |\{x \in S_{v_{<j}} \mid g(x) \oplus a_{f(x)} = v\}|$, for $v \in \{0,1\}^r$. If $a_{v_j}$ is picked at random from $\{0,1\}^r$, the expected number of new collisions, i.e., pairs $(x,y) \in S_{v_{<j}} \times S_{v_j}$ with $g(x) \oplus a_{f(x)} = g(y) \oplus a_{f(y)} = g(y) \oplus a_{v_j}$, is $|S_{v_j}|\,|S_{v_{<j}}|/2^r$. The algorithm aims to introduce at most twice this number of collisions, i.e., to find $a_{v_j}$ such that $\sum_{y \in S_{v_j}} m_{g(y)\oplus a_{v_j}} \leq \lfloor 2\,|S_{v_j}|\,|S_{v_{<j}}|/2^r \rfloor$ (we can round down since the left-hand side is an integer). By Markov's inequality,

the expected number of random attempts required to find such an $a_{v_j}$ is no more than 2. Each attempt takes time $O(|S_{v_j}|)$, so the expected running time for all steps is $O(n)$.

It remains to be seen that the number of collisions of $x \mapsto g(x) \oplus a_{f(x)}$ is no larger than $q'$. Note that the number of collisions of $f$ is $\sum_{v\in\{0,1\}^r} \binom{|S_v|}{2}$ and, by assumption, is at most $q$. Let $j^* = |\{v \in \{0,1\}^r \mid |S_v| > 1\}|$. The number of collisions of $x \mapsto g(x) \oplus a_{f(x)}$ is at most

$$
\begin{aligned}
&\sum_{j=1}^{2^r} \lfloor 2\,|S_{v_j}|\,|S_{v_{<j}}|/2^r \rfloor \\
&\quad \le \sum_{j=1}^{j^*} \lfloor 2^{1-r}\,|S_{v_j}| \sum_{i=1}^{j-1} |S_{v_i}| \rfloor && \text{(as } 2n/2^r < 1\text{)} \\
&\quad \le \sum_{j=1}^{j^*} \lfloor 2^{1-r} \min\{|S_{v_j}|\,n,\ \sum_{i=1}^{j-1} |S_{v_i}|^2\} \rfloor && \text{(as } |S_{v_j}| \le |S_{v_i}| \text{ for } i < j\text{)} \\
&\quad \le \sum_{j=1}^{j^*} \min\{2^{1-r}|S_{v_j}|\,n,\ \lfloor 2^{3-r} \sum_{i=1}^{j-1} \tbinom{|S_{v_i}|}{2} \rfloor\} && \text{(as } |S_{v_i}| \ge 2 \text{ for } i \le j^*\text{)} \\
&\quad \le \min\{n,\ \lfloor 2^{3-r}\,q \rfloor\,n\}\ .
\end{aligned}
$$

### 6.3.3 Derandomizing double displacement

In this section we employ the method of conditional expectations to obtain a deterministic $O(n \log n)$-time version of the algorithm of Section 6.3.2. Recall the problem solved in the randomized part of the algorithm: Given a table of values $m_v$, for $v \in \{0,1\}^r$, and a set $X \subseteq \{0,1\}^r$, find $a \in \{0,1\}^r$ such that $\sum_{x\in X} m_{x\oplus a} \le \lfloor 2^{1-r}\,|X| \sum_{v\in\{0,1\}^r} m_v \rfloor$.

We show how to find $a$ deterministically in $O(|X|\,r) = O(|X| \log n)$ time. That is, the time for finding a displacement value is $O(\log n)$ times that expected for the randomized algorithm. To do this we maintain an extension of the table, storing values $m_u$ for all bit strings $u$ of length at most $r$. For $k = 0, \ldots, r$, let $\pi_k(v)$ denote the $k$-bit prefix of $v \in \{0,1\}^r$, and for $u \in \{0,1\}^k$ define $Z_u = \{v \in \{0,1\}^r \mid \pi_k(v) = u\}$ as the set of bit strings of length $r$ with $u$ as a prefix. Then the extended table is defined by $m_u = \sum_{v\in Z_u} m_v$.

We can think of the extended table as a binary trie whose leaves (indexed by strings of length $r$) contain the original table entries and each of whose internal nodes contains the sum over all leaves in its sub-trie. The extension can be initialized and maintained during $n$ updates of leaves in time $O(nr) = O(n \log n)$.

Starting with $u_0$, the empty string, we show how to find a sequence of bit strings $u_0, \ldots, u_r$, where $u_k \in \{0,1\}^k$, such that the expected value of $\sum_{x\in X} m_{x\oplus a}$, when $a \in Z_{u_k}$ is chosen uniformly at random, is at most

$$2^{-r}|X| \sum_{v\in\{0,1\}^r} m_v$$

for $k = 0, \ldots, r$. Since $Z_{u_r} = \{u_r\}$, we must have

$$\sum_{x \in X} m_{x \oplus u_r} \leq \lfloor 2^{-r} |X| \sum_{v \in \{0,1\}^r} m_v \rfloor$$

(rounding down being justified by the integrality of the left-hand side), and we can take $a = u_r$.

For $u_0$ the requirement is clearly met, so for $1 \leq k < r$ the task is to extend $u_{k-1}$ to $u_k$ without increasing the expected value. By linearity of expectation, we can always achieve this by extending $u_{k-1}$ by either 0 or 1. An appropriate extension can be found by computing the expectations in time $O(|X|)$:

**Lemma 6.6** *For every $u \in \{0,1\}^k$, where $0 \leq k \leq r$, the expectation of $\sum_{x \in X} m_{x \oplus a}$, when $a \in Z_u$ is chosen uniformly at random, is $2^{k-r} \sum_{x \in X} m_{\pi_k(x) \oplus u}$.*

*Proof.* For every $x \in X$ we have $\sum_{a \in Z_u} m_{x \oplus a} = m_{\pi_k(x) \oplus u}$ by definition, so the expected value of $m_{x \oplus a}$, when $a \in Z_u$ is chosen uniformly at random, is $2^{k-r} m_{\pi_k(x) \oplus u}$. The lemma follows by linearity of expectation. □

# Chapter 7

## A trade-off for worst-case efficient dictionaries

The *dictionary* is among the most fundamental data structures. It supports maintenance of a set $S$ under insertion and deletion of elements, called *keys*, from a universe $U$. Data is accessed through membership queries, "$x \in S$?". In case of a positive answer, the dictionary also returns a piece of *satellite data* that was associated with $x$ when it was inserted.

Dictionaries have innumerable applications in algorithms and data structures. They are also interesting from a foundational point of view, as they formalize the basic concept of information retrieval from a "corpus of knowledge" (an associative memory). The main question of interest is: What are the computational resources, primarily time and space, needed by a dictionary? Our interest lies in how good *worst-case* bounds can be achieved. For the sake of simplicity one usually assumes that keys are bit strings in $U = \{0,1\}^w$, for a positive integer parameter $w$, and restricts attention to the case where keys of $U$ fit a single memory location. Memory locations are referred to as *words*, and $w$ is called the *word length*. Each piece of satellite data is assumed to be a single word, which could be a pointer to more bulky data. This means that the best space complexity one can hope for is $O(n)$ words, where $n$ is the size of $S$. We will only consider data structures using $O(n)$ space. Motivated by applications in which queries are somewhat more frequent than updates, our focus is on providing very fast membership queries, while maintaining fast updates.

In comparison-based models the complexity of dictionary operations is well understood. In particular, queries require $\Omega(\log n)$ comparisons in the worst case, and we can implement all dictionary operations in $O(\log n)$ time on a pointer machine using only comparisons [AVL62]. However, for a model of computation more resembling real-world computers, a unit cost RAM with words of size $w$, the last decade has seen the development of algorithms "blasting through" comparison-based lower bounds [FW93, And96, BF99a, AT00], resulting in a dictionary with time $O(\sqrt{\log n / \log\log n})$ for all operations. Other authors have found ways of combining very fast queries with nontrivial update performance. Most notably, [Mil98] achieves constant query time together with update time $O(n^\epsilon)$ for any constant $\epsilon > 0$. It is important to note that all bounds in this chapter are *independent of* $w$, unless explicitly stated otherwise.

All previous schemes have had either query time $(\log n)^{\Omega(1)}$ or update time $2^{\omega(\sqrt{\log n})}$. In this chapter we obtain the following trade-off between query time and update time:

**Theorem 7.1** *There is a deterministic dictionary running on a unit cost RAM with word length $w$ that, when storing $n$ keys of $w$ bits, uses $O(n)$ words of storage, supports insertions and deletions in $O((\log n \log\log n)^2)$ time, and answers membership queries in $O((\log\log n)^2/\log\log\log n)$ time.*

Ignoring polynomial differences, our dictionary has exponentially faster queries or exponentially faster updates than any previous scheme.

## Model of computation

The *word RAM* model used is the same as in the earlier work mentioned above. We only briefly describe the model, and refer to the "multiplication model" in [Hag98b] for details.

The RAM operates on words of size $w$, alternately looked upon as bit strings and integers in $\{0,\ldots,2^w-1\}$. It has the following computational operations: bitwise boolean operations, shifts, addition and multiplication. Note that all operations can also be carried out in constant time on arguments spanning a constant number of words.

Word RAM algorithms are allowed to be *weakly nonuniform*, i.e., use a constant number of word-size constants depending only on $w$. These constants can be thought of as computed at "compile time". In this chapter, weak nonuniformity could be replaced by an extended instruction set, with certain natural but nonstandard instructions from uniform $\mathrm{NC}^1$.

## Related work

As mentioned, the best known worst-case bound holding simultaneously for all dictionary operations is $O(\sqrt{\log n/\log\log n})$. It is achieved by a dynamization of the static data structure of [BF99a] using the exponential search trees of [AT00]. This data structure, from now on referred to as the BFAT data structure, in fact supports *predecessor queries* of the form "What is the largest key of $S$ not greater than $x$?". Its time bound improves significantly if the word length is not too large compared to $\log n$. In particular, if $w=(\log n)^{O(1)}$ the bound is $O((\log\log n)^2/\log\log\log n)$. These properties of the BFAT data structure will play a key role in our construction.

An unpublished manuscript by [Sun93] states an amortized lower bound of time $\Omega(\frac{\log\log_w n}{\log\log\log_w n})$ per operation in the cell probe model of [Yao81], for $w=\Omega(\log n\log\log n)$. In particular, this implies the same lower bound on the word RAM. Note that for $w=(\log n)^{O(1)}$, the BFAT data structure has time per operation polynomially related to the lower bound.

A seminal result of [FKS84] is that in the *static* case (with no updates to the data structure), one can achieve *constant* query time. Chapter 6 describes a dictionary with constant query time, that can be constructed in time $O(n\log n)$.

A standard dynamization then gives, for any "nice" function $q(n) = O(\sqrt{\log n})$, query time $O(q(n))$ and update time $O(n^{1/q(n)})$.

An overview of known trade-offs for linear space dictionaries is given in Fig. 7.1.

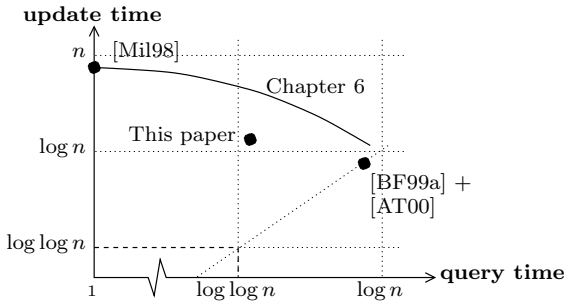


Figure 7.1: Overview of trade-offs for linear space dictionaries.

If one abandons the requirement of good worst-case performance, new possibilities arise. Most notably, one can consider amortized and randomized (expected) time bounds. Using the universal hash functions of [CW79] for chained hashing, one obtains expected constant time for all operations. The query time can be made worst-case constant by dynamizing the static dictionary of [FKS84] to allow updates in amortized expected constant time [DfKM$^+$94]. Subsequent works have described dictionaries in which every operation is done in constant time with high probability [DfMadH90, DfGMP92, Wil00]. The best result with no dependence on $w$ is, for any constant $c$, a success probability of $1 - n^{-c}$ [DfGMP92, DfMadH90].

For a general introduction to dynamic data structures, covering both amortized and worst-case bounds, we refer to the book of Overmars [Ove83].

## Overview

As mentioned, the BFAT data structure is very fast when the word length is $(\log n)^{O(1)}$. Our strategy is to reduce the dictionary problem to a predecessor problem on words of length $(\log n)^{O(1)}$, solved by the BFAT data structure. A query for $x$ translates into a query for $h(x)$ in the predecessor data structure, where $h$ is an efficiently evaluable *universe reduction function*, mapping $U$ to a (smaller) universe $\{0,1\}^r$. This approach is similar to universe reduction schemes previously employed in the static setting (see Chapter 6). One difference is that we use $r = \Theta(\log^2 n)$ rather than $r = \Theta(\log n)$. Moreover, in the dynamic setting we need to dynamically update $h$ when new keys are inserted, which means that $h(x)$ may change for keys $x \in S$. This can be handled, however, as a predecessor query for $h(x)$ may still find the BFAT key for $x$ if it is smaller than $h(x)$. In general there may be other keys between the BFAT key of $x$ and $h(x)$. However, we maintain the invariant that if $x \in S$ then $x$ can be found in a sorted list of $O(\log n)$ keys associated with the predecessor of $h(x)$.

After some preliminary observations and definitions, Section 7.2 introduces

the tools needed for the universe reduction. In Section 7.3 we show how to construct a dictionary with the desired properties, except that the time bound for updates is amortized. Finally, Section 7.4 describes how to extend the technique of the amortized data structure to make the bound worst-case.

## 7.1 Preliminaries

### 7.1.1 Simplifications

Without loss of generality, we may disregard deletions and consider only insertions and membership queries. Deletions can be accommodated within the same time bound as insertions, by the standard technique of *marking* deleted keys and periodically *rebuilding* the dictionary. To be able to mark a key, we let the satellite information point to a marker bit and the "real" satellite information. For the rebuilding one maintains two insertion-only dictionaries: An active one catering insertions and membership queries, and an inactive one to which two new unmarked keys from the active dictionary are transferred each time a key is marked (in both dictionaries). When all unmarked keys have been transferred, the inactive dictionary is made active, and we start over with an empty inactive dictionary. This assures that no more than a constant fraction of the keys are marked at any time. So if the insertion-only dictionary uses linear space, the space usage of the above scheme is $O(n)$.

We may also assume that $w \geq \log^5 n$. Word size $O(\log^5 n)$ can be handled using the BFAT data structure directly, and standard rebuilding techniques can be used to change from one data structure to the other. Similarly, we assume that $n$ is larger than some fixed, sufficiently large constant, since constant size dictionaries are trivial to handle.

### 7.1.2 Notation and definitions

Throughout this chapter $S$ refers to a set of $n$ keys from $U$. When we need to distinguish between values before and after a change of the data structure, we used primed variables to denote the new values. We will look at bit strings also as binary numbers with the most significant bits first. The $i$th last bit of a string $x$ is denoted by $x_i$; in particular, $x = x_l x_{l-1} \ldots x_1$, where $l$ is the length of $x$. We say that $x$ is the *incidence string* of $D \subseteq \{1, \ldots, l\}$ if $i \in D \Leftrightarrow x_i = 1$, for $1 \leq i \leq l$. The *Hamming weight* of $x$ is the number of positions $i$ where $x_i = 1$. The *Hamming distance* between strings $x$ and $y$ is the number of positions $i$ where $x_i \neq y_i$. For clarity, we will distinguish between keys of our dictionary and keys in predecessor data structures, by referring to the latter as *p-keys*. The set of positive integers is denoted by $\mathbb{N}$.

## 7.2 Universe reduction tools

Miltersen [Mil98] has shown the utility of error-correcting codes to deterministic universe reduction. This approach plays a key role in our construction, so we review it here. For further details we refer to Chapter 6, which sums up all

the results and techniques needed here. The basic idea is to employ an error-correcting code $\psi : \{0,1\}^w \to \{0,1\}^{4w}$ (which is fixed and independent of $S$) and look at the transformed set $\{\psi(x) \mid x \in S\}$. For this set it is possible to find a very simple function that is 1-1 and has small range, namely a projection onto $O(\log n)$ bit positions.

The code must have *relative minimum distance* bounded from 0 by a fixed positive constant, that is, there must exist a constant $\delta > 0$ such that any two distinct codewords $\psi(x)$ and $\psi(y)$ have Hamming distance at least $4w\delta$. The infimum of such constants is called the relative minimum distance of the code. We can look at the transformed set without loss of generality, since Miltersen showed that such an error-correcting code can be computed in constant time using multiplication: $\psi(x) = c_w \cdot x$, for suitable $c_w \in \{0,1\}^{3w}$. The choice of $c_w$ is a source of weak nonuniformity. The relative minimum distance of this code is greater than $\delta = 1/25$.

**Definition 7.1** *For an equivalence relation $\equiv$ over $T \subseteq U$, a position $d \in \{1,\ldots,4w\}$ is* discriminating *if, for $K = \{\{x,y\} \subseteq T \mid x \neq y \wedge x \equiv y\}$, $|\{\{x,y\} \in K \mid \psi(x)_d = \psi(y)_d\}| \leq (1-\delta)\,|K|$.*

*For $T \subseteq U$, a set $D \subseteq \{1,\ldots,4w\}$ is* distinguishing *if, for all pairs of distinct keys $x, y \in T$, there exists $d \in D$ where $\psi(x)_d \neq \psi(y)_d$.*

Miltersen's universe reduction function (for a set $T$) is $x \mapsto \psi(x)$ AND $v$, where AND denotes bitwise conjunction and $v$ is the incidence string of a distinguishing set for $T$. A small distinguishing set can be found efficiently by finding discriminating bits for certain equivalence relations:

**Lemma 7.1** *(Miltersen) Let $T \subseteq U$ be a set of $m$ keys, and suppose there is an algorithm that, given the equivalence classes of an equivalence relation over $T$, computes a discriminating position in time $O(m)$. Then a distinguishing set for $T$ of size less than $\frac{2}{\delta}\log m$ can be constructed in time $O(m \log m)$.*

*Proof sketch.* Elements of the distinguishing set may be found one by one, as discriminating positions of the equivalence relation where $x, y \in T$ are equal if and only if $\psi(x)$ and $\psi(y)$ do not differ on the positions already chosen. The number of pairs not distinguished by the first $k$ discriminating positions is at most $(1-\delta)^k \binom{m}{2}$. □

Hagerup [Hag99] showed how to find a discriminating position in time $O(m)$. We will need a slight extension of his result.

**Lemma 7.2** *(Hagerup) Given a set $T \subseteq U$ of $m$ keys, divided into equivalence classes of a relation $\equiv$, a discriminating position $d$ can be computed in time $O(m)$. Further, for an auxiliary input string $b \in \{0,1\}^{4w}$ of Hamming weight $o(w)$ and $w$ larger than some constant, we can assure that $b_d = 0$.*

*Proof sketch.* In [Hag99] it is shown how to employ word-level parallelism to compute $|\{\{x,y\} \subseteq T \mid x \neq y,\ x \equiv y,\ \psi(x)_d = \psi(y)_d\}|$ for all $d \in \{1,\ldots,4w\}$ in time $O(m)$. The algorithm computes a vector of $O(\log m)$-bit numbers compressed into $O(\log m)$ words.

Word-parallel binary search can be used to find the index of the smallest entry, which will be discriminating. To avoid positions where $b_d = 1$, we replace the corresponding entries of the vector with the largest possible integer, before finding the minimum. This corresponds to changing the error-correcting code to be constant (i.e. non-discriminating) on the bit positions indicated by $b$. Since $b$ has Hamming weight $o(w)$, the relative minimum distance of this modified code is still greater than $\alpha$, for $n$ large enough. Hence, this procedure will find a discriminating bit position. □

**Definition 7.2** *A set of positions* $\{d_1, \ldots, d_p\} \subseteq \{1, \ldots, 4w\}$ *is* well separated *if* $|d_i - d_j| \geq 2\,p$ *for all* $i, j$ *where* $1 \leq i < j \leq p$.

We will keep the distinguishing positions used for the universe reduction well separated. This is done by forming a vector $b$ with 1s in positions within some distance of previously chosen positions, and using this in the algorithm of Lemma 7.2. Good separation allows us to "collect" distinguishing bits into consecutive positions (in arbitrary order):

**Lemma 7.3** *For a list* $d_1, \ldots, d_p$ *of well separated positions, there is a function* $f_{\bar{d}} : \{0,1\}^{4w} \to \{0,1\}^p$ *that can be stored in a constant number of words and evaluated in constant time, such that for any* $x \in \{0,1\}^{4w}$ *and* $i \in \{1, \ldots, p\}$, *we have* $f_{\bar{d}}(x)_i = x_{d_i}$. *The function description can be updated in constant time when the list is extended (with position* $d_{p+1}$*) and under changes of positions, given that the resulting list is well separated.*

*Proof.* We will show how to "move" bit $d_i$ of $x \in \{0,1\}^{4w}$ to bit $4w + i$ of a $(4w + p)$-bit string. The desired value can then be obtained by shifting the string by $4w$ bits. We first set all bits outside $\{d_1, \ldots, d_p\}$ to 0 using a bit mask. Then simply multiply $x$ by $M_{\bar{d}} = \sum_{i=1}^{p} 2^{4w+i-d_i}$ (a method adopted from [FW93, p. 428-429]). One can think of the multiplication as $p$ shifted versions of $x$ being added. Note that if there are no carries in this addition, we do indeed get the right bits moved to positions $4w + 1, \ldots, 4w + p$. However, since the positions are well separated, all carries occur either left of the $4w+p$th position (and this has no impact on the values at positions $4w + 1, \ldots, 4w + p$) or right of position $4w - p$ (and this can never influence the values at positions greater than $4w$, since there are more than enough zeros in between to swallow all carries). Note that $M_{\bar{d}}$ can be updated in constant time when a position is added or changed. □

## 7.3 Dictionary with amortized bounds

This section presents the main ideas allowing the universe reduction techniques of Section 7.2 and a predecessor data structure to be combined in an efficient dynamic (insertion-only) dictionary with *amortized* bounds. Section 7.4 will extend the ideas to achieve nearly the same bounds in the worst case.

We will start by outlining how the query algorithm is going to work. A membership query for $x$ proceeds by first computing the value $h(x)$ of the universe reduction function $h$, described in Section 7.3.1. Then we search for the p-key $q$ that is predecessor of $h(x)$ (if it does not exist, the search is unsuccessful). Finally, we search for $x$ in a list $A_q$ associated with $q$. The invariant is kept that if $x \in S$, then $x$ is found in this way.

### 7.3.1 The universe reduction function

We will not maintain a distinguishing set for $S$ itself, but rather maintain $k = \lceil \log(n+1) \rceil$ distinguishing sets $D_1, \ldots, D_k$ for a partition of $S$ into subsets $S_1, \ldots, S_k$. Let $h_{D_i} : U \to \{0,1\}^{|D_i|}$ denote a function "collecting" the bits in positions of $D_i$ from its error-corrected input, i.e., such that for each $d \in D_i$ there is $j \in \{1, \ldots, |D_i|\}$ where $h_{D_i}(x)_j = \psi(x)_d$. The universe reduction function is:

$$h : x \mapsto h_{D_k}(x) \circ 1 \circ h_{D_{k-1}}(x) \circ 1 \circ \cdots \circ h_{D_1}(x) \circ 1 \tag{7.1}$$

where $\circ$ denotes concatenation. The basic idea is that set $S_i$ and distinguishing set $D_i$ changes only once every $2^i$ insertions. This means that during most insertions, function values change only in the least significant bits. Hence, for many keys $x \in S$, a query for the predecessor of $h(x)$ will return the same p-key before and after a change of $h$. In cases where a new p-key becomes predecessor of $h(x)$, we explicitly put $x$ in the list of keys associated with the p-key. Details follow below.

The following invariants are kept:

(1) $|S_i| \in \{0, 2^i\}$, for $1 \leq i \leq k$.

(2) $|D_i| = 2i/\delta$, for $1 \leq i \leq k$.

(3) $|d_1 - d_2| \geq 4k^2/\delta$, for all distinct $d_1, d_2 \in D_1 \cup \cdots \cup D_k$.

The first invariant implies that $|S|$ must be even, but this is no loss of generality since insertions may be processed two at a time. Invariant (7.3.1) is feasible because of the lower bound on $w$. By invariant (7.3.1) the size of $D_1 \cup \cdots \cup D_k$ is at most $2k^2/\delta$, so invariant (7.3.1) implies that $D_1 \cup \cdots \cup D_k$ is well separated. Lemma 7.3 then gives that $h$ can be evaluated in constant time and updated in time $O(\sum_{i=1}^{k} 2i/\delta) = O(\log^2 n)$ when $D_1, \ldots, D_k$ changes. Within this time bound we can also compute a string $b \in \{0,1\}^{4w}$ of Hamming weight $O(\log^4 n) = o(w)$, such that all bits of $b$ with distance less than $4\,(k+1)^2/\delta$ to a position in $D_1, \ldots, D_k$ are 1s: Multiply the incidence string of $D_1 \cup \cdots \cup D_k$ by $2^{4\,(k+1)^2/\delta} - 1$ (whose binary expansion is the string $1^{4\,(k+1)^2/\delta}$) to get a string $v$, and compute the bitwise or of $v$ and $v$ right-shifted $4\,(k+1)^2/\delta$ positions.

We now describe how to update $S_1, \ldots, S_k$ and $D_1, \ldots, D_k$ when new keys are inserted. Let $m$ denote the smallest index such that $S_m = \emptyset$, or let $m = k+1$ if no subset is empty. When new keys $x$ and $y$ are inserted, we compute a distinguishing set $D$ for $S_{m-1} \cup \cdots \cup S_1 \cup \{x, y\}$. By Section 7.2 this can

be done such that $|D| = 2m/\delta$, adding extra positions if necessary. Also, using the string $b$ we can assure that positions in $D$ have distance at least $4\,(k+1)^2/\delta$ from positions in $D_1, \ldots, D_k$ as well as from each other. We then perform the following updates: $S'_m = S_{m-1} \cup \cdots \cup S_1 \cup \{x, y\}$, $D'_m = D$, and $S'_{m-1} = \cdots = S'_1 = \emptyset$. The invariants are easily seen to remain satisfied.

To see that the amortized time per insertion is $O(\log^2 n)$, note that no key at any time has been part of more than $k$ distinguishing set computations, and that each such computation took time $O(\log n)$ per key.

### 7.3.2 Using the predecessor data structure

The predecessor data structure has one p-key for each key in $S$. For $x \in S_i$ the key is

$$q_x = h_{D_k}(x) \circ 1 \circ h_{D_{k-1}}(x) \circ 1 \circ \cdots \circ h_{D_i}(x) \circ 1 \circ 0^{z_i} \tag{7.2}$$

where $z_i$ is the number of bits in $h_{D_{i-1}}(x) \circ 1 \circ \cdots \circ h_{D_1}(x) \circ 1$. Note that $q_x \leq h(x) < \mu_x$, where $\mu_x = q_x + 2^{z_i}$. We also observe that, since $h_{D_i}$ is 1-1 on $S_i$, p-keys belong to unique keys of $S$, and that $q_x$ and $\mu_x$ are fixed during the time where $x \in S_i$. Associated with $q_x$ is a sorted list of the following keys, each represented by a pointer to a unique instance:

$$A_{q_x} = \{y \in S \mid q_y \leq q_x < \mu_y\} \ . \tag{7.3}$$

For $y \in S_j$, the condition $q_y \leq q_x < \mu_y$ holds if and only if $q_x$ has prefix $h_{D_k}(y) \circ 1 \circ \cdots \circ h_{D_j}(y) \circ 1$. Since $h_{D_j}$ is 1-1 on $S_j$, it follows that for $x \in S_i$, the set $A_{q_x}$ consists of at most one key from each of $S_k, \ldots, S_i$. Also, $A_{q_x}$ is constant during the time where $x \in S_i$. To see that our query algorithm is sound, note that for $x \in S$ the search for a predecessor of $h(x)$ returns a p-key $q$ with $q_x \leq q \leq h(x) < \mu_x$.

When $S_1, \ldots, S_k$ and $D_1, \ldots, D_k$ have changed, we must update the predecessor data structure accordingly. We first delete the old p-keys of keys in $S'_m = S_1 \cup \cdots \cup S_{m-1}$. For each key $x \in S'_m$ we then compute the new p-key $q'_x$, and its predecessor is searched for. If $q'_x$ has no predecessor, then its associated list must contain only $x$. If there is a predecessor $q'_y$, the associated list consists of $x$ plus a subset of $A_{q'_y}$. To see this, recall that no key from $S'_m \backslash \{x\}$ can be in $A_{q'_x}$, and that by invariant $A_{q'_y} = \{v \in S' \mid q'_v \leq q'_y < \mu'_v\}$. Specifically, we have $A_{q'_x} = \{v \in A_{q'_y} \mid q'_x < \mu'_v\} \cup \{x\}$. Thus, in time $O(\log n)$ per key we can create and insert p-keys and associated lists for all $x \in S'_m$. Again, this means that the amortized cost of updating the predecessor data structure is $O(\log^2 n)$.

Apart from the predecessor query, the time used by the query algorithm is $O(\log k) = O(\log \log n)$. Hence, if we use the BFAT predecessor data structure, the time to perform a query is $O((\log \log n)^2 / \log \log \log n)$. The only part of our data structure not immediately seen to be in linear space is the set of associated lists. For $x \in S_i$, the set $A_{q_x}$ contains at most $k+1-i$ keys, so the total length of all associated lists is $O(\sum_{i=1}^{k}(k+1-i)\,2^i) = O(n\,(1+\sum_{i=1}^{k}(k-i)\,2^{-(k-i)})) = O(n)$.

**Proposition 7.1** *There is a deterministic dictionary that, when storing a set of $n$ keys, uses $O(n)$ words of storage, answers membership queries in time*

$O((\log\log n)^2/\log\log\log n)$, *and supports insertions and deletions in amortized time* $O(\log^2 n)$.

## 7.4 Dictionary with worst-case bounds

Whereas the dictionary described in the previous section efficiently performs any sequence of operations, the worst-case time for a single insertion is $\Omega(n \log n)$ (this happens when we compute a distinguishing set for $S_k$, which has size $\Omega(n)$). We now describe a similar dictionary that has the worst-case time bounds stated in Theorem 7.1. Focus is on the aspects different from the amortized case. In particular, the description must be read together with Section 7.3 to get the full picture.

The worst-case dynamization technique used is essentially that of [OvL81a]. By maintaining the partition of $S$ slightly differently than in Section 7.3, it becomes possible to start computation of distinguishing sets for future partitions early, such that a little processing before each insertion suffices to have the distinguishing sets ready when the partition changes. Similarly, predecessor data structure updates can be done "in advance", leaving little work to be done regardless of whether an insertion triggered a small or a large change in the partition.

The partition of $S$ now involves $2k$ sets, $S_1, \ldots, S_k$ and $T_1, \ldots, T_k$, where $|S_i|, |T_i| \in \{0, 2^i\}$. When two nonempty sets with the same index $i$ arise, computation of a distinguishing set for $S_i \cup T_i$ is initiated, proceeding at a pace of $O(\log n)$ steps per insertion (this is the first part of what will be referred to as the "processing at level $i$"). At a designated time after this computation has finished, either $S_{i+1}$ or $T_{i+1}$ is replaced by $S_i \cup T_i$.

We use two predecessor data structures containing the p-keys of $S_1, \ldots, S_k$ and $T_1, \ldots, T_k$, respectively. The tricky part is to update the p-keys in these data structures such that the properties of the amortized scheme are preserved. For example, we have to make sure that old p-keys do not interfere with searches during the time it takes to delete them. In the following we describe only the universe reduction function and predecessor data structure for p-keys of $S_1, \ldots, S_k$, as everything is completely symmetric with respect to switching the roles of $S_1, \ldots, S_k$ and $T_1, \ldots, T_k$.

### 7.4.1 The universe reduction function

The universe reduction function is again (7.1), and invariants (7.3.1), (7.3.1) and (7.3.1) are still kept. However, updates are performed in a different manner. We describe the way in which $S_1, \ldots, S_k$ and $T_1, \ldots, T_k$ are updated; the corresponding distinguishing sets are implicitly updated accordingly. Keys are inserted in pairs. Just before the $p$th pair is inserted, from now on referred to as "time step $p$", we spend $O(\log n\,(\log\log n)^2)$ time on processing at each level $i$ for which $S_i$ and $T_i$ are both nonempty. The processing has two parts: computation of a distinguishing set for $S_i \cup T_i$, followed by insertions and updates of p-keys in one of the predecessor data structures (the latter part is described in Section 7.4.2). Processing at level $i$ starts at time step $2^i z + 2^{i-1}$, for $z \in \mathbb{N}$, and

proceeds for $2^{i-2}$ time steps (we make sense of half a time step by considering each time step to have two subparts).

At time step $2^i(z+1)-1$, for $z \in \mathbb{N}$, we then perform an update by moving the keys of $S_i \cup T_i$ to level $i+1$, i.e., either $S_{i+1}$ or $T_{i+1}$ is set to $S_i \cup T_i$. Note that the distinguishing set for $S_i \cup T_i$ has been computed at this point (in fact, already at time step $2^i(z+1)-2^{i-2}$). If $z$ is even, processing is about to begin at level $i+1$, and $S_i \cup T_i$ replaces the empty set at level $i+1$. Otherwise $T_{i+1} \cup S_{i+1}$ is either empty or about to move to level $i+2$, and we arbitrarily replace one of $S_{i+1}$ and $T_{i+1}$ by $S_i \cup T_i$ and the other by $\emptyset$ (this is done at all levels simultaneously). At even time steps, the new pair replaces the empty set at level 1. At odd time steps, we arbitrarily replace one of $T_1$ and $S_1$ by the newly inserted pair and the other by $\emptyset$.

An important property of the scheme described above is, that even though processing occurs at $k$ levels simultaneously, distinguishing sets that are going to appear together in the universe reduction function are computed one at a time. Specifically, consider the update at time step $2^i z - 1$, where $z > 1$ is odd. Here, new distinguishing sets for $S_1 \cup T_1, \ldots, S_i \cup T_i$ are about to become part of the universe reduction function. The distinguishing set for $S_j \cup T_j$, $j \leq i$, was computed from time step $2^i z - 2^{j-1}$ to time step $2^i z - 2^{j-2} - 1$, for $1 \leq j \leq i$. Distinguishing sets at level $i+1$ and above were computed well before time step $2^i z - 2^{i-1}$. We may thus use the method of Section 7.3 to keep invariant (7.3.1) satisfied.

### 7.4.2 Using the predecessor data structure

The predecessor data structure (for $S_1, \ldots, S_k$) still contains the p-key $q_x$ of equation (7.2) for each $x \in S_1 \cup \cdots \cup S_k$. Such p-keys are called *active*. Additionally, there will be *previous* p-keys (that used to be active but whose corresponding key is no longer in $S_1 \cup \cdots \cup S_k$) and *future* p-keys (that will become active after an upcoming change of the partition). As a consequence, there may be up to two keys corresponding to each p-key. We consider p-keys with two corresponding keys as *two* p-keys; in particular, a p-key can be both future and active, or both active and previous.

What follows is explained more easily if we switch to the language of strings. Specifically, we will look at a p-key of the form $q_k \circ 1 \circ q_{k-1} \circ 1 \circ \cdots \circ q_i \circ 1 \circ 0^{z_i}$, where $q_j \in \{0,1\}^{2j/\delta}$, as a string consisting of the characters $q_k q_{k-1} \ldots q_i$. We refer to $i$ as the *level* of the p-key. In particular, for $x \in S_i$ the level of $q_x$ is $i$. Similarly, the range of the universe reduction function is looked upon as a set of strings.

At any level we maintain the invariant that there cannot be both previous p-keys and future p-keys in the predecessor data structure. Also, future p-keys are distinct and so are previous p-keys. This means that each p-key can have at most two corresponding elements: one for which it is the current p-key, and one for which it is a previous or future p-key.

The list associated with a p-key $q$ now contains up to $2k$ keys, including:

$$A_q = \{y \in S_1 \cup \cdots \cup S_k \mid q_y \text{ is a prefix of } q\} \ . \tag{7.4}$$

Membership queries are performed by searching the associated list of the predecessor of $h(x)$ (in both predecessor data structures). The predecessor $q$ of $h(x)$ will be one of the p-keys having the longest common prefix with $h(x)$. Thus, $A_q$ contains all keys whose p-keys are prefixes of $h(x)$. In particular, if $x \in S_1 \cup \cdots \cup S_k$ the p-key $q_x$ is a prefix of $h(x)$, so $x$ is found in the associated list.

Now consider the processing initiated at level $i$ at time step $2^i z + 2^{i-1}$, $z \in \mathbb{N}$. As described earlier, the processing spans $2^{i-2}$ time steps, so the computation of a distinguishing set for $S_i \cup T_i$ is completed before time step $2^i(z+1) - 2^{i-2}$. At time step $2^i(z+1) - 1$ an update will replace a set at level $i+1$ by $S_i \cup T_i$, and hence (active) p-keys at level $i+1$ must be present for $S_i \cup T_i$ (in the predecessor data structure for $S_1, \ldots, S_k$ if $S_i \cup T_i$ replaces $S_{i+1}$, which we may assume by symmetry). A crucial point is that the future p-keys are known as soon as the distinguishing set for $S_i \cup T_i$ has been computed. This is because any changes to distinguishing sets at level $i+2$ and above that will take effect at time step $2^i(z+1) - 1$ were determined before time step $2^i z + 2^{i-1}$.

Part of the processing is to insert the future p-keys at level $i+1$. Future p-keys that will become active at time step $2^i(z+1) - 1$ and are above level $i+1$, were inserted before time step $2^i(z+1) - 2^{i-1}$. For each new p-key $q = q_k q_{k-1} \ldots q_{i+1}$ we look up any keys corresponding to prefixes $q_k$, $q_k q_{k-1}, \ldots,$ $q_k q_{k-1} \ldots q_{i+1}$, thus finding the up to $2(k-i+1)$ keys that are in $A_q$ *before and after* time step $2^i(z+1) - 1$. The keys are sorted and put into the list associated with $q$, and $q$ is inserted in the predecessor data structure. In the same way we compute new associated lists for the active p-keys at level $i+1$. The dominant part of the processing time is used for the $O(2^i \log n)$ lookups in the predecessor data structure. Hence, the total time is $O(2^i \log n \, (\log\log n)^2)$, which amounts to $O(\log n \, (\log\log n)^2)$ time per insertion.

At time step $2^i(z+1) - 1$ the active p-keys at level $i$ change status to previous, and the future p-keys at level $i+1$ change status to active. During the following $2^{i-1}$ time steps the previous p-keys at level $i$ are deleted. This means that there are no previous p-keys at time step $2^i(z+1) + 2^{i-1}$ when processing begins again at level $i$. The time per insertion for the deletions at level $i$ is negligible.

## 7.5 Open problems

An interesting open question is whether both queries and updates in a linear space deterministic dictionary can be accommodated in time $(\log n)^{o(1)}$. For example, a bound of $(\log\log n)^{O(1)}$ would mean that Sundar's lower bound is tight up to a polynomial (not considering upper bounds dependent on $w$). For $w = (\log n)^{O(1)}$ the bound is achieved by the BFAT data structure. Thus, large word length seems to be the main enemy, and new dynamic universe reduction schemes with faster updates appear a promising approach.

# Chapter 8

## Dispersing hash functions

Universal families of hash functions [CW79] are widely used in various areas of computer science (data structures, derandomization, cryptology). They have the property, among others, that any set $S$ is *dispersed* by a random function from the family. More precisely, for a universal family $F$ and any subset $S$ of the domain of its functions, if we pick a function $h$ uniformly at random from $F$, the expected value of $|S| - |h[S]|$ is not much larger than the expectation if $h$ had been a truly random function with the same range. Another way of putting this property is that a dispersing family is good at *distinguishing* the elements of any set: the average function maps the elements to many different values. For comparison, a universal family is good at distinguishing any *pair* of elements (few functions map them to the same value).

In Section 8.2 we will see that hash function families much smaller than any universal family can be dispersing. In other words, dispersion is a property strictly weaker than universality. While our first upper bound is nonconstructive, Section 8.3 explores explicit construction of small dispersing families. In particular, we exhibit a strong connection to the construction of extractors.

Small families of functions with random properties are important for derandomization (removing or reducing the use of random bits in algorithms). It is hard to characterize the situations in which a dispersing family could be used instead of a universal one. Indeed, the derandomization examples given in Section 8.4 use dispersing families in somewhat different ways than one would use universal families. We also give an example from the literature where replacing a universal family with a dispersing one immediately gives an improved result.

We will also consider a weaker form of dispersing families, where we only care about the *existence* of a single function $h$ in the family for which $|S| - |h[S]|$ is small. One special case of this has previously been intensely studied, namely perfect hash function families, where a function with $|h[S]| = |S|$ always exists. In Section 8.5 we will see that the size of existentially dispersing families explodes once we require $|S| - |h[S]|$ to be smaller than that expected for a random function $h$. In other words, storing a "near-perfect" hash function is nearly as expensive as storing a perfect one.

### Related work

The dispersion property of universal families was shown and first used in [FKS84]. It has since found application in several papers [FNSS92, LS98], and is used in Chapter 5.

Another way of stating the dispersion property of a hash function family $\{h_i\}$ is that that $\mathbf{E}_i|h_i[S]|$ should be "large". The definition of a *disperser* is similar to this in that one requires $|\cup_i h_i[S]|$ to be "large". However, in the usual treatment of dispersers, the range $R$ has size $|S|$ or smaller (whereas we will be interested in $|R| \geq |S|$), and "large" means greater than $(1-\epsilon)\,|R|$, for some choice of parameter $\epsilon$ (while we can only hope for some fraction of $|S|$). Nisan's survey [Nis96] gives a good introduction to dispersers. It also covers the stronger notion of extractors, where the requirement is near-uniformity of the random variable $h_i(x)$, for uniformly and independently chosen $h_i$ and $x \in S$. As we will see in Section 8.3, extractors are closely related to dispersing hash function families. Construction of extractors and dispersers has been intensely researched in recent years. We refer to [TSUZ01] for an overview of the results as of 2001.

Mehlhorn [Meh84] has given tight bounds (up to a constant factor) on the number of bits needed to represent perfect and universal hash functions, i.e., determined the size of such families up to a polynomial (see also [FK84,Rad92]).

### Notation

In the following, $S$ denotes a subset of $U = \{1, \ldots, u\}$, $|S| = n$, and we consider functions from $U$ to $R = \{1, \ldots, r\}$ where $r \geq n > 1$ and $u \geq 2r$. The set of all functions from $U$ to $R$ is denoted by $(U \to R)$, and $\binom{U}{n}$ denotes the subsets of $U$ of size $n$. The number of *collisions* for $S$ under $h$ is $C(S,h) = n - |h[S]|$. A uniform random choice is denoted by $\in_R$, and is always independent of other events. The base of "log" is 2, the base of "ln" is $e = 2.718\ldots$

## 8.1 The family of all functions

As preparation for the results on dispersing families, this section contains some results on the distribution of $C(S,h)$ for $h \in_R (U \to R)$. The probability that $i \notin h[S]$ is $(1-\frac{1}{r})^n$ for $i \in \{0, \ldots, r-1\}$. Thus the expected size of $R \backslash h[S]$ is $(1-\frac{1}{r})^n\, r$, and the expected size of $h[S]$ is

$$\mu := r\,(1-(1-\tfrac{1}{r})^n) = r\,(\tbinom{n}{1}/r - \tbinom{n}{2}/r^2 + \ldots) = n - \Theta(\tfrac{n^2}{r})\ . \tag{8.1}$$

Hence the expected value of $C(S,h)$ is:

$$\lambda := n - \mu = \sum_{i=1}^{n-1} \tbinom{n}{i+1}(-1/r)^i \in [\tfrac{n^2}{4r}; \tfrac{n^2}{2r}]\ . \tag{8.2}$$

We now turn to giving tail bounds. Let $S = \{s_1, \ldots, s_n\}$ and let $X_i$ be the 0-1 random variable that assumes the value 1 iff $h(s_i) \in \{h(s_1), \ldots, h(s_{i-1})\}$. Clearly $C(S,h) = \sum_i X_i$. The random variables $X_1, \ldots, X_n$ are not independent; however, they are *negatively related*:

**Definition 8.1** *(Janson [Jan93]) Indicator random variables $(I_i)_{i=0}^n$ are* negatively related *if for each $j$ there exist random variables $(J_{ij})_{i=0}^n$ with distribution equal to the conditional distribution of $(I_i)_{i=0}^n$ given $I_j = 1$, and so that $J_{ij} \leq I_i$ for every $i$.*

The random variables $Y_{ij}$ corresponding to the condition $X_j = 1$, $j > 1$, are defined in the same way as the $X_i$, except that we pick $h$ from the set of functions satisfying $h(s_j) \in \{h(s_1), \ldots, h(s_{j-1})\}$. The negative relation means that we can employ the Chernoff-style tail bounds of [Jan93, Theorem 4]. In particular, tail bound (1.9) states that, for $c \leq 1$,

$$\Pr[C(S,h) \leq c\lambda] \leq \exp(-(1-c)^2\lambda/2) \ . \tag{8.3}$$

And tail bound (1.5) gives that, for $c \geq 1$,

$$\Pr[C(S,h) \geq c\lambda] \leq \left(\frac{e^{c-1}}{c^c}\right)^{\lambda} \ . \tag{8.4}$$

Analogously, for a sequence $h_1, \ldots, h_b \in_R (U \to R)$, one can derive the following estimate, for $c \geq 1$:

$$\Pr\left[\frac{1}{b}\sum_{i=1}^{b} C(S,h_i) \geq c\lambda\right] \leq \left(\frac{e^{c-1}}{c^c}\right)^{\lambda b} \ . \tag{8.5}$$

## 8.2 Dispersing families

**Definition 8.2** *A family of functions $F \subseteq (U \to R)$, where $|U| = u$ and $|R| = r$, is* $(c,n,r,u)$-dispersing *if for any $S \subseteq U$, $|S| = n$, the expected value of $C(S,h)$ for $h \in_R F$ is at most $c\lambda$, where $\lambda$ is given by (8.2). When parameters $n$, $r$ and $u$ follow from the context, we shall use the term* $c$-dispersing.

By definition of $\lambda$, the family $(U \to R)$ is 1-dispersing. Thus, parameter $c$ is a measure of how well a random function from a family disperses compared to a truly random function.

We now give a simple nonconstructive argument (using the probabilistic method) that a small family of $c$-dispersing functions exists for $c \geq 1+\epsilon$, where $\epsilon > 0$ is a constant. (The requirement on $c$ ensures that the constant factor of the bound does not depend on $c$.) Let $k(c) = \ln(c^c/e^{c-1}) = \Theta(c \log c)$. The family can be constructed by picking $h_1, \ldots, h_b \in_R (U \to R)$, where $b > \frac{n \ln(ue/n)}{k(c)\lambda} = O(\frac{r \log(u/n)}{n\, c \log c})$. With nonzero probability this gives a family with the desired property, namely $\frac{1}{b}\sum_{i=1}^{b} C(S,h_i) \leq c\,\lambda$ for any $S \in \binom{U}{n}$. By inequality (8.5),

$$\Pr\left[\frac{1}{b}\sum_{i=1}^{b} C(S,h_i) \geq c\lambda\right] \leq \left(\frac{e^{c-1}}{c^c}\right)^{\lambda b} < (ue/n)^{-n} \ .$$

Since there are less than $(ue/n)^n$ sets in $U$ of size $n$ (see, e.g., [Juk00, Section 1.1]), the probability of failure for at least one set is less than 1, as desired.

We now show a lower bound on the size of a $c$-dispersing family. Take any such family $F = \{h_1, \ldots, h_b\}$. We construct $U_1, \ldots, U_k$, with $U_k \subseteq U_{k-1} \subseteq \cdots \subseteq U_1 \subseteq U_0 = U$ such that $|U_i| \geq u\,(n/2r)^i$ and $|h_i[U_k]| \leq n/2$ for $i \leq k$. The set $U_i$ is constructed from $U_{i-1}$ using the pigeonhole principle to pick a subset with $|h_i[U_i]| \leq n/2$ of size at least $|U_{i-1}|/(2r/n)$. Setting $k = \lfloor \log(u/n)/\log(2r/n) \rfloor$ we have $|U_k| \geq n$ and can take $S \subseteq U_k$ of size $n$. Since $F$ is $c$-dispersing we must have $\sum_i C(S, h_i) \leq b\,c\,\lambda$. On the other hand, by construction $\sum_i C(S, h_i) \geq k\,n/2$, so we must have:

$$b \geq \frac{k\,n}{2\,c\,\lambda} \geq \frac{r \log(u/n)}{2\,n\,c \log(2r/n)} \quad .$$

We summarize the bounds as follows:

**Theorem 8.1** *For $r \geq n > 1$, $u \geq 2r$ and $c > 1 + \epsilon$, for constant $\epsilon > 0$, a minimal size $(c, n, r, u)$-dispersing family $F$ satisfies:*

$$\frac{r \log(u/n)}{2n\,c \log(2r/n)} \leq |F| = O\left(\frac{r \log(u/n)}{n\,c \log c}\right) \quad .$$

The gap between the bounds is $O(\frac{\log(2r/n)}{\log c})$. In a certain sense we have a tight bound in most cases: Parameter $c$ ranges from $1 + \epsilon$ to $O(r/n)$, and for $c = (r/n)^{\Omega(1)}$ the bounds differ by a constant factor.

A random function $h$ from a $(\frac{\delta n}{\lambda}, n, r, u)$-dispersing family has expected size of $h[S]$ at least $(1 - \delta)\,n$. This makes the following version of Theorem 8.1 convenient.

**Corollary 8.1** *For $r \geq n > 1$, $u \geq 2r$ and $\delta > (1 + \epsilon)\,\lambda/n$, for constant $\epsilon > 0$, a minimal size $(\frac{\delta n}{\lambda}, n, r, u)$-dispersing family $F$ satisfies:*

$$\frac{\log(u/n)}{2\delta \log(2r/n)} \leq |F| = O\left(\frac{\log(u/n)}{\delta \log(4\delta r/n)}\right) \quad .$$

### 8.2.1 An impossibility result

We have seen examples of $c$-dispersing families for $c \geq 1$. A natural question is whether such families exist for $c < 1$. This section supplies a generally negative answer for any constant $c < 1$. However, it is possible to disperse *slightly* more than a totally random function by using the family of all "evenly distributing" functions. This is analogous to universal hash functions, where it is also possible to improve marginally upon the performance of a truly random function [Sar80, Woe99].

**Example 8.1** *Consider the case $n = r = 3$, $u = 6$, where $\lambda = 8/9$. If we pick a function at random from those mapping two elements of $U$ to each element in the range, the expected number of collisions is $3/5$. That is, this family is $27/40$-dispersing.*

We need the following special case of a more general result shown in Section 8.5.

**Lemma 8.1** *For $c < 1$ let $k(c) = \frac{(1-c)^2}{100\log(4/(1-c))}$. Assume $r \leq k(c)\,n^2$ and $u \geq \frac{100\,r}{1-c}$, and let $h : U \to R$ be any function. For $S \in_R \binom{U}{n}$ we have $\Pr[C(S,h) \leq \frac{1+c}{2}\lambda] < 1 - \frac{2c}{1+c}$ .*

**Corollary 8.2** *For $0 < c < 1$, $r \leq k(c)\,n^2$ and $u \geq \frac{100\,r}{1-c}$, no $(c,n,r,u)$-dispersing family exists.*

*Proof.* Suppose $F$ is a $(c,n,r,u)$-dispersing family with parameters satisfying the above. By an averaging argument, there must exist a function $h \in F$ such that for $S \in_R \binom{U}{n}$ the expected value of $C(S,h)$ is at most $c\lambda$. In particular, Markov's inequality gives that the probability of $C(S,h) \leq \frac{1+c}{2}\lambda$ must be at least $1 - \frac{2c}{1+c}$, contradicting Lemma 8.1. □

## 8.3 Explicit constructions

This section concerns the explicit construction of dispersing families, concentrating on $O(1)$-dispersing families. By explicit families $F_{c,n,r,u}$ we mean that there is a Turing machine that, given parameters $c$, $n$, $r$ and $u$, the number of some function $f$ in (an arbitrary ordering of) $F_{c,n,r,u}$, and $x \in U$, computes $f(x)$ in time $(\log|F_{c,n,r,u}| + \log(u+c))^{O(1)}$. The general goal, only reached here for some parameters, would be explicit families of sizes polynomially related to the bounds of Theorem 8.1. Picking a random function from such a family uses a number of random bits that is within a constant factor of optimal, i.e., the *sample complexity* is optimal.

### 8.3.1 Universal families

**Definition 8.3** *A family $F \subseteq (U \to R)$ is* $c$-universal *if for any $x, y \in U$, $x \neq y$ and $h \in_R F$, $\Pr[h(x) = h(y)] \leq c/r$. It is* strongly $c$-universal *if for any $a, b \in R$, $\Pr[h(x) = a,\, h(y) = b] \leq c/r^2$.*

One strongly $(1+\epsilon)$-universal (and thus $(1+\epsilon)$-universal) family for $0 < \epsilon \leq 1$ is:

$$F_{\text{su}} = \{x \mapsto ((t\,x + s) \bmod p) \bmod r \mid m/2 \leq p \leq m,\ p \text{ prime},\ 0 \leq s, t < p\}$$

where $m = 24\,r^2\log(u)/\epsilon$. The universality proof is given in appendix 8.7. Note that the family has size $(r\log(u)/\epsilon)^{O(1)}$, and that, given parameters $s$, $t$ and $p$, we can compute a function value in polynomial time. As for universal families with parameter $c \geq 2$, we note that taking *any* $2/c$ fraction of a 2-universal family yields a $c$-universal family.

We now establish that universal families are dispersing, slightly generalizing an observation in [FKS84]:

**Proposition 8.1** *A $c$-universal family is $2c$-dispersing.*

*Proof.* Let $F$ be a $c$-universal family, and take $S \in \binom{U}{n}$. For $h \in_R F$ consider $K(S,h) = |\{\{x,y\} \in \binom{S}{2} \mid h(x) = h(y)\}|$. Since $C(S,h) \leq K(S,h)$ we just need to bound the expected value of $K(S,h)$. By $c$-universality this is at most $\binom{n}{2} c/r$, and by (8.2) we have the bound $\binom{n}{2} c/r < c\,n^2/2r \leq 2\,c\,\lambda$. □

Mehlhorn [Meh84, Theorem D] has shown that a $c$-universal family can have size no smaller than $r\,(\lceil \log u / \log r \rceil - 1)/c$. This is $\Omega(n \log c / \log r)$ times larger than the upper bound of Theorem 8.1. Hence, dispersion is a property strictly weaker than universality. The minimum sizes of $c$-universal and $O(c)$-dispersing families are polynomially related when $r/n \geq n^{\Omega(1)} + c^{1+\Omega(1)}$. Under the same condition, the family $F_{\mathrm{su}}$, as well as a $c$-universal subfamily constructed as stated above, has size polynomially related to the minimum. In particular, we have explicit $O(1)$-dispersing families of optimal sample complexity for $r = n^{1+\Omega(1)}$.

### 8.3.2 Extractor based construction

This section addresses the construction of $O(1)$-dispersing families for $r = n^{1+o(1)}$, where universal families do not have optimal sample complexity (except for very large universes). We give an explicit construction of an $O(1)$-dispersing family from an an *extractor* (see definition below). Plugging in an explicit optimal extractor would yield an explicit $O(1)$-dispersing family with optimal sample complexity (except perhaps for very small universes). We need only consider the case where $r$ is a power of two, since such families are also $O(1)$-dispersing for ranges up to two times larger.

**Definition 8.4** *A random variable $X$ with values in a finite set $T$ is* $\epsilon$-close to uniform *if*

$$\sum_{i \in T} |\Pr[X = i] - 1/|T|| \leq 2\,\epsilon.$$

**Definition 8.5** *A function $E : U \times \{0,1\}^s \to \{0,1\}^t$ is an* $(n,\epsilon)$-extractor *if for any $S \in \binom{U}{n}$, the distribution of $E(x,y)$ for $x \in_R S$, $y \in_R \{0,1\}^s$ is $\epsilon$-close to uniform over $\{0,1\}^t$.*

Nonconstructive arguments show that for $t \leq \log n$ and $\epsilon > 0$ there exist $(n,\epsilon)$-extractors with $s = O(\log(\log(u)/\epsilon))$. As mentioned in the introduction, much research effort is currently directed towards explicit construction of such functions.

**Theorem 8.2** *Suppose that $r$ is a power of 2, $E : U \times \{0,1\}^s \to \{0,1\}^t$ is an $(\lfloor n/2 \rfloor, \epsilon)$-extractor, where $\epsilon = O(n/r)$, $F' \subseteq (U \to \{0,1\}^s)$ is strongly $(1+\epsilon)$-universal, and $F'' \subseteq (U \to \{0,1\}^{\log(r)-t})$ is 2-universal. Then the family*

$$F_1 = \{x \mapsto E(x, f'(x)) \circ f''(x) \mid f' \in F',\ f'' \in F''\} \subseteq (U \to R)$$

*where $\circ$ denotes bit string concatenation, is $O(1)$-dispersing.*

*Proof.* Let $S \in \binom{U}{n}$. By the extractor property, the distribution of $E(x,z)$ for $x \in_R S$ and $z \in_R \{0,1\}^s$ is $\epsilon$-close to uniform. We can therefore identify a set $B \subseteq \{0,1\}^t$ of "bad" points, such that $|B| \leq \epsilon\, 2^t$ and for $i \in \{0,1\}^t \backslash B$ and $x \in_R S$ we have:

$$\Pr_{z \in_R \{0,1\}^s}[E(x,z) = i] \leq 2^{-t+1} \ . \tag{8.6}$$

Also note that the distribution of $E(x, f'(x))$ for $x \in_R S$ and $f' \in_R F'$ must be $\gamma$-close to uniform for $\gamma = O(\epsilon)$. We choose $f' \in_R F'$, $f'' \in_R F''$, set $h(x) = E(x, f'(x)) \circ f''(x)$, and bound the expected value of $C(S,h)$:

$$\begin{aligned}\mathbf{E}[C(S,h)] \leq\ & \mathbf{E}[|\{x \in S \mid E(x,f'(x)) \in B\}|] + \\ & \mathbf{E}[|\{\{x_1,x_2\} \in \tbinom{S}{2} \mid h(x_1) = h(x_2) \wedge E(x_1,f'(x_1)) \notin B \wedge E(x_2,f'(x_2)) \notin B\}|] \ .\end{aligned} \tag{8.7}$$

For $x \in_R S$ the first term is:

$$n \Pr[E(x,f'(x)) \in B] \leq (\gamma + \epsilon)\, n = O(n^2/r) \ . \tag{8.8}$$

For $\{x_1,x_2\} \in_R \binom{S}{2}$, the second term is:

$$\begin{aligned}&\tbinom{n}{2} \Pr[h(x_1) = h(x_2) \wedge E(x_1,f'(x_1)) \notin B \wedge E(x_2,f'(x_2)) \notin B] \\ &\quad = \tbinom{n}{2} \sum_{i \in \{0,1\}^t \backslash B} \Pr[E(x_1,f'(x_1)) = E(x_2,f'(x_2)) = i \wedge f''(x_1) = f''(x_2)] \\ &\quad \leq \tbinom{n}{2} 2^{-\log(r)+t+1} \sum_{i \in \{0,1\}^t \backslash B} \Pr[E(x_1,f'(x_1)) = E(x_2,f'(x_2)) = i] \ .\end{aligned} \tag{8.9}$$

To bound $\Pr[E(x_1,f'(x_1)) = E(x_2,f'(x_2)) = i]$ for $i \in \{0,1\}^t \backslash B$ we note that the random choice $\{x_1,x_2\} \in_R \binom{S}{2}$ can be thought of in the following way: First choose $S_1 \in_R \binom{S}{\lfloor n/2 \rfloor}$ and then choose $x_1 \in_R S_1$, $x_2 \in_R S \backslash S_1$. By symmetry, this procedure yields the same distribution. Since for any $S_1$, we choose $x_1$ and $x_2$ independently from disjoint sets of size at least $\lfloor n/2 \rfloor$, we have:

$$\begin{aligned}&\Pr_{f' \in_R F'}[E(x_1,f'(x_1)) = E(x_2,f'(x_2)) = i] \\ &\qquad \leq (1+\epsilon) \Pr_{z_1,z_2 \in_R \{0,1\}^s}[E(x_1,z_1) = E(x_2,z_2) = i] \\ &\qquad = (1+\epsilon) \Pr_{z_1 \in_R \{0,1\}^s}[E(x_1,z_1) = i] \Pr_{z_2 \in_R \{0,1\}^s}[E(x_2,z_2) = i] \\ &\qquad \leq (1+\epsilon) \left(\tfrac{n}{\lfloor n/2 \rfloor}\, 2^{-t+1}\right)^2 \ .\end{aligned} \tag{8.10}$$

The factor of $\frac{n}{\lfloor n/2 \rfloor}$ relative to (8.6) is due to the fact that $x_1$ and $x_2$ are sampled from a fraction $\geq \frac{\lfloor n/2 \rfloor}{n}$ of $S$. Plugging this into (8.9) we obtain:

$$\begin{aligned}&\tbinom{n}{2} \Pr[h(x_1) = h(x_2) \wedge E(x_1,f'(x_1)) \notin B \wedge E(x_2,f'(x_2)) \notin B] \\ &\qquad \leq n^2\, 2^{-\log(r)+t}\, 2^t\, (1+\epsilon) \left(\tfrac{n}{\lfloor n/2 \rfloor}\, 2^{-t+1}\right)^2 \\ &\qquad \leq 36\, (1+\epsilon)\, n^2/r = O(n^2/r) \ .\end{aligned} \tag{8.11}$$

Hence, the expected value of $C(S,h)$ is $O(n^2/r)$, as desired. □

**Corollary 8.3** *Given an explicit* $(\lfloor n/2 \rfloor, \epsilon)$*-extractor* $E : U \times \{0,1\}^s \to \{0,1\}^t$ *with* $\epsilon = O(n/r)$ *and* $t = \log n - O(1)$*, there is an explicit* $O(1)$*-dispersing family with sample complexity* $O(\log(r \log(u)/n) + s)$*.*

*Proof.* We use the construction of Section 8.3.1 for the universal families of the above construction. The number of bits needed to sample $f' \in_R F'$ is $O(\log(2^s \log(u)/\epsilon)) = O(s + \log(r \log(u)/n))$. The number of bits needed to sample $f'' \in F''$ is $O(\log(2^{\log(r)-t} \log u)) = O(\log(r \log(u)/n))$. □

Of course, Theorem 8.2 and Corollary 8.3 are trivial in cases where the $O(1)$-parameter of the dispersing family is bigger than $n/\lambda = O(r/n)$. In these cases we can get a nontrivial family by using Corollary 8.3 to obtain an $O(1)$-dispersing family with range $\{1, \ldots, r'\}$, where $r'/r$ is a suitably large constant power of 2. To get the family $F$ with range $R$, simply cut away $\log(r'/r)$ bits of the output. This only decreases sizes of images of sets by a constant factor, which means that for $h \in_R F$ and $S \in \binom{U}{n}$ the expected size of $h[S]$ is still $\Omega(n)$.

#### Explicit extractors

The best explicit extractor in current literature with the parameters required by Corollary 8.3 has seed length $s = O((\log\log(ur/n))^3)$ [RRV99b]. For constant $\epsilon$ (applicable when $r = O(n)$), the best known seed length is $O(\log\log u + (\log\log n)^{2+o(1)})$ [TSUZ01]. In particular, when $\log\log u \geq (\log\log n)^{2+\Omega(1)}$ this gives a $O(1)$-dispersing family of hash functions with $r = O(n)$ that has size $(\log u)^{O(1)}$, which is polynomial in the lower bound of Theorem 8.1.

### 8.3.3 Extractors from dispersing families

This section points out that nontrivial $O(1)$-dispersing families with range $\{0,1\}^{\log n}$ are also $(n, \epsilon)$-extractors for a constant $\epsilon < 1$.

**Proposition 8.2** *Let* $\{h_z\}_{z \in \{0,1\}^s} \subseteq (U \to \{0,1\}^t)$*,* $t = \log n - O(1)$*, be such that for* $z \in_R \{0,1\}^s$ *the minimum expected size of* $h_z[S]$*, for* $S \in \binom{U}{n}$*, is* $\Omega(n)$*. Then* $E(x,z) = h_z(x)$ *is an* $(n, 1 - \Omega(1))$*-extractor.*

*Proof.* Let $S \in \binom{U}{n}$. For $x \in_R S$ and $z \in_R \{0,1\}^s$, let $B \subseteq \{0,1\}^t$ consist of the values $i \in \{0,1\}^t$ for which $\Pr[h_z(x) = i] < 2^{-t}$. We have:

$$\begin{aligned}\sum_{i \in B} (2^{-t} - \Pr[h_z(x) = i]) &= 1 - \sum_{i \in \{0,1\}^t} \min\{\Pr[h_z(x) = i], 2^{-t}\} \\ &\leq 1 - \mathbf{E}[|h_z[S]|]/2^t = 1 - \Omega(1) \ .\end{aligned}$$

This implies that the distribution of $E(x,z)$ is $(1 - \Omega(1))$-close to uniform. □

It should be noted that there is an efficient explicit way of converting an extractor with nontrivial constant error into an extractor with almost any smaller error [RRV99a]. Unfortunately, this conversion slightly weakens other parameters, so the problem of constructing optimal extractors with small error cannot be said to be quite the same as that of constructing optimal dispersing families.

## 8.4 Applications

The model of computation used for our applications is a unit cost RAM with word size $w$. We assume that the RAM supports dispersing families, i.e., given the parameters of a dispersing family, a "function number" $i$ and an input $x$, it can compute $f_i(x)$ in constant time, where $f_i$ is the $i$th function from a dispersing family with the given parameters and optimal sample complexity. The RAM also has an instruction to generate random numbers.

### 8.4.1 Relational joins

Suppose we have two lists of at most $n$ machine words in total, each of which has no duplicate elements. We consider the problem of identifying all elements that occur in both lists. This task is known as a *relational join*, and is a fundamental task in databases. Using a relational join one can also determine whether two sets are identical, and whether one is a subset of the other.

Comparison-based algorithms for these problems require $\Omega(n \log n)$ time, and performing a relational join is easily reduced to sorting. The currently best deterministic linear space sorting algorithm runs in time $O(n \log \log n)$ [Han02]. Using randomization and universal hashing the time for relational joins can be improved to $O(n)$, expected, using linear space. The number of random bits used for this is $\Omega(\log n + \log w)$. We now show how to decrease the number of random bits to $O(\log w)$, and still solve the problem in expected $O(n)$ time and linear space, using dispersing hash functions.

Pick a function $h$ at random from a $(n/\log n, n, n^2, 2^w)$-dispersing family. According to Theorem 8.1 there is such a family of size $O(w)$, which means that the sample complexity is $O(\log w)$ bits. The algorithm goes as follows. First compute the function values under $h$ for all elements in the two lists, and sort the elements into buckets according to their hash function values. This can be done in time $O(n)$ using radix sort. All duplicates in buckets with at most two elements can be identified by a linear time scan. We then list the elements in buckets of size more than two, and sort them using an $O(n \log n)$ time sorting algorithm. As a last step, we go through the sorted list and report all duplicates.

The correctness of the algorithm is straightforward. It remains to be argued that the expected running time is $O(n)$. Let $S_1$ and $S_2$ be the two sets of elements. For the sake of analysis let $S'$ be a set of $n - |S_1 \cup S_2|$ elements, disjoint from $S_1$ and $S_2$. By definition of dispersing families, the expected size of $h[S_1 \cup S_2 \cup S']$ is $n - O(n/\log n)$. In particular, the number of elements in the set $S_1 \cup S_2$ that share their function value with another element in $S_1 \cup S_2$ is $O(n/\log n)$, expected. Since the elements in the two lists are distinct, this means that the expected number of elements in buckets of size more than two is $O(n/\log n)$, meaning that the sorting step takes $O(n)$ expected time.

### 8.4.2 Element distinctness

We now consider the element distinctness problem for a list of $n$ machine words. Again, in a comparison-based model this problem requires time $\Omega(n \log n)$, and the problem can be reduced to sorting. As before, with universal hashing the problem can be solved in expected linear time, and linear space. We show how to decrease the number of random bits to $O(\log w)$, using dispersing hash functions.

Again, pick $h$ at random from a $(n/\log n, n, n^2, 2^w)$-dispersing family of size $O(w)$. Arguing as above, the expected size of $h[S]$ is $|S| - O(n/\log n)$, where $S$ is the set of machine words considered. Words involved in a collision are put into a balanced binary search tree, each in time $O(\log n)$. Since no more than $2(|S| - |h[S]|)$ elements can be inserted before a duplicate (if any) is found, this takes time $O((|S| - |h[S]|) \log n)$, which is expected $O(n)$.

**Remark.** It would be interesting if the more general problem of determining the number of distinct elements in a list of machine words (the set cardinality problem), could be solved in a similar way. Possibly, a slightly stronger property than dispersion is needed.

### 8.4.3 Static dictionary construction

The next problem considered is that of constructing a static dictionary, i.e., a data structure for storing a set $S \subseteq U = \{0,1\}^w$, $|S| = n$, allowing constant time lookup of elements (plus any associated information) and using $O(n)$ words of memory. The set is supposed to be given as an array of $n$ distinct elements. The best known deterministic algorithm runs in time $O(n \log n)$, see Chapter 6. Randomized algorithms running in time $O(n)$, can be made to use as few as $O(\log n + \log w)$ random bits [DfGMP92]. Here we will show how to achieve another trade-off, namely expected time $O(n \log^\epsilon n)$, for any constant $\epsilon > 0$, using $O(\log w)$ random bits.

#### Randomized universe reduction

Picking random functions from a $(n/\log n, n, n^2, 2^w)$-dispersing family of size $O(w)$, we can first reduce our problem to two subproblems: one within a universe of size $n^2$ and one with $O(n/\log n)$ colliding elements. The set of $O(n/\log n)$ elements can be handled using the deterministic $O(n \log n)$ algorithm. The set within a universe of size $n^2$ is handled by an algorithm described below. Note that when randomly picking the above hash functions, there is a positive constant probability of finding one reducing the problem to two subproblems with the desired parameters. Thus, the expected number of attempts before suitable functions are found is constant. The expected number of random bits needed to find a suitable reduction function is $O(\log w)$, and the reduction takes linear expected time.

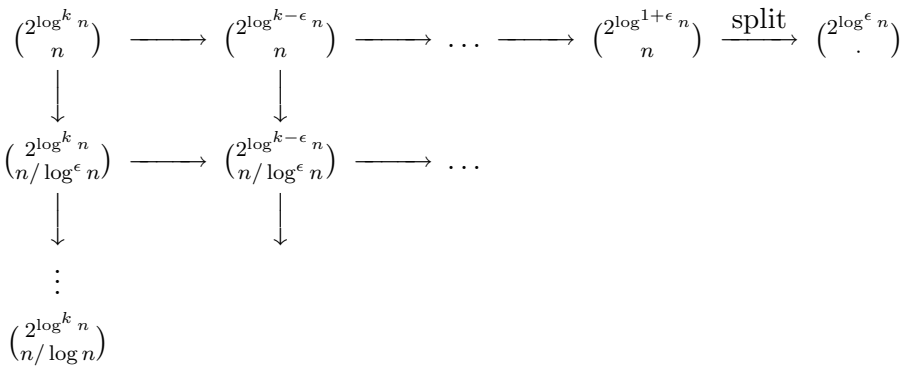


Figure 8.1: Overview of subproblems for the deterministic dictionary construction algorithm.

### A deterministic algorithm for "small" universes

By the above, we only need to deal with the case $w \leq 2 \log n$. However, we will make the weaker assumption that $w = \log^k n$ for some constant $k$. Let $\epsilon > 0$ be arbitrary. We start with a $(\frac{2^{\log^{k-\epsilon} n}}{n \log^{\epsilon} n}, n, 2^{\log^{k-\epsilon} n}, 2^w)$-dispersing family of size $O(\log^{O(\epsilon)} n)$. Running through the functions we find in time $O(n \log^{O(\epsilon)} n)$ a function $h$ such that $|h[S]| \geq n - n/\log^{\epsilon} n$. The size of $h[S]$ can be found by sorting in time $O(n \log \log n)$ [Han02]. Now choose $S_1 \subseteq S$ maximally such that $h$ is 1-1 on $S$. We have reduced our problem to two subproblems: A dictionary for $S \backslash S_1$ (which has size at most $n/\log^{\epsilon} n$) and a dictionary for $h[S_1]$ (which is contained in a universe of size $2^{\log^{k-\epsilon} n}$). For the $S_1$ dictionary, we need to store the original elements as associated information, since $h$ is not 1-1 on $U$. The subproblems are split in a similar way, recursively. After a constant number of steps (taking $O(n \log^{O(\epsilon)} n)$ time), each subproblem either has $O(n/\log n)$ elements or a universe of size at most $2^{\log^{1+\epsilon} n}$. All the dictionaries for small sets can be constructed in $O(n)$ time using the $O(n \log n)$ algorithm. The small universes can be split into $n$ parts of size $2^{\log^{\epsilon} n}$. Using the $O(n \log n)$ algorithm on each part takes total time $O(n \log^{\epsilon} n)$. The "translations" using dispersing functions are sketched in Figure 8.1.

We sum up our derandomization results as follows:

**Theorem 8.3** *On a unit cost RAM supporting dispersing families we can, using $O(\log w)$ random bits, expected, and space $O(n)$:*

- *Perform relational join in expected time $O(n)$.*
- *Solve the element distinctness problem in expected time $O(n)$.*
- *Construct a static dictionary in time $O(n \log^{\epsilon} n)$, for any constant $\epsilon > 0$.*

### 8.4.4 An implicit dictionary

As a final application, we consider the implicit dictionary scheme of Fiat et al. [FNSS92]. In an implicit dictionary, the elements of $S$ are placed in an array of $n$ words. A result of Yao states that without extra memory, a lookup requires $\log n$ table lookups in the worst case [Yao81]. The question considered in [FNSS92] is how little extra memory is needed to enable constant worst case lookup time. The information stored outside the table in their construction is the description of (essentially) a universal hash function, occupying $O(\log n + \log w)$ bits. However, the only requirement on the function is that it is, say, $(2, n, O(n), 2^w)$-dispersing, so we can reduce the extra memory to $O(\log w)$ bits. (This result was also shown in a follow-up paper [FN93], using an entirely different construction.)

## 8.5 Existentially dispersing families

By the result of Section 8.2.1, we cannot expect $C(S,h) \leq \lambda/2$ (or better) when picking $h$ at random from some family. But there are situations in which such functions are of interest, e.g., in perfect hashing. This motivates the following weaker notion of dispersion.

**Definition 8.6** *A family $F \subseteq (U \to R)$ is* existentially $(c, n, r, u)$-dispersing *if, for any $S \subseteq U$, $|S| = n$, there exists $h \in F$ with $C(S,h) \leq c\,\lambda$, where $\lambda$ is given by (8.2). When parameters $n$, $r$ and $u$ follow from the context, we shall use the term* existentially $c$-dispersing.

Existentially 0-dispersing families are of course better known as *perfect* families. It is well known that perfect hash function families have program size $\Theta(n^2/r + \log n + \log\log_r u)$ [Meh84] (i.e., the description of a perfect hash function requires this number of bits). In this section we investigate the "nearly perfect" families between existentially 0-dispersing and existentially 1-dispersing.

### 8.5.1 A dual tail bound

We will need a tail bound "dual" to the one in (8.3). Specifically, $h$ is fixed to some function, and we consider $C(S,h)$ for $S \in_R \binom{U}{n}$. We want to upper bound the probability that $C(S,h) \leq c\lambda$, for $c < 1$. First a technical lemma:

**Lemma 8.2** *Take $0 < c < 1$ and $k \in \mathbf{N}$ with $k < (1-c)\,n^2/4r$. For any $h \in (U \to R)$, when picking $S \in_R \binom{U}{n}$ we have*

$$\Pr[C(S,h) \leq c\lambda] \leq (\tfrac{5}{2}\,n^2/ku)^k + \exp(-(1 - c - 4\,k\,r/n^2)^2\,n^2/8r) \ .$$

*Proof.* In this proof we will use the inequalities of Section 8.1 with various values substituted for $r$, $n$ and $c$ ($\lambda$ is a function of these). We use primed variables to denote the substitution values.

We consider the random experiment in which $n+k$ elements $Y_1, \ldots, Y_{n+k}$ are picked randomly and independently from $U$. It suffices to bound the probability

of the event "$|\{Y_1, \ldots, Y_{n+k}\}| < n$ or $|h(\{Y_1, \ldots, Y_{n+k}\})| \geq n - c\,\lambda$", which occurs with greater probability than $C(S,h) \leq c\,\lambda$. (Given that $|\{Y_1, \ldots, Y_{n+k}\}| \geq n$, the sets of $\binom{U}{n}$ are contained in $\{Y_1, \ldots, Y_{n+k}\}$ with the same positive probability.) The probability that $|\{Y_1, \ldots, Y_{n+k}\}| < n$ is at most $e^{-n^2/4u}(\frac{e\,(n+k)^2}{2\,k\,u})^k$, by use of (8.4) with parameters $r' = u$, $n' = n+k$ and $c' = k/\lambda'$. Since $k < n/4$, we have the bound $(\frac{5}{2}\,n^2/ku)^k$.

To bound the probability that $|h(\{Y_1, \ldots, Y_{n+k}\})| \geq n - c\,\lambda$, first notice that without loss of generality we can assume the function values $h(Y_i)$ to be uniformly distributed over $R$ — this can only increase the probability. Now (8.3) can be used with $r' = r$, $n' = n + k$ and $c' = (c\lambda + k)/\lambda' \leq c + 4\,k\,r/n^2$: The probability of $|h(\{Y_1, \ldots, Y_{n+k}\})| \geq n - c\,\lambda$ is at most $\exp(-(1-c')^2\lambda'/2) \leq \exp(-(1 - c - 4\,k\,r/n^2)^2\,n^2/8r)$. □

We can now show the dual tail bound, which also implies Lemma 8.1.

**Lemma 8.3** *Suppose $0 < c < 1$, $r \leq \frac{(1-c)^2}{25}\,n^2$ and $u \geq \frac{100}{1-c}\,r$. For any $h \in (U \to R)$, when picking $S \in_R \binom{U}{n}$ we have*

$$\Pr[C(S,h) \leq c\lambda] \leq 2^{-\frac{(1-c)\,n^2}{20\,r}} + e^{-\frac{(1-c)^2 n^2}{25\,r}} < 1 \ .$$

*In particular,*

$$\Pr[C(S,h) \leq c\lambda] = 2^{-\Omega((1-c)^2 n^2/r)} \ .$$

*Proof.* We choose a positive integer $k \in (\frac{(1-c)\,n^2}{20\,r}; \frac{(1-c)\,n^2}{10\,r}]$ and apply Lemma 8.2. Since $u \geq \frac{100}{1-c}\,r$, we have $(\frac{5}{2}\,n^2/ku)^k \leq 2^{-k} < 2^{-\frac{(1-c)\,n^2}{20\,r}}$. By choice of $k$, $(1 - c - 4k\,r/n^2)^2/8 \geq \frac{(1-c)^2}{25}$. Finally, $\frac{(1-c)^2}{25}\,n^2/r \geq 1$, so the sum is at most $2^{-1} + e^{-1} < 1$. The second inequality follows directly. □

**Proof of Lemma 8.1.** Applying Lemma 8.3, we see that

$$\Pr[C(S,h) \leq \tfrac{1+c}{2}\lambda] < 2^{-\frac{(1-c)\,n^2}{40\,r}} + e^{-\frac{(1-c)^2\,n^2}{100\,r}} \leq 2^{1-\frac{(1-c)^2\,n^2}{100\,r}} \ .$$

For this to be less than $1 - \frac{2c}{1+c} > (1-c)/2$, it suffices that $\frac{(1-c)^2 n^2}{100\,r} \geq \log(\frac{4}{1-c})$. The lemma follows by isolating $r$.

### 8.5.2 Program size of existentially dispersing hash functions

Given Lemma 8.3, a lower bound on the program size of existentially $c$-dispersing families, for $c < 1$, follows.

**Theorem 8.4** *For any constant $c$, $0 < c < 1$, there exist further constants $k_1, k_2, k_3 > 0$ such that for $u \geq k_1\,r$ and $r \leq k_2\,n^2$, elements of an existentially $(c, n, r, u)$-dispersing family $F$ cannot be stored using less than*

$$\max(k_3\,n^2/r,\ \log\lfloor \log(u/n)/\log(2\,r/n)\rfloor) \ \textit{bits.}$$

*Proof.* Take $k_1 = 100/(1-c)$, $k_2 = (1-c)^2/25$. Then by Lemma 8.3 there exists $k_3 > 0$ such that any function $h \in (U \to R)$, less than a fraction $2^{-k_3 n^2/r}$ of $S \in \binom{U}{n}$ has $C(S,h) \leq n\,c\,\lambda$. Since for each $S \in \binom{U}{n}$ there must be a function $h \in F$ with $C(S,h) \leq c\lambda$, we must have $|F| \geq 2^{k_3 n^2/r}$.

The lower bound of $\log\lfloor \log(u/n)/\log(2r/n)\rfloor$ stems from the observation that $S$ in the lower bound proof of Section 8.2 is constructed to have an image of size at most $n/2$ for $\lfloor \log(u/n)/\log(2r/n)\rfloor$ arbitrary functions. □

The bound of the theorem is within a constant factor of the upper bound on the size of perfect families when $r = O(n^2/\log n)$. On the other hand, for $r > \binom{n}{2}$ and $c < 1$, a $c$-dispersing family must be perfect, so the two notions coincide. So, except possibly for the range $n^2/\log n < r < \binom{n}{2}$, being "nearly perfect" is nearly as hard as being perfect.

## 8.6 Open problems

The most obvious open problem is to find explicit dispersing families of close to minimal size for a wider range of parameters. As shown in Section 8.3.2, explicit $O(1)$-dispersing families with optimal sample complexity will follow if optimal extractors are constructed, and such families are themselves extractors with nontrivial error. The issue of constructing explicit $c$-dispersing families with parameter $c$ close to $r/n$ is interesting in view of the derandomization applications given in Section 8.4.

Another issue is that only quite weak lower bounds are known on the size of existentially $c$-dispersing families for $c > 1$. The best upper bounds are the same as for $c$-dispersing families, but can existentially dispersing families be even smaller? If so, complete derandomizations such as the dictionary construction algorithm in Section 8.4.3 could be made faster by employing a smaller existentially dispersing family.

## 8.7 Appendix: Universality proof

This appendix gives a proof of strong $(1+\epsilon)$-universality, for arbitrarily small $\epsilon > 0$, of a small explicit family of functions. To the knowledge of the author, such a proof never appeared explicitly in the literature. However, the proof is along the lines of the universality proof in [FKS84].

**Theorem 8.5** *For $0 < \epsilon \leq 1$ and $m \geq 24\,r^2 \log(u)/\epsilon$ the family*

$$F_{\text{su}} = \{x \mapsto ((t\,x + s) \bmod p) \bmod r \mid m/2 \leq p \leq m,\ p \text{ prime},\ 0 \leq s,t < p\}$$

*is strongly $(1+\epsilon)$-universal.*

*Proof.* (Sketch) For any $a, b \in R$ and $x, y \in U$, where $x \neq y$, and $h \in_R F_{\text{su}}$ we must show that we have $\Pr[h(x) = a \wedge h(y) = b] \leq (1+\epsilon)/r^2$. So pick a random prime $p$ in the range $m/2$ to $m$ and $s, t \in_R \{0, \ldots, p-1\}$. According to the prime number theorem, $p$ divides $x\,y\,|x-y|$ with probability at most

$$\ln(u^3)/\ln(m/2)/(m/\ln(m)/3) < 12\log(u)/m \leq 1/2r^2$$

using $m > 4$. With probability at most $2/m \leq 1/4r^2$, parameter $t$ is zero. Hence, with probability at least $1 - 3\epsilon/4r^2$ we have that $p$ does not divide $x\,y\,|x-y|$ and $t \neq 0$.

Conditioned on this, $x$ and $y$ are distinct nonzero numbers in $\mathbb{Z}_p$, and hence the values $t\,x + s$ and $t\,y + s$ are independent and uniformly distributed in $\mathbb{Z}_p$. This means that the probability that $h(x) = a$ and $h(y) = b$ is at most $\lceil p/r \rceil^2/p^2 \leq 1/r^2 + 2/pr + 1/p^2 \leq (1 + r/p + (r/p)^2)/r^2 < (1 + \epsilon/4)/r^2$. Summing up, the chances of $h(x) = a$ and $h(y) = b$ are at most $(1 + \epsilon)/r^2$, as desired. □

# Chapter 9

## On the cell probe complexity of membership and perfect hashing

In this chapter we consider two fundamental static data structure problems, MEMBERSHIP and PERFECT HASHING, in Yao's cell probe model [Yao81]. In this model, a data structure is a numbered sequence of $s$ "cells", each containing an element of $\{0,1\}^b$, for a positive integer parameter $b$. The *worst case complexity* of a query is the number of cells that a deterministic algorithm needs to probe to answer it, in the worst case over all possible data, for the optimal choice of data structure, in the worst case over all queries. In this chapter we mainly consider worst case cell probe complexity, but we will mention some results on average and expected case complexity. For positive integer $x$ define $[x] = \{1, 2, \ldots, x\}$. We state the problems as follows.

MEMBERSHIP$(u, n, s, b)$:
Given a set $S \subseteq [u]$, $|S| = n \leq u/2$, use a data structure with $s$ cells of $b$ bits to accommodate membership queries for $x \in [u]$ ("Is $x$ in $S$?").

PERFECT HASHING$_{\mathcal{H}}(u, n, r, s, b)$:
Suppose $\mathcal{H}$ is a family of functions from $[u]$ to $[r]$. Given a set $S \subseteq [u]$, $|S| = n \leq r \leq u/2$, store a perfect hash function for $S$, i.e., a function $f \in \mathcal{H}$ that is 1-1 on $S$, in a data structure with $s$ cells of $b$ bits, and accommodate function value queries for $x \in [u]$ ("What is the value of $f$ on $x$?").

Informally, a membership query determines *whether* an element is present in a certain subset of $[u]$, and a perfect hashing query can provide a pointer to *where* associated information is located (by 1-1ness the location of this information is unique to the element). Frequently, one considers the "combined" problem, that is, to design a data structure that on query $x$ answers whether $x \in S$, and if so, returns the value of a perfect hash function pointing to some "satellite information" unique to $x$. This problem will be referred to as DICTIONARY$_{\mathcal{H}}(u, n, r, s, b)$.

We are particularly interested in the cases $b = 1$, where cell probes are referred to as *bit probes*, and $b = \log u$, where cells are referred to as *words*. (For simplicity we assume that $u$ is a power of 2.) We pay special attention to PERFECT HASHING in the case $r = n$, referred to as "minimal perfect hashing".

Very efficient data structures exist for most instances of the two problems. Our interest lies in an exact understanding of how efficient the query algorithms can be. The cell probe model ignores the cost of computation, but as random memory accesses in real hardware are becoming orders of magnitude slower than computational instructions, it focuses on a main practical bottleneck. The precise space usage is of secondary concern, though our data structures use within a constant factor of minimum space for most parameters.

## 9.1 Background

### Membership

The minimum number of bits with which it is possible encode a MEMBERSHIP data structure is $s_{\mathrm{m}}(u,n) = \log \binom{u}{n} = \Theta(n \log(u/n))$. (When the parameters are understood we will simply write $s_{\mathrm{m}}$.) We do not consider data structures of size less than one cell, so a space optimal data structure is one that occupies $O(s_{\mathrm{m}}/b + 1)$ cells.

The *average case* bit probe complexity of MEMBERSHIP was studied by Minsky and Papert in the book *Perceptrons* [MP69, Sect. 12.6], where it is shown that a constant number of bit probes suffice on average over all queries. The study of *randomized* (Monte Carlo) bit probe complexity, where the query algorithm makes coin flips and is allowed some error probability, was initiated recently by Buhrman et al. [BMRV00]. Notably, one can get two-sided error probability $\epsilon$ using *one bit probe* and $O(\frac{n}{\epsilon^2} \log u)$ bits of space (which is $O(s_{\mathrm{m}})$ for constant $\epsilon > 0$ and $u = n^{1+\Omega(1)}$). The scheme is nonexplicit, that is, it is not shown that there are efficient (polynomial time) procedures for constructing the data structure and carrying out queries. Ta-Shma [TS] has given an explicit version of the construction using space $n \cdot 2^{O((\log\log u)^3)}$.

Buhrman et al. were also the first to study directly the worst case bit probe complexity of MEMBERSHIP. They showed a lower bound of $\Omega(\log(u/n))$ bit probes for space $O(s_{\mathrm{m}})$, matching the $O(\log u)$ upper bound of Fredman et al. [FKS84] for $u = n^{1+\Omega(1)}$. More generally, the space required when using $t$ bit probes was shown to be between $(u/n)^{\Omega(1/t)} nt$ and $u^{O(1/t)} n \log u$. The upper bound is nonexplicit; explicit schemes using few bitprobes have been considered by Radhakrishnan et al. in [RRR01], resulting in a scheme using $O(\log\log n)$ bit probes and $O(u \log n / 2^{\log_n u})$ bits of space.

In addition, Buhrman et al. considered the *adaptivity* of (deterministic) membership query algorithms, that is, the way in which probe locations depend on the results of previous probes. Surprisingly, adaptivity does not help in general, as space $O(n \log u)$ and $O(\log u)$ bit probes can be achieved by a *nonadaptive* scheme, i.e., where all probe locations are determined by the query element. Again, this scheme is not explicit; Ta-Shma's explicit construction [TS] uses space $n \cdot 2^{O((\log\log u)^3)}$ and $2^{O((\log\log u)^3)}$ bit probes for nonadaptive queries.

Motivated by applications of storing a set of machine words on real world computers, upper bounds on the RAM (and hence the word probe) complexity of MEMBERSHIP are of considerable interest. Two seminal papers dealing with this subject are among the most cited in the data structures literature: Carter

and Wegman [CW79] gave a randomized data structure of $O(n)$ words, that uses $O(1)$ word probes per operation, expected, even permitting dynamic changes to the set stored. Fredman, Komlós and Szemerédi [FKS84] showed that $O(1)$ word probes suffice in the worst case for data structures of $O(n)$ words. If $u$ is sufficiently larger than $n$, three word probes suffice, the first one to a fixed location. The space usage has since been lowered to $s_{\mathrm{m}}+o(s_{\mathrm{m}})$ bits (see [BM99] and Chapter 5) at the cost of a large constant number of word probes.

## Perfect hashing

A space optimal data structure for Perfect Hashing uses $s_{\mathrm{ph}}(u,n,r) = \Theta(n^2/r + \log n + \log\log_r u)$ bits. (When the parameters are understood we will simply write $s_{\mathrm{ph}}$.) Apart from the trivial $\log n$ lower bound, a proof of this can be found in e.g. [Meh84, III.2.3]. In this chapter we will be concerned mainly with the case $r = O(n)$, for which $s_{\mathrm{ph}} = \Theta(n + \log\log u)$. It is optimal to use $O(s_{\mathrm{ph}}/b + 1)$ cells of memory.

The data structure of Fredman et al. [FKS84] is in fact a hash table along with a solution to Perfect Hashing$_{\mathcal{H}}(u, n, O(n), O(n), \log u)$ requiring (for $u$ large enough) two cell probes, one of which is to a fixed location. An efficient scheme for Perfect Hashing$_{\mathcal{H}}(u, n, n, O(n), O(\log n+\log\log u))$ (i.e., minimal perfect hashing) using two cell probes is presented in Chapter 3, where it was also shown that one cell probe schemes are impossible for $b = o(\frac{n}{1+n^2/u})$. Schmidt and Siegel [SS90b] presented a coding scheme for essentially the perfect hash function in [FKS84], using space $O(s_{\mathrm{ph}})$ for $r = n$. The construction can be extended to give space $O(s_{\mathrm{ph}})$ for $r = O(n^2/\log^2 n)$. A space bound of $s_{\mathrm{ph}} + o(s_{\mathrm{ph}})$, still with constant evaluation time, was obtained for minimal perfect hashing by Hagerup and Tholey in [HT01]. The number of cell probes, as well as the constant hidden in the order notation, is quite high for these space optimal schemes, so they are mainly of theoretical interest.

The bit probe complexity of perfect hashing does not seem to have been studied directly before. An upper bound of $O(\log n + \log\log u)$ bit probes follows directly from the Schmidt-Siegel construction [SS90b].

In a relaxation of perfect hashing considered by Schmidt and Siegel [SS90b], an "oblivious $t$-probe hash function" computes a *set* of $t$ values, one of which (for appropriate arrangement of the set in a table) is the location of the input if it is in the set. It was shown that $n/2^{O(t)}$ bits are needed to represent such a function, in the case $r = n$. However, it is mentioned that there is a probabilistic argument showing $O(\log n + \log\log u)$ bits to suffice when $r = (1+\Omega(1))\, n$, for some constant $t$. Such probabilistic arguments appear explicitly in [ABKU99, CS97], in the context of "balanced allocations". In our context, a result of [CS97] implies that there exists a family of oblivious 4-probe hash functions for hash tables of size $2n$, whose functions can be stored in $O(\log n + \log\log u)$ bits.

## 9.2 This chapter

This chapter contains a collection of results on the cell probe complexity of membership and perfect hashing, as summarized in the remainder of this section.

### Membership

We prove in Section 9.3.1 that the bit probe lower bound of Buhrman et al. [BMRV00] is tight, by giving an explicit MEMBERSHIP scheme that uses $O(\log(u/n))$ bit probes and space $O(s_{\mathrm{m}})$. The construction makes use of a bounded concentrator, which is a weak expander graph.

As for word probe complexity, we show in Section 9.3.2 that if one wants to use $O(n)$ words of space, and if $u$ is not too small in terms of $n$, schemes probing two words essentially require a perfect hash function with linear range to be storable in one word. In particular, such schemes are only possible when $u = 2^{\Omega(n)}$. Thus, there is no hope of improving the word probe count of three achieved by the scheme of Fredman et al. [FKS84].

In Section 9.3.3 we investigate the previously mentioned comment of Schmidt and Siegel [SS90b] in the case $k = 2$, for which the results in [ABKU99, CS97] do not say anything. The result is a scheme that is an improvement of [FKS84] in that it allows for *parallelism* in the word probes: After the first (fixed) probe, the two last probes can be determined and carried out in parallel. Additionally, our scheme works also in the case where $u$ is not much larger than $n$, and uses space $O(s_{\mathrm{m}})$, unless $u$ is close to $n$. The improved parallelism seems interesting from a practical point of view, as CPU pipelining could potentially decrease the time relative to the case of adaptive probes by a factor of nearly two. In case the data structure resides in external memory, split to multiple disks, any item can be retrieved in one parallel I/O, using minimal internal memory. Our scheme is explicit when $u \geq n^{\beta \log n}$, for a constant $\beta$, in which case the first probe can be used to read a function from a (nearly) $O(\log n)$-wise independent family. By increasing the number of fixed word probes to $O(\log n)$, which can be argued to be practical, for example in the case of external memory, we get an explicit scheme in all cases.

### Perfect hashing

We show in Section 9.4.1 that the bit probe complexity of the Schmidt–Siegel scheme is optimal for PERFECT HASHING$_{\mathcal{H}}(u, n, r = n, O(s_{\mathrm{ph}}), 1)$, i.e., the bit probe complexity of minimal perfect hashing, using optimal space, is $\Theta(\log n + \log\log u)$.

In Section 9.4.2 we give an alternative space optimal scheme, conceptually much simpler than that of Schmidt and Siegel, that can be implemented with just one adaptive cell probe reading less than $\log n$ bits. The scheme is non-explicit, but can be made explicit using (nearly) $O(\log n)$-wise independence, increasing the number of bits read in fixed cell probes from $O(\log n + \log\log u)$ to $O(\log^2 n + \log\log u)$.

In Section 9.4.3 we turn the attention to range size $r > n$, showing that *one* adaptive bit probe suffices for $r = O(n)$, using minimal space and $O(\log n + \log\log u)$ fixed bit probes. Again, this scheme is nonexplicit, but can be made explicit at the cost of increasing the number of fixed bit probes by $O(\log^2 n)$. The explicit scheme is much simpler than previous perfect hash functions using minimal memory. We also show that such a scheme is not possible for $r < (\log e - \Omega(1))\, n$, no matter how much space is used. If one wants range $r = n$, $\Omega(\log\log n)$ adaptive bit probes are required in general.

Finally, in Section 9.4.4, we consider query schemes with no adaptivity, and show that for $u = 2^{\Theta(n)}$, a constant fraction of any space minimal $\Theta(n)$-bit data structure must be read to determine the function value. Recall that $O(\log n + \log\log u) = O(\log n)$ bit probes, of which just one is adaptive, is enough. This appears to be the first problem for which an exponential separation between adaptive and nonadaptive query schemes has been shown. Note that no super-exponential separation can exist, as any adaptive $t$ bit probe scheme can be converted to a nonadaptive $2^t - 1$ bit probe scheme. It is interesting to compare this to the fact that MEMBERSHIP has efficient nonadaptive query schemes.

### Notation

For convenience, we introduce a notation for expressing the adaptivity of a query algorithm. An $(a_0, a_1, \ldots, a_m)$-*probe* query scheme is one that:

1. Starts by performing $a_0$ fixed cell probes, not depending on the input (this is the 0th round).
2. In the $i$th round, for $i = 1, \ldots, m$, makes $a_i$ probes to cells determined by the input and by the outcomes of probes in previous rounds.

## 9.3 Membership

### 9.3.1 Tight bit probe bound

In this section we present an explicit MEMBERSHIP scheme that has optimal space and bit probe complexity. It can be thought of as a "generalization" of bit vectors that is space efficient even for sparse sets. In fact we give a *trade-off* between the number of bitprobes and the space usage. The data structure also allows PERFECT HASHING$_{\mathcal{H}}(u, n, r, O(s_{\mathrm{m}}), 1)$ queries in $O(\log(u/r))$ cell probes, for $r/n$ larger than some constant, i.e., it solves the DICTIONARY problem.

**Theorem 9.1** *There exists a constant $\beta$ such that for $r > \beta n$ there is a function family $\mathcal{H}$ with an explicit* DICTIONARY$_{\mathcal{H}}(u, n, r, O(n\log(u/r) + r), 1)$ *scheme having cell probe complexity $O(\log(u/r))$.*

*Proof.* Our scheme is recursive. It uses a data structure of $O(n)$ bits that specifies a superset of $S$ and a 1-1 mapping from this superset to $[u']$, where $u'$ is a constant factor smaller than $u$. A constant number of bit probes suffice to determine if the query element is not in the superset, in which case we can

immediately return a valid answer. If the query element is in the superset, its mapping can be computed by probing $O(1)$ bits. We thus either terminate or get a new DICTIONARY problem for a set $S' \subseteq [u']$ (using that each element of $[u']$ corresponds to exactly one element in $[u]$).

The recursion ends after $O(\log(u/r))$ steps when the set is contained in $[r]$, and a perfect hash function value is simply the identity function. A bit vector of length $r$ can be used to determine membership in $S$.

The essential ingredient in our solution is a family of *bounded concentrators*, which are constant degree bipartite graphs with $x$ vertices on the left, at most $\theta x$ vertices on the right, for constant $\theta < 1$, having the property that any set $P$ of up to $x/2$ left hand vertices can be matched to $|P|$ vertices on the right hand side. Simple, explicit constructions of bounded concentrators exist [GG81]. We denote left hand vertices by $p_1, \ldots, p_x$ and right hand vertices by $q_1, \ldots, q_y$, where $y \leq \theta v$, and assume some fixed ordering of the neighbors of each vertex.

If $u \leq \frac{4n}{1-\theta}$ the recursion is stopped. Otherwise we construct a data structure as follows: Split $[u]$ into $x = 2n$ parts $U_1, \ldots, U_x$ of size at most $\lceil u/2n \rceil$, and consider the set $P = \{p_i \mid S \cap U_i \neq \emptyset\}$ which has size at most $n = x/2$. By definition of a bounded concentrator, the vertices of $P$ can be matched to a set $Q \subseteq \{q_1, \ldots, q_y\}$. Suppose that $p_i \in P$ is matched to its $k$th neighbor. Then we write the $O(1)$ bit number $k$ as entry $i$ of a $2n$-element table $T$. Table entries of vertices not in $P$ are set to a special value.

An element $\mu \in U_i$ is in the abovementioned superset if and only if $p_i \in P$. In this case we map $\mu$ to an element in $[u']$, where $u' = y\lceil u/2n \rceil$, as follows: If $p_i$ is matched to $q_j$, map $\mu$ to $f(x) = j\lceil u/2n \rceil - \mathrm{rank}_{U_i}(x)$, where $\mathrm{rank}_{U_i}(x)$ is the number of $x$ in some fixed numbering $0, \ldots, \lceil u/2n \rceil - 1$ of $U_i$. By the matching property, no other element of $[u]$ maps to $f(x)$. Note that $u'/u \leq \frac{1+\theta}{2} = 1 - \Omega(1)$, as desired. □

**Corollary 9.1** *The cell probe complexity of* MEMBERSHIP$(u, n, O(s_{\mathrm{m}}), 1)$ *is* $\Theta(\log(u/n))$.

*Proof.* The upper bound follows by Theorem 9.1 by setting $r = O(n)$. The lower bound was shown in [BMRV00]. □

It can be noted that a constant fraction of all possible queries are resolved at each step. Thus, $O(1)$ probes suffice on average over all queries, so our scheme is optimal also with respect to average case complexity. If we consider just queries for elements in $S$, the lower bound of [BMRV00] extends to show that $\Omega(\log(u/n))$ bit probes are needed on average.

### 9.3.2 Two word probes do not suffice

Yao [Yao81] studied MEMBERSHIP in a model where the query algorithm first looks up $b = \log u$ fixed bits, and then, in a table of length $s$, probes a single word containing some element of $S$, answering "yes" if it is equal to the input. He observed that for this problem to be solvable there has to be a size $u$ perfect family of hash functions with range $[s]$. Hence, such two-probe schemes are

possible only if $u \geq 2^{s_{\mathrm{ph}}(u,n,s)}$. In the cell probe model there is no restriction on the contents of the second word probed, or on how it is interpreted. However, for the case where $u$ is not too close to $n$ or $s$, we show that again, such a scheme is not possible unless $u = 2^{\Omega(s_{\mathrm{ph}}(u,n,s))}$. The lower bound carries over to the case where the first word probe is not necessarily fixed, meaning that the MEMBERSHIP scheme of Fredman et al. [FKS84], that probes three words, is in general word probe optimal among schemes using $O(n)$ words.

**Theorem 9.2** *For any constants $\epsilon > 0$ and $k \in \mathbf{N}$, if $u > s^{1+\epsilon}2^{\epsilon b}$, $s \geq n$, and* MEMBERSHIP$(u, n, s, b)$ *has a $(k, 1)$-probe query scheme, then $b = \Omega(s_{\mathrm{ph}}(u, n, s))$.*

*Proof.* We employ a "volume bound" similar to the lower bound for the program size of perfect hash functions in [Meh84]. More specifically, we bound the number of sets that be accommodated for each of the $2^{kb}$ possible bit patterns read in the $k$ fixed probes. This turns out to be a very small fraction of all $n$-subsets, giving a lower bound on $kb$.

Without loss of generality we assume that $s \geq 2n$. Fix a $kb$-bit pattern, and let $S_1, \ldots, S_l \subseteq [u]$ be the $n$-subsets for which this bit pattern is used. For each $i \in [s]$, let $U_i \subseteq [u]$ be the set of elements for which the query algorithm uses the adaptive probe to probe cell $i$. Since a cell can contain $2^b$ different bit patterns, the set $\{U_i \cap S_j \mid j \in [l]\}$ can have size at most $2^b$. We will use this to get an upper bound on $l$.

For a positive integer $m$ to be determined later, let $F$ be the family of functions $f : [s] \to \{0, \ldots, m\}$ for which $\sigma_f \stackrel{\mathrm{def}}{=} \sum_{i=1}^{s} f(i) \leq n$. The following fact is easily shown by induction on $m$.

**Fact 9.1** *There are at most $\binom{s}{n} n^m$ functions in $F$.*

For any $S_j$, $j \in [l]$, there is a function in $F$ for which $f(i) = |S_j \cap U_i|$ if $|S_j \cap U_i| \leq m$ and $f(i) = 0$ otherwise. We bound the number of sets by summing over all functions in $F$, and the number $x$ of indices $i \in [s]$ for which $|S_j \cap U_i| > m$. Given $x$ and $f \in F$ there are $\binom{s}{x}$ possible ways of choosing these indices. At most $2^{bx}$ possible sets can be specified by the contents of the corresponding cells, and no more than $\prod_{i=1}^{s} |U_i|^{f(i)}$ different sets can be specified by the remaining cells. Thus we have

$$\begin{aligned} l &< \sum_{0 \leq x < \frac{n}{m}} \sum_{\substack{f \in F \\ \sigma_f \leq n - mx}} \binom{s}{x} 2^{bx} \prod_{i=1}^{s} |U_i|^{f(i)} \\ &< \sum_{0 \leq x < \frac{n}{m}} s^x \binom{s}{n} n^m \, 2^{bx} \left(\frac{u}{s}\right)^{n-mx} \\ &< n^m \binom{s}{n} \left(\frac{u}{s}\right)^n \sum_{x \geq 0} \left(\frac{s^{m+1} 2^b}{u^m}\right)^x \quad . \end{aligned}$$

The second inequality uses convexity, i.e., that the sum is maximized when the $|U_i|$ are equal. For suitable $m = O(1/\epsilon)$ the last sum is bounded by 2. In conclusion, we must have $kb > \log \binom{u}{n} - \log \left(2n^m \binom{s}{n} \left(\frac{u}{s}\right)^n\right) = \Omega(n^2/s - m \log n)$,

where the last bound is derived as in [Meh84]. By the bit probe lower bound of [BMRV00], and since $u > n^{1+\epsilon}$, we must have $kb = \Omega(\log u)$. Together, these bounds give the desired bound $b = \Omega(n^2/s + \log\log_s u)$. □

**Corollary 9.2** *For any constant $\delta > 0$, if $u > s^{2+\delta}$, $s \geq n$, and* MEMBERSHIP$(u, n, s, \log u)$ *has a $(0, 1, 1)$-probe query scheme, then $u = 2^{\Omega(s_{\mathrm{ph}}(u,n,s))}$.*

*Proof.* For some set of queries $U \subset [u]$ of size at least $u/2s > s^{1+\delta}/2$, and size a power of two, the query algorithm probes the same first word. We can apply Theorem 9.2 with $\epsilon = \Omega(\delta)$ to this set. Finally note that $s_{\mathrm{ph}}(|U|, n, s) = \Theta(s_{\mathrm{ph}}(u, n, s))$. □

### 9.3.3 Two parallel adaptive word probes suffice

In this section we show the existence of a MEMBERSHIP scheme whose query algorithm on input $x$ probes one fixed word, containing a description of two functions $f_0$ and $f_1$, and then probes indices $f_0(x)$ and $f_1(x)$ independently in two linear size tables. The set contains $x$ if and only if it is found in one of the two adaptively probed words. The locations of elements also define a perfect hash function, so we have a scheme for DICTIONARY. Our result is summarized in the below theorem.

**Theorem 9.3** *There is a function family $\mathcal{H}$ such that* DICTIONARY$_{\mathcal{H}}(u, n, r = O(n), O((s_{\mathrm{m}} + n\log\log n)/\log u), \log u)$ *has a $(1, 2)$-probe query scheme. The scheme can be made explicit for $u \geq n^{\beta\log n}$, where $\beta$ is a constant.*

Note that the $n\log\log n$ term is only significant for $u \leq n(\log n)^{o(1)}$. Our hash functions will be taken from families that are "nearly $k$-wise independent" in the sense of $(c, k)$-universality.

**Definition 9.1** *A family $\{\phi_i\}_{i\in I}$ of functions $\phi_i : [u] \to [r]$ is* $(c, k)$-universal *if, for any $k$ distinct elements $x_1, \ldots, x_k \in [u]$, any $y_1, \ldots, y_k \in [r]$, and $i \in I$ chosen uniformly at random, $\Pr[\phi_i(x_1) = y_1, \ldots, \phi_i(x_k) = y_k] \leq c/r^k$.*

#### Analysis

Let $c$ be any constant, and let $f_0, f_1 : [u] \to [r]$ be functions chosen independently at random from a $(c, k)$-universal family. (Constructions of such families can be found, for example, in [Sie89].) We claim that for any $\epsilon$, $0 < \epsilon < 1$, range $r \geq (1 + \epsilon)n$ and $k > 2\log(r)/\epsilon$ suffice to make it possible, with high probability, to arrange the elements of $S$ in two tables such that for all $x \in S$, $x$ resides in either cell $f_0(x)$ of table number one, or cell $f_1(x)$ of table number two.

Suppose that no arrangement is possible. By Hall's theorem [Hal35] this means that there is a subset $S' \subseteq S$ such that $|f_0[S']| + |f_1[S']| < |S'|$. Assume $S'$ to be a set of minimum size for which the inequality is satisfied, and consider the bipartite graph with left vertices labelled by $[r]$, right vertices labelled by $[r]$,

and edge multiset $\{(f_0(x), f_1(x)) \mid x \in S'\}$. As the number of edges is greater than the number of non-isolated vertices, $|f_0[S]| + |f_1[S]|$, there are at least two vertices of degree 3 or one of degree at least 4, and by minimality of $S'$ there can be no vertex of degree one. Thus, starting and ending in a vertex of degree more than two, we can form a path of $w \geq 3$ edges through at most $w - 1$ vertices. We call such a path a *$w$-witness*.

Even in the graph with edge multiset $\{(f_0(x), f_1(x)) \mid x \in S\}$, a $w$-witness is unlikely to exist for any $w$. We first consider the case $w \leq k$. There at at most $w^2 r^{w-1}$ possible $w$-witnesses. By $(c, k)$-universality, any configuration of $w \leq k$ edges has probability at most $c^2/r^{2w}$. Thus, the expected number of such $w$-witnesses is at most

$$\sum_{w=3}^{k} w^2 r^{w-1} n^w c^2 r^{-2w} = \frac{c^2}{r} \sum_{w=3}^{k} w^2 (n/r)^w < \frac{13\, c^2}{\epsilon r} \ .$$

For $w > k > 2\log(r)/\epsilon$ we consider a subgraph of a $w$-witness, namely a path of $k$ edges through at most $k+1$ vertices. Again, such a graph is not very likely to occur in $\{(f_0(x), f_1(x)) \mid x \in S\}$. The expected number of such paths is bounded by

$$r^{k+1} n^k c^2 r^{-2k} = c^2 r (n/r)^k < c^2 r (1+\epsilon)^{-2\log(r)/\epsilon} < c^2/r \ .$$

The above shows that for $r$ larger than some constant (dependent on $c$ and $\epsilon$), the probability that a randomly chosen pair of functions from a $(c, k)$-universal family does not work for a particular set is smaller than $1/3$.

To get a family of pairs of functions where a pair can be described in one word, we use a probabilistic construction. Consider a family of $u$ independently and randomly chosen function pairs from a $(c, k)$-universal family. For any set, the probability that none of these pairs works is less than $3^{-u}$. Hence, with overwhelming probability there is a good pair for any set. Enumerating the pairs of any good family, the appropriate pair can be described in one word. (In fact, a family of size $\log \binom{u}{n} \leq n \log u$, and hence $\log n + \log\log u$ fixed bits, would suffice.) For constant $n$, one can easily get a $(1, 1)$-probe query scheme, using universal hashing [CW79] to a table of size $O(n^2)$.

An explicit construction where the function pair can be described using $O(\log^2 n + \log\log u)$ bits is arrived at as follows. First use a function from a $(O(1), 2)$-universal family the map $S$ injectively to a set $S' \subseteq [u']$, where $u' = O(n^2)$. A function from such a family can be described in $O(\log n + \log\log u)$ bits. Then use a $(c, k)$-universal family from $[u']$ to $[r]$ as described above. Two functions from such a family can be described in $O(k \log u') = O(\log^2 n)$ bits. When $u \geq n^{\beta \log n}$ for a suitable constant $\beta$, all function descriptions fit in one word of $\log u$ bits.

If one increases space to $n^{1+\epsilon}$ words, the analysis can be performed using $(O(1), O(1))$-universal hash functions, giving explicit constant time schemes on the RAM.

**Bibliographical remark:** The above argument shares some features with an analysis of Karp et al. [KLMadH96] that looks at the same random graph,

with slightly different parameters, in connection with PRAM simulation. It is different from those in [ABKU99, CS97], which do not seem to go through with less than $n$-wise independence.

### Reducing space

As stated, the scheme uses $O(n \log u)$ rather than $O(s_{\mathrm{m}} + O(n \log \log n))$ bits of space. This means that we have to asymptotically reduce the space usage when $u \leq n(\log n)^{o(1)}$. The tool used for this is called *quotienting* (see Chapter 5 and [Knu98, Exercise 6.4.13]): Rather than explicitly storing elements of $[u]$ in the hash tables, the element in cell $i$ is represented relative to the subset of $[u]$ hashing to $i$. If the hash functions used have the property of mapping $[u]$ evenly to $[r]$, this saves an optimal $\log r - O(1)$ bits per hash table entry. To benefit from the decrease in the number of bits needed, one packs the largest possible number of table elements into each cell.

Known constructions of $(c, k)$-universal families map $O(ku/r)$ elements to each value in $[r]$, giving a savings of $\log(r/k) - O(1)$ bits per cell. In particular, for $k = O(\log n)$ this yields a space usage of $O(n \log \log n)$ bits more than the information theoretical minimum.

### Construction algorithm

We have not described how to compute the positions of set elements in the tables. As it is a matching problem, it can clearly be done in polynomial time. However, there are only two possibilities for each element, so the problem can be solved in expected linear time by a simple reduction to 2-SAT.

We create one variable $v_x$ for each element $x \in S$, and clauses stating that there is no collision for each pair of elements with the same value under $f_0$ or $g$:

$$v_x \vee v_y \text{ for all } x, y \in S \text{ where } x \neq y \text{ and } f_0(x) = f_0(y)$$

$$\bar{v}_x \vee \bar{v}_y \text{ for all } x, y \in S \text{ where } x \neq y \text{ and } f_1(x) = f_1(y)$$

If a $(O(1), 2)$-universal family is used (as above), the expected number of clauses is $O(n)$. A linear time 2-SAT algorithm is outlined in [EIS76].

### Related hashing schemes

It is interesting to compare our scheme to *open addressing* hashing schemes. Such schemes evaluate a fixed sequence of hash functions to determine which cells to probe. Dynamic insertions are done greedily, i.e., by inserting the element in the first available cell probed. It was shown by Yao [Yao85] that for such schemes, under the assumption of truly random hash functions, the expected average number of probes to perform a lookup of an element in the table is at least $\frac{1}{\alpha} \ln \frac{1}{1-\alpha} - o(1)$, where $\alpha$ denotes the ratio between the number of elements and the table size. In our scheme $\alpha \approx \frac{1}{2}$, so for large enough $n$ and small $\epsilon$, the expected number of probes is at least $2/\ln(2) - \epsilon \geq 1.38$. By randomly deciding which table to look in first, we can get an expected probe

count of 1.5, and at the same time guarantee that no more than two probes are needed, as opposed to the $\Omega(\log n)$ expected worst case for open addressing.

Our scheme is related to a dynamic hashing strategy called *2-way chaining* [ABKU99]. It is a variant of chained hashing in which two hash functions are used, call them $f_0$ and $f_1$. Insertion of an element $x$ is performed in the shortest of chain number $f_0(x)$ and chain number $f_1(x)$. Under the assumption of totally random hash functions it was shown in [ABKU99] that the expected maximal chain length for this scheme is $O(\log \log n)$, with a low constant. This can also be shown to be true for $O(\log n)$-wise independent families.

## 9.4 Perfect hashing

### 9.4.1 Tight bit probe bound for minimal range

A natural question is whether Theorem 9.1 can be extended to DICTIONARY$_{\mathcal{H}}(u, n, r = n, O(s_{\mathrm{m}}), 1)$, i.e., to supply a *minimal* perfect hash function. We answer this question negatively by showing that, no matter how much space is used, $\lfloor \log n \rfloor$ bits must be probed. In fact, we show that the bit probe complexity achieved in [SS90b] is optimal.

**Theorem 9.4** *There exists a function family $\mathcal{H}$ such that the cell probe complexity of* PERFECT HASHING$_{\mathcal{H}}(u, n, r = n, s = O(s_{\mathrm{ph}}), 1)$ *is* $O(\log n + \log \log u)$. *Conversely, for $n \geq 2$ and every function family $\mathcal{H}$ the cell probe complexity of* PERFECT HASHING$_{\mathcal{H}}(u, n, r = n, s = O(s_{\mathrm{ph}}), 1)$ *is* $\Omega(\log n + \log \log u)$.

*Proof.* As mentioned, the upper bound was shown in [SS90b]. As for the lower bound, we first show that the cell probe complexity is at least $\lfloor \log n \rfloor$. Suppose to the contrary that for some function family $\mathcal{H}$, PERFECT HASHING$_{\mathcal{H}}(u, n, r, s, 1)$ has cell probe complexity $t \leq \log(n) - 1$. Each element $x \in [u]$ can map to at most $2^t$ different values in $[r]$, in total for all functions in $\mathcal{H}$. Furthermore, there can be no set of $n$ elements that map only to a set of $n - 1$ values in $[r]$. As each set of $2^t$ values lies within $\binom{r-2^t}{n-1-2^t}$ sets of size $n - 1$, the size of the universe must be less than $n \binom{r}{n-1} / \binom{r-2^t}{n-1-2^t} = n\, r/(n - 2^t) \leq 2r$. But this can not be the case by the problem definition.

Now we turn to a lower bound in terms of $u$. For each $x \in [u]$ there is a decision tree specifying how to perform the sequence of probes and which hash value to return in every case. The number of possible decision trees is bounded by $(sn)^{2^t}$, and there cannot be two elements with the same decision tree, as these would hash to the same value for every function in $\mathcal{H}$. Hence, $u \leq (sn)^{2^t} = n^{2^{t+1}}$ and thus $t \geq \log \log_n(u) - 1$. As we need consider only the case $u > 2^n$, this yields the desired bound. □

### 9.4.2 Minimal range using one adaptive cell probe

A simple nonexplicit scheme, seemingly not described in the literature, achieves optimal space and bit probe performance, looking only at bits from one fixed and one adaptively determined word. First note that sets of constant size can

be handled using universal hashing to a quadratic size table containing function values. For $n$ larger than a suitable constant we can use the following properties of random functions:

- A random function $\rho : [u] \to [2n^2]$ is $1-1$ on $S$ with probability more than $3/4$.
- For some $r = O(n/\log n)$, the following holds for a random function $h : [2n^2] \to [r]$.
  - $h$ maps no more than $\frac{1}{6}\log n$ elements of $\rho[S]$ to each value in $[r]$, with probability at least $7/8$.
  - The number of elements of $\rho[S]$ mapping to $[i]$ is within $n^{2/3}$ of the expectation $in/r$ for all $i \in [r]$ with probability at least $7/8$.

Picking $\rho$ from an $(O(1), 2)$-universal family, and $h$ from an $(O(1), O(\log n))$-universal family in fact also achieves the above, by results in [FKS84] and [SSS93, Theorem 2.5].

In summary, a randomly chosen pair of functions $(h, \rho)$ has all the above properties for fixed $S$ with probability more that $1/2$, As in Section 9.3.3 we can now argue that there exists a family of $\log \binom{u}{n}$ pairs of functions so that for any set $S$ of size $n$ there is a pair with the properties. Our query algorithm can thus read the description of such a pair $(h, \rho)$ using one word probe. For $i \in [r]$, let $B_i = \{\rho(x) \mid x \in S,\, h(\rho(s)) = i\}$. In the second probe, the query algorithm reads cell $j = h(\rho(x))$ of a table that contains:

- The deviation of $\sum_{i<j} |B_i|$ from $\lfloor in/r \rfloor$, using $\lceil \frac{2}{3}\log n \rceil$ bits.
- A minimal perfect hash function $h_j : [2n^2] \to [|B_j|]$ for $B_j$, using at most $\lfloor \frac{1}{3}\log n \rfloor$ bits. (This can be done for large enough $n$ by Mehlhorn's bound [Meh84, III.2.3], see [HT01] for an explicit function.)

Clearly this information fits one word and suffices to compute a minimal perfect hash function values.

If $\log u$ is much larger than $\log n$, the space usage of the scheme as described is not very good. Again, by packing the largest possible number of table entries into each word, we can make sure that at least half of the bits in each cell are utilized. With this modification a space optimal scheme is obtained.

We can get an explicit construction in the case $u \geq n^{\beta \log n}$, where $\beta$ is a suitable constant, using exactly the same technique as for the explicit construction in Section 9.3.3.

**Theorem 9.5** *There is a function family $\mathcal{H}$ such that* Perfect Hashing$_{\mathcal{H}}(u, n, r = n, O(s_{\text{ph}}/\log u + 1), \log u)$ *has a $(1,1)$-probe query scheme. The scheme can be made explicit for $u \geq n^{\beta \log n}$, where $\beta$ is a constant.*

As is easily seen, only $O(\log n + \log\log u)$ bits of the two words probed by the nonexplicit scheme are looked at. In this sense the nonexplicit scheme is also bit probe optimal.

### 9.4.3 Linear range using one adaptive bit probe

Theorem 9.3 shows that it is possible to probe just $O(\log n + \log\log u)$ fixed bits and get a choice of two cells in one of which the query element must be, if present in $S$. We now describe a strengthening of this result, namely that it is possible to look up a *single bit* in a table of size $O(n)$ telling which of the two choices is the right one. This defines a perfect hash function. The range of the perfect hash function we achieve is linear, but the constant needed in the present analysis is large.

Again, we choose two functions $f_0, f_1 : [u] \to [r]$ independently at random from a $(c,k)$-universal family, where $c \geq 1$ is any constant, and $k$ is specified later. Additionally we choose a random function $g : [u] \to [s]$ from a $(c,k)$-universal family, where parameter $s$ is essentially the space to be used. For a bit string $a = a_1 \ldots a_s$ we consider the function

$$h_a : x \mapsto f_{a_{g(x)}}(x) \ . \tag{9.1}$$

We will show that, for $r \geq 2^8 c^3\, n$, $s = n$, and $k = O(\log n)$, with probability $\Omega(1)$ there exists a string $a$ for which $h_a$ is 1-1 on $S$. Our proof is similar to that of Theorem 9.3, but is more complicated. It uses a characterization of satisfiable 2-SAT instances rather than Hall's theorem.

The requirements on $a$ for $h_a$ to be 1-1 on $S$ can be expressed as an instance of 2-SAT. Let $a_i^0$ denote the negation of $a_i$, and let $a_i^1$ be synonymous with $a_i$. The 2-SAT instance can be expressed by the following set of implications:

$$\{a^z_{g(x)} \to a^{z'}_{g(y)} \mid z, z' \in \{0,1\},\ x, y \in S,\ x \neq y,\ f_z(x) = f_{1-z'}(y)\} \tag{9.2}$$

A well known characterization of satisfiable 2-SAT instances states that all implications can be satisfied if and only if there is no sequence of implications of the form $a_i^1 \to \cdots \to a_i^0 \to \cdots \to a_i^1$. So if all implications cannot be satisfied, there is some shortest sequence of implications witnessing this:

$$a^{z_1}_{i_1} \to a^{z_2}_{i_2} \to \cdots \to a^{z_w}_{i_w} \to a^{z_1}_{i_1} \quad \text{where } z_j \in \{0,1\} \tag{9.3}$$

where for some $l$, $1 < l \leq w$, $i_l = i_1$ and $z_l = 1 - z_1$. By minimality, variables can occur only twice in the sequence, once negated and once unnegated, except $a_{i_1}$ which occurs three times.

If the sequence (9.3) exists, there are pairs of distinct elements

$$(x_1, y_2), (x_2, y_3), \ldots, (x_w, y_1) \in S \times S$$

such that the following set of equations hold. (Here and in the following, indices of variables are to be interpreted "modulo $w$", e.g., $y_{w+1}$ is the same as $y_1$.)

$$g(x_j) = g(y_j) \quad \text{for } j = 1, \ldots, w \tag{9.4}$$

$$f_{z_j}(x_j) = f_{1-z_{j+1}}(y_{j+1}) \quad \text{for } j = 1, \ldots, w \ . \tag{9.5}$$

An element $x \in S$ occurs only in equations of (9.5) corresponding to implications involving $a_{g(x)}$. For each occurrence of $a_{g(x)}$ in (9.3) there is one equation

involving $f_0(x)$ and one equation involving $f_1(x)$. Thus, there are at most two occurrences of each of $f_0(x)$ and $f_1(x)$ in (9.5). Also, an element $x \in S$ can occur at most twice in each of the sequences $x_1, \ldots, x_w$ and $y_1, \ldots, y_w$.

We now proceed to show that there is a good probability that the equations (9.4) and (9.5) cannot hold. We will refer to a numbering of the equations in (9.4) and (9.5), namely the one given by $j$.

**The case $w \leq k$.** Consider particular sequences of elements $x_1, \ldots, x_w \in S$ and $y_1, \ldots, y_w \in S$, and a particular sequence $z_1, \ldots, z_w \in \{0, 1\}$. There is a set $D \subseteq [w]$ of the equations (9.5) that necessarily hold if all previous equations in the numbering hold, no matter how $f_0$ and $f_1$ are chosen. Equations in $D$ must necessarily involve two function values that occurred in an earlier equation. Let $d = |D|$. Recall that the function values $f_0(x)$ and $f_1(x)$, $x \in S$, can each occur in the equations at most twice, so we must have $d \leq w/2$. By $(c, k)$-universality the probability of $f_0$ and $f_1$ satisfying the equations (9.5) is at most $c^2 r^{-w+d}$.

Let $D'$ be the set of equations in (9.4) that necessarily hold if all previous equations hold, and let $d' = |D'|$. The probability that equations (9.4) hold is, by $(c, k)$-universality, at most $cs^{-w+d'}$.

The next step is to bound the number of possible ways of choosing the sequences $x_1, \ldots, x_w$ and $y_1, \ldots, y_w$, for given $z_1, \ldots, z_w$, $D$ and $D'$. The main observation is that two certain subsequences of $x_1, \ldots, x_w$ and $y_1, \ldots, y_w$, with $w - d$ and $w - d'$ elements, respectively, suffice to determine *all* elements in the sequences. We argue by induction on $j$ that there are subsequences of $j - |D \cap [j]|$ and $j - |D' \cap [j]|$ elements from $x_1, \ldots, x_j$ and $y_1, \ldots, y_j$ that suffice to determine all elements in these prefixes. This is trivial for $j = 0$. Otherwise, by the induction hypothesis the prefixes $x_1, \ldots, x_{j-1}$ and $y_1, \ldots, y_{j-1}$ are determined by subsequences of $j - 1 - |D \cap [j-1]|$ and $j - 1 - |D' \cap [j-1]|$ elements. If $j \in D$ the element $y_j$ is the unique element in $\{x_1, \ldots, x_{j-1}, y_1, \ldots, y_{j-1}\}$ that necessarily has the value $f_{z_{j-1}}(x_{j-1})$ under $f_{1-z_j}$ (using the fact that each function value occurs in the equations at most twice). Similarly, if $j \in D'$ $y_j$ is the unique element in $\{x_1, \ldots, x_{j-1}, y_1, \ldots, y_{j-1}\}$ that must have the value $g(x_j)$ under $g$ (using that each element occurs at most twice in the sequences $x_1, \ldots, x_j$ and $y_1, \ldots, y_j$). This completes the induction step.

In summary, for any choice of $z_1, \ldots, z_w$, $D$ and $D'$ there are at most $n^{2w-d-d'}$ possible sequences of elements. This means that the total probability that one of these sequences occurs is at most

$$n^{2w-d-d'} \, c^2 r^{-w+d} \, cs^{-w+d'} = c^3 (n^2/rs)^w (r/n)^d (s/n)^{d'} \leq c^3 (2^8 c^3)^{-w/2} \qquad (9.6)$$

using that $s = n$, $r \geq 2^8 c^3 \, n$ and $d \leq w/2$.

There are no more than $2^{3w}$ ways of choosing $z_1, \ldots, z_w$, $D$ and $D'$. Thus, the probability that equations (9.4) and (9.5) hold for some choice of $w \leq k$, $x_1, \ldots, x_w$, $y_1, \ldots, y_w$, $z_1, \ldots, z_w$, $D$ and $D'$ is bounded by

$$\sum_{w \geq 2} 2^{3w} c^3 (2^8 c^3)^{-w/2} \leq \sum_{w \geq 2} 2^{-w} = 1/2 \ .$$

**The case $w > k$.** For large $w$ we bound the probability that $k$ successive implications in (9.3) exist. This would imply that equations $2, \ldots, k$ in (9.4) and equations $1, \ldots, k$ in (9.5) hold for some sequences $x_1, \ldots, x_k \in S$, $y_2, \ldots, y_{k+1} \in S$, and $z_1, \ldots, z_{k+1} \in \{0,1\}$.

As above, let $d \leq k/2$ be the number of equations $D$ in (9.5) that hold if the previous ones do, and let $d'$ be the number of equations $D'$ in (9.4) that hold if the previous ones do. Similar to before, the probability of satisfying the equations (9.5) is bounded by $c^2 r^{-k+d}$, and the equations (9.4) hold with probability at most $cs^{-k+1+d'}$.

Given $z_1, \ldots, z_{k+1}$, $D$, and $D'$, there are $n^{2k-d-d'}$ possible sequences of elements $x_1, \ldots, x_k$ and $y_2, \ldots, y_{k+1}$, by arguments like those above. The sets $D$ and $D'$, and the sequence $z_1, \ldots, z_{k+1}$ can be chosen in at most $2^{3k}$ ways. Hence, the probability of $k \geq \log(c^3 n) + 2$ successive implications is bounded by

$$2^{3k}\, c^2 r^{-k+d}\, cs^{-k+1+d'} n^{2k-d-d'} = c^3 n\, (8n/r)^k (r/n)^d \leq c^3 n\, (8\sqrt{n/r})^k \leq 1/4,$$

using $s = n$, $d \leq k/2$, and $r \geq 2^8 n$.

To sum up, randomly chosen hash functions yield a perfect hash function with probability at least $1/4$. The theorem now follows by arguments like those concluding the proof of Theorem 9.3.

**Theorem 9.6** *There is a function family $\mathcal{H}$ such that* Perfect Hashing$_{\mathcal{H}}(u, n, r = O(n), O(s_{\text{ph}}), 1)$ *has a $(O(\log n + \log\log u), 1)$-probe query scheme. The scheme can be made explicit for $u \geq n^{\beta \log n}$, where $\beta$ is a constant.*

### Lower bounds

We conclude this section by showing that there is no hope of improving the range of the above perfect hash function to (nearly) minimum: One needs either $r > (\log(e) - o(1))\, n$ or $\Omega(s_{\text{ph}})$ fixed bit probes, regardless of the size of the data structure.

**Lemma 9.1** *For any function family $\mathcal{H}$, if there is a $(k, 1)$-probe query scheme for* Perfect Hashing$_{\mathcal{H}}(u, n, r, s, 1)$ *then $k \geq s_{\text{ph}} - (n + \log n + \log\log u + O(1))$.*

*Proof.* For every set $S \subseteq [u]$ of size $n$, there must be at least one data structure that encodes a perfect hash function for $S$. The set of cells probed by the query algorithm on inputs in $S$ has size at most $k + n$. Since bits outside these cells can be set arbitrarily, a fraction $2^{-(k+n)}$ of $\{0,1\}^s$ can encode a perfect hash function for $S$. Hence, there is a function $f$ that is perfect for a fraction $2^{-(k+n)}$ of all $n$-subsets of $[u]$. Let $\sigma$ be a random permutation on $[u]$. Then for every set $S \subseteq [u]$ of size $n$, the function $\sigma \circ f$ is perfect for $S$ with probability $2^{-(k+n)}$. The probabilistic method now yields that there exists a perfect hash function family with $O(2^{k+n} n \log u)$ functions. Therefore $\log(2^{k+n} n \log u) + O(1) \geq s_{\text{ph}}$, implying the lemma. □

**Corollary 9.3** *For any function family $\mathcal{H}$, if $u = n^{2+\Omega(1)}$, $\log\log u = o(n)$, $r \leq (\log(e) - \Omega(1))\, n$ and* Perfect Hashing$_{\mathcal{H}}(u, n, r, s, 1)$ *has a $(k, 1)$-probe query scheme then $k = \Omega(n)$.*

*Proof.* For $u = n^{2+\Omega(1)}$ we have that $s_{\text{ph}} \geq \log(e)n^2/r - O(\log r)$, see for example [FK84]. Then by Lemma 9.1, $k \geq \log(e)n^2/r - (n + \log\log u + O(\log r)) = \Omega(n)$. □

While the above lower bound depends heavily on the fact that only one bit is probed adaptively, slightly increasing the number of adaptive probes does not help with respect to implementing minimal perfect hashing.

**Theorem 9.7** *For any function family $\mathcal{H}$, if* Perfect Hashing$_{\mathcal{H}}(u, n, r = n, s, 1)$ *has a $(k, t)$-probe query scheme then $k \geq n/2^{2^{O(t)}}$.*

*Proof.* Any adaptive $(k, t)$-probe query scheme can be transformed to a nonadaptive $(k, 2^t - 1)$-probe query scheme. A lower bound due to Schmidt and Siegel [SS90b, Theorem 1] then says that $k \geq n/2^{O(2^t)}$. □

### 9.4.4 Adaptivity yields exponential speedup

All good upper bounds for Perfect Hashing have used adaptive cell probes. We show that this is no coincidence, by exhibiting the largest possible gap between adaptive and nonadaptive schemes.

**Proposition 9.1** *There is a constant $\beta$ such that for $s = O(s_{\text{ph}})$ and $u > \beta^s$, there is no function family $\mathcal{H}$ such that* Perfect Hashing$_{\mathcal{H}}(u, n, r, s, 1)$ *has a $(0, o(s))$-probe scheme.*

*Proof.* Suppose there is a nonadaptive scheme using, without loss of generality, exactly $t$ probes. On input $x \in [u]$ the scheme probes cells $C_x \subseteq [s]$ where $|C_x| = t$. Let $s' \stackrel{\text{def}}{=} s_{\text{ph}}(\lfloor\sqrt{u}\rfloor, n, r)$. Note that $s' = \Omega(s_{\text{ph}})$ for $\beta$ large enough. It is sufficient to show that we must have $t > s'/2$; so assume to the contrary that $t \leq s'/2$. Each set $C_x$ is contained in $\binom{s-t}{s'-1-t}$ sets of size $s' - 1$. Since $\binom{s}{s'-1} = \binom{s-t}{s'-1-t}\binom{s}{t}/\binom{s'-1}{t}$ and $u \geq \lfloor\sqrt{u}\rfloor\, \binom{s}{t}/\binom{s'-1}{t}$ when $c$ is sufficiently large, there must be a set $U' \subseteq [u]$ of $\lfloor\sqrt{u}\rfloor$ elements such that $|\cup_{x\in U'} B_x| < s'$. But the cells $\cup_{x\in U'} B_x$ can encode a perfect hash function for any $n$ elements of $U'$, contradicting the definition of $s'$. □

Thus, a constant fraction of the data structure must be probed if $O(s_{\text{ph}})$ space is to be used. This should be compared to the upper bound of Theorem 9.6 that uses $O(\log n + \log\log u) = O(\log s_{\text{ph}})$ bit probes, of which just one is adaptive.

Using more space does not help much, e.g., for space $s = O(n \log u)$ we still have a lower bound of $\Omega(s)$ when $u = 2^{\Theta(n \log n)}$. This particular case is interesting, as a Membership data structure of $O(n \log u)$ bits allows queries using $O(\log u) = O(\sqrt{s/\log s})$ nonadaptive bit probes [BMRV00]. So for these parameters, Membership is *strictly easier* than Perfect Hashing among nonadaptive schemes.

## 9.5 Open problems

An apparent open problem is whether our nonexplicit data structures can be efficiently implemented on a RAM with a standard instruction set. In particular, explicit versions of Theorems 9.3 and 9.5 for small $u$ could be interesting from a practical point of view.

From a theoretical perspective we lack a tight bit probe bound for perfect hashing. The lower bound technique in the first part of the proof of Theorem 9.4 breaks down for range just slightly larger than $n$. On the other hand, the upper bound of Theorem 9.1 shows that very few bit probes suffice when the universe is small compared to the set, and the size of the range is not too close to $n$.

# Chapter 10

## One-probe search

The *dictionary* is one of the most well-studied data structures. A dictionary represents a set $S$ of elements (called *keys*) from some universe $U$, along with information associated with each key in the set. Any $x \in U$ can be looked up, i.e., it can be reported whether $x \in S$, and if so, what information is associated with $x$. We consider this problem on a unit cost word RAM in the case where keys and associated information have fixed size and are not too big (see below). The most straightforward implementation, an array indexed by the keys, has the disadvantage that the space usage is proportional to the size of $U$ rather than to the size of $S$. On the other hand, arrays are extremely time efficient: A single memory probe suffices to retrieve or update an entry.

It is easy to see that there exists no better deterministic one-probe dictionary than an array. In this chapter we investigate *randomized* one-probe search strategies, and show that it is possible, using much less space than an array implementation, to look up a given key with probability arbitrarily close to 1. The probability is over coin tosses performed by the lookup procedure. In case the memory probe did not supply enough information to answer the query, this is realized by the lookup procedure, and it produces the answer "don't know". In particular, by iterating until an answer is found, we get a Las Vegas lookup procedure that can have an expected number of probes arbitrarily close to 1.

It should be noted that one-probe search is impossible if one has no idea how much data is stored. We assume that the query algorithm knows the size of the data structure – a number that only changes when the size of the key set changes by a constant factor. The fact that the size, which may rarely or never change, is the only kind of global information needed to query the data structure means that it is well suited to support concurrent lookups (in parallel or distributed settings). In contrast, all known hash function based lookup schemes have some kind of global hash function that must be changed regularly. Even concurrent lookup of the *same* key, without accessing the same memory location, is supported to some extent by our dictionary. This is due to the fact that two lookups of the same key are not very likely to probe the same memory location.

A curious feature of our lookup procedure is that it makes its decision based on a constant number of *equality* tests – in this sense it is comparison-based. However, the data structure is not *implicit* in the sense of Munro and

Suwanda [MS80], as it stores keys not in $S$.

Our studies were inspired by recent work of Buhrman et al. [BMRV00] on randomized analogs of bit vectors. They presented a Monte Carlo data structure where one bit probe suffices to retrieve a given bit with probability arbitrarily close to 1. When storing a sparse bit vector (few 1s) the space usage is much smaller than that of a bit vector. When storing no associated information, a dictionary solves the *membership* problem, which can also be seen as the problem of storing a bit vector. Our Las Vegas lookup procedure is stronger than the Monte Carlo lookup procedure in [BMRV00], as a wrong answer is never returned. The price paid for this is an *expected* bound on the number of probes, a slightly higher space usage, and, of course, that we look up one word rather than one bit. The connection to [BMRV00] is also found in the underlying technique: We employ the same kind of unbalanced bipartite expander graph as is used there. Recently, explicit constructions[1] of such graphs with near-optimal parameters have been found [TS, TSUZ01].

Let $u = |U|$ and $n = |S|$. We assume that one word is large enough to hold one of $2u+1$ different symbols plus the information associated with a key. (Note that if this is not the case, it can be simulated by accessing a number of consecutive words rather than one word – an efficient operation in many memory models.) Our main theorem is the following:

**Theorem 10.1** *For any constant $\epsilon > 0$ there exists a nonexplicit one-probe dictionary with success probability $1 - \epsilon$, using $O(n \log \frac{2u}{n})$ words of memory. Also, there is an explicit construction using $n \cdot 2^{O((\log\log u)^3)}$ words of memory.*

Note that the space overhead for the nonexplicit scheme, a factor of $\log \frac{2u}{n}$, is exponentially smaller than that of an array implementation.

In the second part of the chapter we consider dynamic updates to the dictionary (insertions and deletions of keys). The fastest known dynamic dictionaries use hashing, i.e., they select at random a number of functions from suitable families, which are stored and subsequently used deterministically to direct searches.

A main point in this work is that a fixed structure with random properties (the expander graph) can be used to move random choices from the data structure itself to the lookup procedure. The absence of hash functions in our data structure has the consequence that updates can be performed in a very local manner. We show how to deterministically perform updates by probing and changing a number of words that is nearly linear in the degree of the expander graph (which, for optimal expanders, is at most logarithmic in the size of the universe). Current explicit expanders are not fast enough for our dynamic data structure to improve known results in a standard RAM model. However, if we augment the RAM with an instruction for computing neighbors in an optimal expander graph with given numbers of vertices, an efficient dynamic dictionary can be implemented.

[1] Where a given neighbor of a vertex can be computed in time polylogarithmic in the number of vertices.

**Theorem 10.2** *In the expander-augmented RAM model, there is a dictionary where a sequence of $a$ insertions/deletions and $b$ lookups in a key set of size at most $n$ takes time $O(a(\log \frac{2u}{n})^{1+o(1)}+b+t)$ with probability $1-2^{-\Omega(a+t/(\log \frac{2u}{n})^{1+o(1)})}$. The space usage is $O(n \log \frac{2u}{n})$ words.*

When the ratio between the number of updates and lookups is small, the expected average time per dictionary operation is constant. Indeed, if the fraction of updates is between $(\log \frac{2u}{n})^{-1-\Omega(1)}$ and $n^{-\omega(1)}$, and if $u = 2^{n^{1-\Omega(1)}}$, the above yields the best known probability, using space polynomial in $n$, that a sequence of dictionary operations take average constant time. The intuitive reason why the probability bound is so good, is that time consuming behavior requires bad random choices in many invocations of the lookup procedure, and that the random bits used in different invocations are *independent*.

## 10.1 Related work

As described above, this chapter is related to [BMRV00], in scope as well as in tools. The use of expander graphs in connection with the membership problem was earlier suggested by Fiat and Naor [FN93], as a tool for constructing an efficient implicit dictionary.

Yao [Yao81] showed an $\Omega(\log n)$ worst case lower bound on the time for dictionary lookups on a restricted RAM model allowing words to contain only keys of $S$ or special symbols from a fixed set whose size is a function of $n$ (e.g., pointers). The lower bound holds when space is bounded by a function of $n$, and $u$ is sufficiently large. It extends to give an $\Omega(\log n)$ lower bound for the expected time of randomized Las Vegas lookups.

Our data structure violates Yao's lower bound model in two ways: 1. We allow words to contain certain keys not in $S$ (accessed only through equality tests); 2. We allow space depending on $u$. The second violation is the important one, as Yao's lower bound can be extended to allow 1. Yao also considered deterministic one-probe schemes in his model, showing that, for $n \leq u/2$, a space usage of $u/2 + O(1)$ words is necessary and sufficient for them to exist.

The worst case optimal number of word probes for membership is studied in Chapter 9 in the case where $U$ equals the set of machine words. It was shown that *three* word probes are necessary when using $m$ words of space, unless $u = 2^{\Omega(n^2/m)}$ or $u \leq n^{2+o(1)}$. Sufficiency of three probes was shown for all parameters (in most cases it followed by the classic dictionary of Fredman et al. [FKS84]). In the expected sense, most hashing based dictionaries can be made to use arbitrarily close to 2 probes per lookup by expanding the size of the hash table by a constant factor.

Dictionaries with sublogarithmic lookup time that also allow efficient deterministic updates have been developed in a number of papers [AT00, BF99a, FW93], see also Chapters 6 and 7. Let $n$ denote an upper bound on the number of keys in a dynamic dictionary. For lookup time $t = o(\log \log n)$, the best known update time is $n^{O(1/t)}$, achieved in Chapter 6. The currently best *probabilistic* guarantee on dynamic dictionary performance, first achieved by Dietzfelbinger and Meyer auf der Heide in [DfMadH90], is that each operation

takes constant time with probability $1-O(m^{-c})$, where $c$ is any constant and $m$ is the space usage in words (which must be some constant factor larger than $n$). This implies that a sequence of $a$ insertions/deletions and $b$ lookups takes time $O(a+b+t)$ with probability $1-O(m^{-t/n})$.

## 10.2 Preliminaries

In this section we define $(n,d,\epsilon)$-expander graphs and state some results concerning these graphs. For the rest of this chapter we will assume $\epsilon$ to be a multiple of $1/d$, as this makes statements and proofs simpler. This will be without loss of generality, as the statements we show do not change when rounding $\epsilon$ down to the nearest multiple of $1/d$.

Let $G=(U,V,E)$ be a bipartite graph with left vertex set $U$, right vertex set $V$, and edge set $E$. We denote the set of neighbors of a set $S\subseteq U$ by $\Gamma(S)=\bigcup_{s\in S}\{v \mid (s,v)\in E\}$. We use $\Gamma(x)$ as a shorthand for $\Gamma(\{x\})$, $x\in U$.

**Definition 10.1** *A bipartite graph $G=(U,V,E)$ is $d$-regular if the degree of all nodes in $U$ is $d$. A bipartite $d$-regular graph $G=(U,V,E)$ is an $(n,d,\epsilon)$-expander if for each $S\subseteq U$ with $|S|\leq n$ it holds that $|\Gamma(S)|\geq(1-\epsilon)d|S|$.*

**Lemma 10.1** *For $0<\epsilon<1$ and $d\geq 1$, if $|V|\geq(1-\epsilon)dn(2u/n)^{1/\epsilon d}e^{1/\epsilon}$ then there exists an $(n,d,\epsilon)$-expander graph $G=(U,V,E)$, where $|U|=u$.*

*Proof.* Our proof is a standard application of the probabilistic method. Let $G=(U,V,E)$ be a randomly generated graph created by the following procedure. For each $u\in U$ choose $d$ neighbors with replacement, i.e., an edge can be chosen more than once, but then the double edges are removed. We will argue that the probability that this graph fails to be a $(n,d,\epsilon)$-expander graph is less than 1 for the choices of $|V|$ and $d$ as stated in the lemma. The degrees of the nodes in $U$ in this graph may be less than $d$, but if there exists a graph that is expanding with degree at most $d$ for all nodes, then there clearly exists a graph that is expanding with exactly degree $d$ as well.

We must bound the probability that some subset of $i\leq n$ vertices from $U$ has fewer than $(1-\epsilon)di$ neighbors. A subset $S\subseteq U$ of size $i$ can be chosen in $\binom{u}{i}$ ways and a set $V'\subseteq V$ of size $(1-\epsilon)di$ can be chosen in $\binom{|V|}{(1-\epsilon)di}$ ways. (Note that $|V|\geq(1-\epsilon)di$.) The probability that such a set $V'$ contains all of the neighbors for $S$ is $(\frac{(1-\epsilon)di}{|V|})^{di}$. Thus, the probability that some subset of $U$ of size $i\leq n$ has fewer than $(1-\epsilon)di$ neighbors is at most

$$\begin{aligned}
&\sum_{i=1}^{n}\binom{u}{i}\binom{|V|}{(1-\epsilon)di}\left(\frac{(1-\epsilon)di}{|V|}\right)^{di}\\
&<\sum_{i=1}^{n}\left(\frac{ue}{i}\right)^{i}\left(\frac{|V|e}{(1-\epsilon)di}\right)^{(1-\epsilon)di}\left(\frac{(1-\epsilon)di}{|V|}\right)^{di}\\
&\leq\sum_{i=1}^{n}\left(\left(\frac{(1-\epsilon)di}{|V|}\right)^{\epsilon d}e^{d}u/i\right)^{i}\quad.
\end{aligned}$$

If the term in the outermost parentheses is bounded by 1/2, the sum is less than 1. This is the case when $|V|$ fulfills the requirement stated in the lemma. □

**Corollary 10.1** *For any constants $\alpha, \epsilon > 0$ there exist an $(n, d, \epsilon)$-expander $G = (U, V, E)$ for the following parameters:*

- $|U| = u$, $d = O(\log(2u/n))$ *and* $|V| = O(n \log(2u/n))$.
- $|U| = u$, $d = O(1)$ *and* $|V| = O(n\,(2u/n)^{\alpha})$.

**Theorem 10.3** *(Ta-Shma [TS]) For any constant $\epsilon > 0$ and $d = 2^{O((\log\log u)^3)}$, there exists an* explicit *$(n, d, \epsilon)$-expander $G = (U, V, E)$ with $|U| = u$ and $|V| = n \cdot 2^{O((\log\log u)^3)}$.*

## 10.3 Static data structure

Let $S \subseteq U$ denote the key set we wish to store. Our data structure is an array denoted by $T$. Its entries may contain the symbol $x$ for keys $x \in S$, the symbol $\neg x$ for keys $x \in U \backslash S$, or the special symbol $\perp \notin U$. (Recall our assumption that one of these symbols plus associated information fits into one word.) For simplicity we will consider the case where there is no information associated with keys, i.e., we solve just the membership problem. Extending this to allow associated information is straightforward. We make use of a $(2n+1, d, \epsilon/2)$-expander with neighbor function $\Gamma$. Given that a random element in the set $\Gamma(x)$ can be computed quickly for $x \in U$, the one-probe lookup procedure is very efficient.

```
procedure lookup_ε(x)
    choose v ∈ Γ(x) at random;
    if T[v] = x then return 'yes'
    else if T[v] ∈ {¬x, ⊥} then return 'no'
    else return 'don't know';
end;
```

The corresponding Las Vegas lookup algorithm is the following:

```
procedure lookup(x)
    repeat
        choose v ∈ Γ(x) at random;
    until T[v] ∈ {x, ¬x, ⊥};
    if T[v] = x then return 'yes' else return 'no';
end;
```

### 10.3.1 Requirements to the data structure

The success probability of $\text{lookup}_\epsilon(x)$ and the expected time of $\text{lookup}(x)$ depends on the content of the entries indexed by $\Gamma(x)$ in $T$. To guarantee correct-

ness and success probability $1-\epsilon$ in each probe for $x$, the following conditions should hold:

1. If $x \in S$, at least a fraction $1-\epsilon$ of the entries $T[v]$, $v \in \Gamma(x)$, contain $x$, and none contain $\neg x$ or $\perp$.

2. If $x \notin S$, at least a fraction $1-\epsilon$ of the entries $T[v]$, $v \in \Gamma(x)$, contain either $\neg x$ or $\perp$, and none contain $x$.

By inserting $\perp$ in all entries of $T$ except the entries in $\Gamma(S)$, condition 2 will be satisfied for all $x \notin S$ with $|\Gamma(x) \cap \Gamma(S)| \leq \epsilon d$. A key notion in this chapter is the set of *$\epsilon$-ghosts* for a set $S$, which are the keys of $U$ that have many neighbors in common with $S$. For each $\epsilon$-ghost $x$ we will need some entries in $T$ with content $\neg x$.

**Definition 10.2** *Given a bipartite graph $G = (U, V, E)$, a key $x \in U$ is an* $\epsilon$-ghost *for the set $S \subseteq U$ if $|\Gamma(x) \cap \Gamma(S)| > \epsilon|\Gamma(x)|$ and $x \notin S$.*

**Lemma 10.2** (Buhrman et al. [BMRV00]) *There are at most $n$ $\epsilon$-ghosts for a set $S$ of size $n$ in a $(2n+1, d, \epsilon/2)$-expander graph.*

In order to fulfill conditions 1 and 2, we need to assign entries in $T$ to the keys in $S$ and to the $\epsilon$-ghosts for $S$.

**Definition 10.3** *Let $G = (U, V, E)$ be a bipartite $d$-regular graph and let $0 < \epsilon < 1$. An* assignment *for a set $S \subseteq U$, is a subset $A \subseteq E \cap (S \times \Gamma(S))$ such that for $v \in \Gamma(S)$, $|A \cap (S \times \{v\})| = 1$. A* $(1-\epsilon)$-balanced assignment *for $S$ is an assignment $A$, where for each $s \in S$ it holds that $|A \cap (\{s\} \times \Gamma(s))| \geq (1-\epsilon)d$.*

**Lemma 10.3** *If a graph $G = (U, V, E)$ is an $(n, d, \epsilon)$-expander then there exists a $(1-\epsilon)$-balanced assignment for every set $S \subseteq U$ of size at most $n$.*

To show the lemma we will use Hall's theorem [Hal35]. A *perfect matching* in a bipartite graph $(U, V, E)$ is a set of $|U|$ edges such that for each $x \in U$ there is an edge $(x, v) \in E$, and for each $v \in V$ there is at most one edge $(x, v) \in E$.

**Theorem 10.4** (Hall's theorem) *In any bipartite graph $G = (U, V, E)$, where for each subset $U' \subseteq U$ it holds that $|U'| \leq |\Gamma(U')|$, there exists a perfect matching.*

*Proof of Lemma 10.3.* Let $S$ be an arbitrary subset of $U$ of size $n$. Let $G' = (S, \Gamma(S), E')$ be the subgraph of $G$ induced by the nodes $S$ and $\Gamma(S)$, i.e., $E' = \{(s, v) \in E \mid s \in S\}$. To prove the lemma we want to show that there exists an assignment $A$ such that for each $s \in S$, $|A \cap (\{s\} \times \Gamma(s))| \geq (1-\epsilon)d$. The idea is to use Hall's theorem $(1-\epsilon)d$ times by repeatedly finding a perfect matching and removing the nodes from $\Gamma(S)$ in the matching.

Since $G$ is an $(n, d, \epsilon)$-expander we know that for each subset $S' \subseteq S$ it holds that $|\Gamma(S')| \geq (1-\epsilon)d|S'|$. Assume that we have $i$ perfect matchings from $S$ to non-overlapping subsets of $\Gamma(S)$ and denote by $M$ the nodes from $\Gamma(S)$ in the

matchings. For each subset $S' \subseteq S$ it holds that $|\Gamma(S')\backslash M| \geq ((1-\epsilon)d - i)|S'|$. If $(1-\epsilon)d - i \geq 1$ then the condition in Hall's theorem holds for the graph $G_i = (S, (\Gamma(S)\backslash M), E'\backslash E_i)$, where $E_i$ is the set of edges incident to nodes in $M$, and there exists a perfect matching in $G_i$. From this it follows that at least $(1-\epsilon)d$ non-overlapping (in $\Gamma(S)$) perfect matchings can be found in $G'$. The edges in the matchings define a $(1-\epsilon)$-balanced assignment. □

### 10.3.2 Construction

We store the set $S$ as follows:

1. Write $\perp$ in all entries not in $\Gamma(S)$.
2. Find the set $\bar{S}$ of $\epsilon$-ghosts for $S$.
3. Find a $(1-\epsilon)$-balanced assignment for the set $S \cup \bar{S}$.
4. For $x \in S$ write $x$ in entries assigned to $x$.
   For $x \in \bar{S}$ write $\neg x$ in entries assigned to $x$ in $\Gamma(S)$.

By Lemma 10.2 the set $\bar{S}$ found in step 2 contains at most $n$ keys, and by Lemma 10.3 it is possible to carry out step 3. Together with the results on expanders in Section 10.2, this concludes the proof of Theorem 10.1.

We note that step 2 takes time $\Omega(|U\backslash S|)$ if we have only oracle access to $\Gamma$. When the graph has some structure it is sometimes possible to do much better. Ta-Shma shows in [TS] that this step can be performed for his class of graphs in time polynomial in the size of the right vertex set, i.e., polynomial in the space usage. All other steps are clearly also polynomial time in the size of the array.

In the dynamic setting, covered in Section 10.4, we will take an entirely different approach to ghosts, namely, we care about them only if we see them. We then argue that the time spent looking for a particular ghost before it is detected is not too large, and that there will not be too many different ghosts.

## 10.4 Dynamic updates

In this section we show how to implement efficient dynamic insertions and deletions of keys in our dictionary. We will use a slightly stronger expander graph than in the static case, namely a $(4n', d, \epsilon/3)$-expander where $n'$ is an upper bound on the size of the set that can be handled. The parameter $n'$ is assumed to be known to the query algorithm. Note that $n'$ can be kept in the range, say, $n$ to $2n$ at no asymptotic cost, using standard global rebuilding techniques. Our dynamic dictionary essentially maintains the static data structure described in the previous section. Additionally, we maintain the following auxiliary data structures:

- A priority queue with all keys in $S$ plus some set $\bar{S}$ of keys that appear negated in $T$. Each key has as priority the size of its assignment, which is always at least $(1-\epsilon)d$.

- Each entry $T[v]$ in $T$ is augmented with
  - A pointer $T_p[v]$ which, if entry $v$ is assigned to a key, points to that key in the priority queue.
  - A counter $T_c[v]$ that at any time stores the number of keys in $S$ that have $v$ as a neighbor.

Since all keys in the priority queue are assigned $(1-\epsilon)d$ entries in $T$, the performance of the lookup procedure is the desired one, except when searching for $\epsilon$-ghosts not in $\bar{S}$. We will discuss this in more detail later.

### 10.4.1 Performing updates

We first note that it is easy to maintain the data structure during deletions. All that is needed when deleting $x \in S$ is decreasing the counters $T_c[v]$, $v \in \Gamma(x)$, and replacing $x$ with $\neg x$ or $\perp$ (the latter if the counter reaches 0). Finally, $x$ should be removed from the priority queue. We use a simple priority queue that requires space $O(d+n)$, supports `insert` in $O(d)$ time, and `increasekey`, `decreasekey`, `findmin` and `delete` in $O(1)$ time. The total time for a deletion in our dictionary is $O(d)$.

When doing insertions we have to worry about maintaining a $(1-\epsilon)$-balanced assignment. The idea of our insertion algorithm is to assign *all* neighbors to the key being inserted. In case this makes the assignment of other keys too small (easily seen using the priority queue), we repeat assigning all neighbors to them, and so forth. Every time an entry in $T$ is reassigned to a new key, the priority of the old and new key are adjusted in the priority queue. The time for an insertion is $O(d)$, if one does not count the associated cost of maintaining assignments of other keys. The analysis in Section 10.4.2 will bound this cost. Note that a priori it is not even clear whether the insertion procedure terminates.

A final aspect that we have to deal with is ghosts. Ideally we would like $\bar{S}$ to be at all times the current set of $\epsilon$-ghosts for $S$, such that a $(1-\epsilon)$-balanced assignment was maintained for all ghosts. However, this leaves us with the hard problem of finding new ghosts as they appear. We circumvent this problem by only including keys in $\bar{S}$ if they are selected for examination and found to be $\epsilon$-ghosts. A key is selected for examination if a lookup of that key takes more than $\log_{1/\epsilon} d$ iterations. The time spent on examinations and on lookups of a ghost before it is found, is bounded in the next section.

The sequence of operations is divided up into stages, where each stage (except possibly the last) contains $n'$ insert operations. After the last insertion in a stage, all keys in $\bar{S}$ that are no longer $\epsilon$-ghosts are deleted. This is done by going through all keys in the priority queue. Keys of $\bar{S}$ with at least $(1-\epsilon)d$ neighbors containing $\perp$ are removed from the priority queue. Hence, when a new stage starts, $\bar{S}$ will only contain $\epsilon$-ghosts.

### 10.4.2 Analysis

We now sketch the analysis of our dynamic dictionary. First, the total work spent doing assignments and reassignments is analyzed. Recall that the algo-

rithm maintains a $(1-\epsilon)$-balanced assignment for the set $S \cup \bar{S}$ of keys in the priority queue. Keys enter the priority queue when they are inserted in $S$, and they may enter it when they are $\epsilon$-ghosts for the current set. It clearly suffices to bound the total work in connection with insertions in the priority queue, as the total work for deletions cannot be larger than this. We will first show a bound on the number of keys in $\bar{S}$.

**Lemma 10.4** *The number of keys in the set $\bar{S}$ never exceeds $2n'$.*

*Proof.* Let $S$ be the set stored at the beginning of a stage. $\bar{S}$ only contains $\epsilon$-ghosts for $S$ at this point. Let $S'$ denote the keys inserted during the stage. New keys inserted into $\bar{S}$ have to be $\epsilon$-ghosts for $S \cup S'$. According to Lemma 10.2, the fact that $|S \cup S'| \leq 2n'$ implies that there are at most $2n'$ $\epsilon$-ghosts for $S \cup S'$ (including the $\epsilon$-ghosts for $S$). Thus, the number of keys in $\bar{S}$ during any stage is at most $2n'$. □

It follows from the lemma that the number of insertions in the priority queue is bounded by 3 times the number of insertions performed in the dictionary. The remainder of our analysis of the number of reassignments has two parts: We first show that our algorithm performs a number of reassignments (in connection with insertions) that is within a constant factor of *any* scheme maintaining a $(1-\epsilon/3)$-balanced assignment. The scheme we compare ourselves to may be *off-line*, i.e., know the sequence of operations in advance. Secondly, we give an off-line strategy for maintaining a $(1-\epsilon/3)$-balanced assignment using $O(d)$ reassignments per update. This proof strategy was previously used for an assignment problem by Brodal and Fagerberg [BF99b].

In the following lemmas, the set $M$ is the set for which a balanced assignment is maintained, and the insert and delete operations are insertions and deletions in this set. In our data structure $M$ corresponds to $S \cup \bar{S}$.

**Lemma 10.5** *Let $G = (U, V, E)$ be a $d$-regular bipartite graph. Suppose $O$ is a sequence of insert and delete operations on a dynamic set $M \subseteq U$. Let $B$ be an algorithm that maintains a $(1-\frac{\epsilon}{3})$-balanced assignment for $M$, and let $C$ be our "assign all" scheme for maintaining a $(1-\epsilon)$-balanced assignment for $M$. If $B$ makes at most $k$ reassignments during $O$, then $C$ assigns all neighbors to a key at most $\frac{3}{\epsilon}(k/d + |M|_{\text{start}})$ times, where $|M|_{\text{start}}$ is the initial size of $M$.*

*Proof.* To show the lemma we will argue that the assignment of $C$, denoted $A_C$, will become significantly "less different" from the assignment of $B$, denoted $A_B$, each time $C$ assigns all neighbors of a key to that key. At the beginning $|A_B \backslash A_C| \leq d|M|_{\text{start}}$, since $|A_B| \leq d|M|_{\text{start}}$. Each of the $k$ reassignments $B$ performs causes $|A_B \backslash A_C|$ to increase by at most one. This means that the reassignments made by $C$ during $O$ can decrease $|A_B \backslash A_C|$ by at most $k + d|M|_{\text{start}}$ in total.

Each time $C$ assigns all entries in $\Gamma(x)$ to a key $x$, at least $\epsilon d$ reassignments are done, since the assignment for $x$ had size less than $(1-\epsilon)d$ before the reassignment. At this point at least $(1-\frac{\epsilon}{3})d$ pairs $(x, e)$ are included in $A_B$, i.e., at most $\frac{\epsilon}{3}d$ of the neighbors of $x$ are not assigned to $x$ in $A_B$. This means

that at least $\frac{2\epsilon}{3}d$ of the reassignments made by $C$ decrease $|A_B \backslash A_C|$, while at most $\frac{\epsilon}{3}d$ reassignments may increase $|A_B \backslash A_C|$. In total, $|A_B \backslash A_C|$ is decreased by at least $\frac{\epsilon}{3}d$ when $C$ assigns all neighbors to a key. The lemma now follows, as $|A_B \backslash A_C|$ can decrease by $\frac{\epsilon}{3}d$ at most $(k + d|M|_{\text{start}})/(\frac{\epsilon}{3}d)$ times. □

**Lemma 10.6** *Let $G = (U, V, E)$ be a $(4n', d, \epsilon/3)$-expander. There exists an off-line algorithm maintaining a $(1 - \frac{\epsilon}{3})$-balanced assignment for a dynamic set $M \subseteq U$, during a stage of $3n'$ insertions, by performing at most $4dn'$ reassignments, where $|M| \leq n'$ at the beginning of the stage.*

*Proof.* Let $M'$ be the set of $3n'$ keys to insert. Define $\tilde{M} = M \cup M'$; we have $|\tilde{M}| \leq 4n'$. Let $A_{\tilde{M}}$ be a $(1 - \frac{\epsilon}{3})$-balanced assignment for $\tilde{M}$ (shown to exist in Lemma 10.3).

The off-line algorithm knows the set $M'$ of keys to insert from the start, and does the following. First, it assigns neighbors to the keys in $M$ according to the assignment $A_{\tilde{M}}$, which requires at most $dn'$ reassignments. Secondly, for each insertion of a key $x \in M'$, it assigns neighbors to $x$ according to $A_{\tilde{M}}$, which requires at most $d$ reassignments. This will not cause any key already in the set to lose an assigned neighbor, hence no further reassignments are needed to keep the assignment $(1 - \frac{\epsilon}{3})$-balanced. It follows that he total number of reassignments during the $3n'$ insertions is at most $4dn'$, proving the lemma. □

The above two lemmas show that in a sequence of $a$ updates to the dictionary there are $O(a)$ insertions in the priority queue, each of which gives rise to $O(d)$ reassignments in a certain off-line algorithm, meaning that our algorithm uses $O(ad)$ time for maintaining a $(1 - \epsilon)$-balanced assignment for the set $S \cup \bar{S}$ in the priority queue.

We now turn to analyzing the work done in the lookup procedure. First we will bound the number of iterations in all searches for keys that are *not* undetected $\epsilon$-ghosts. Each iteration has probability at least $1 - \epsilon$ of succeeding, independently of all other events, so we can bound the probability of many iterations using Chernoff bounds. In particular, the probability that the total number of iterations used in the $b$ searches exceeds $\frac{2}{1-\epsilon}b + t$ is less than $e^{-\frac{1-\epsilon}{4}t}$.

When searching for a key that is not an undetected $\epsilon$-ghost, the probability of selecting it for examination is bounded from above by $1/d$. In particular, by Chernoff bounds we get that, for $k > 0$, the total number of examinations during all $b$ lookups is at most $b/d + k$ with probability $1 - (\frac{e}{1+kd/b})^{b/d+k}$. For $k = (2e - 1)b/d + t/d$ we get that the probability of more than $2eb/d + t/d$ examinations is bounded by $2^{-t/d}$. Each examination costs time $O(d)$, so the probability of spending $O(b+t)$ time on such examinations is at least $1 - 2^{-t/d}$.

We now bound the work spent on finding $\epsilon$-ghosts. Recall that an $\epsilon$-ghost is detected if it is looked up, and the number of iterations used by the lookup procedure exceeds $\log_{1/\epsilon} d$. Since we have an $\epsilon$-ghost, the probability that a single lookup selects the ghost for examination is at least $\epsilon^{\log_{1/\epsilon} d - 1} = \Omega(1/d)$. We define $d' = O(d)$ by $1/d' = \epsilon^{\log_{1/\epsilon} d - 1}$. Recall that there are at most $2n'$ $\epsilon$-ghosts in a stage, and hence at most $2a$ in total. We bound the probability

that more than $4ad' + k$ lookups are made on undetected $\epsilon$-ghosts, for $k > 0$. By Chernoff bounds the probability is at most $e^{-k/4d'}$. Each lookup costs $O(\log d)$ time, so the probability of using time $O(ad \log d + t)$ is at least $1 - e^{-t/4d' \log d}$.

In summary, we have bounded the time spent on four different tasks in our dictionary:

- The time spent looking up keys that are not undetected $\epsilon$-ghosts is $O(b+t)$ with probability $1 - 2^{-\Omega(t)}$.
- The time spent examining keys that are not undetected $\epsilon$-ghosts is $O(b+t)$ with probability $1 - 2^{-\Omega(t/d)}$.
- The time spent looking up $\epsilon$-ghosts before they are detected is $O(ad \log d + t)$ with probability $1 - 2^{-\Omega(t/d \log d)}$.
- The time spent assigning, reassigning and doing bookkeeping is $O(ad)$.

Using the above with the first expander of Corollary 10.1, having degree $d = O(\log \frac{2u}{n'})$, we get the performance bound stated in Theorem 10.2. Using the constant degree expander of Corollary 10.1 we get a data structure with constant time updates. This can also be achieved in this space with a trie, but a trie would use around $1/\alpha$ word probes for lookups of keys in the set, rather than close to 1 word probe, expected.

## 10.5 Conclusion and open problems

In this chapter we studied dictionaries for which a single word probe with good probability suffices to retrieve any given key with associated information. The main open problem we leave is whether the space usage of our dictionary is the best possible for one-probe search.

It is known that three word probes are necessary and sufficient in the worst case for lookups in dictionaries, even when using superlinear space. An obvious open question is how well one can do using two word probes and a randomized lookup procedure. Can the space utilization be substantially improved? Another point is that we bypass Yao's lower bound by using space dependent on $u$. An interesting question is: How large a dependence on $u$ is necessary to get around Yao's lower bound. Will space $n \log^* u$ do, for example?

# Bibliography


[ABI86] Noga Alon, László Babai, and Alon Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *J. Algorithms*, 7(4):567–583, 1986.

[ABKU99] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. *SIAM J. Comput.*, 29(1):180–200, 1999.

[ABR01] Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. Optimal static range reporting in one dimension. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC '01)*, pages 476–482, 2001.

[ADfM+97] Noga Alon, Martin Dietzfelbinger, Peter Bro Miltersen, Erez Petrank, and Gábor Tardos. Is linear hashing good? In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC '97)*, pages 465–474. ACM Press, 1997.

[AGHP92] Noga Alon, Oded Goldreich, Johan Håstad, and René Peralta. Simple constructions of almost $k$-wise independent random variables. *Random Structures & Algorithms*, 3(3):289–304, 1992.

[AH87] Leonard M. Adleman and Ming-Deh Huang. Recognizing primes in random polynomial time. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC '87)*, pages 462–469. ACM Press, 1987.

[AHNR98] Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in linear time? *J. Comput. System Sci.*, 57(1):74–93, 1998.

[AHR98] Stephen Alstrup, Thore Husfeldt, and Theis Rauhe. Marked ancestor problems. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science (FOCS '98)*, pages 534–543. IEEE Comput. Soc. Press, 1998.

[AHU75] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley Publishing Co., Reading, Mass.-London-Amsterdam, 1975. Second printing, Addison-Wesley Series in Computer Science and Information Processing.

[AL86] Alfred V. Aho and David Lee. Storing a dynamic sparse table. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science (FOCS '82)*, pages 55–60. IEEE Comput. Soc. Press, 1986.

[Alo86] Noga Alon. Eigenvalues and expanders. *Combinatorica*, 6(2):83–96, 1986.

[AMRT96] Arne Andersson, Peter Bro Miltersen, Søren Riis, and Mikkel Thorup. Static dictionaries on $AC^0$ RAMs: Query time $\Theta(\sqrt{\log n / \log \log n})$ is necessary and sufficient. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science (FOCS '96)*, pages 441–450. IEEE Comput. Soc. Press, 1996.

[AMT99] Arne Andersson, Peter Bro Miltersen, and Mikkel Thorup. Fusion trees can be implemented with $AC^0$ instructions only. *Theoret. Comput. Sci.*, 215(1-2):337–344, 1999.

[AN96] Noga Alon and Moni Naor. Derandomization, witnesses for Boolean matrix multiplication and construction of perfect hash functions. *Algorithmica*, 16(4-5):434–449, 1996.

[And96] Arne Andersson. Faster deterministic sorting and searching in linear space. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science (FOCS '96)*, pages 135–141. IEEE Comput. Soc. Press, 1996.

[AT00] Arne Andersson and Mikkel Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC '00)*, pages 335–342. ACM Press, 2000.

[AV88] Alok Aggarwal and Jeffrey S. Vitter. The Input/Output Complexity of Sorting and Related Problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[AVL62] Georgii Maksimovich Adel'son-Vel'skii and Evgenii Mikhailovich Landis. An algorithm for organization of information. *Dokl. Akad. Nauk SSSR*, 146:263–266, 1962.

[BAG97] Amir M. Ben-Amram and Zvi Galil. When can we sort in $o(n \log n)$ time? *J. Comput. System Sci.*, 54(2, part 2):345–370, 1997.

[BCFM00] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations. *J. Comput. System Sci.*, 60(3):630–659, 2000.

[BCH86] Paul W. Beame, Stephen A. Cook, and H. James Hoover. Log depth circuits for division and related problems. *SIAM J. Comput.*, 15(4):994–1003, 1986.

[BCSV00] Petra Berenbrink, Artur Czumaj, Angelika Steger, and Berthold Vöcking. Balanced allocations: the heavily loaded case. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC '00)*, pages 745–754. ACM Press, 2000.

[BF99a] Paul Beame and Faith Fich. Optimal bounds for the predecessor problem. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing (STOC '99)*, pages 295–304. ACM Press, 1999.

[BF99b] Gerth Stølting Brodal and Rolf Fagerberg. Dynamic representations of sparse graphs. In *Proceedings of the 6th International Workshop on Algorithms and Data Structures (WADS '99)*, volume 1663 of *Lecture Notes in Computer Science*, pages 342–351. Springer-Verlag, 1999.

[BH91] Holger Bast and Torben Hagerup. Fast and reliable parallel hashing. In *3rd Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '91)*, pages 50–61. ACM Press, 1991.

[BK00] Andrei Z. Broder and Anna R. Karlin. Multilevel adaptive hashing. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '90)*, pages 43–53. ACM Press, 2000.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

[BM94] Andrej Brodnik and J. Ian Munro. Membership in constant time and minimum space. In *Proceedings of the 2nd European Symposium on Algorithms (ESA '94)*, volume 855 of *Lecture Notes in Computer Science*, pages 72–81. Springer-Verlag, 1994.

[BM99] Andrej Brodnik and J. Ian Munro. Membership in constant time and almost-minimum space. *SIAM J. Comput.*, 28(5):1627–1640, 1999.

[BM01] Andrei Broder and Michael Mitzenmacher. Using multiple hash functions to improve IP lookups. Proceedings of INFOCOM 2001, 2001.

[BMM97] Andrej Brodnik, Peter Bro Miltersen, and J. Ian Munro. Transdichotomous algorithms without multiplication — some upper and lower bounds. In *Proceedings of the 5th Workshop on Algorithms and Data Structures (WADS '97)*, volume 1272 of *Lecture Notes in Computer Science*, pages 426–439. Springer-Verlag, 1997.

[BMRV00] Harry Buhrman, Peter Bro Miltersen, Jaikumar Radhakrishnan, and S. Venkatesh. Are bitvectors optimal? In *Proceedings*

*of the 32nd Annual ACM Symposium on Theory of Computing (STOC '00)*, pages 449–458. ACM Press, 2000.

[Bre73] Richard P. Brent. Reducing the retrieval time of scatter storage techniques. *Communications of the ACM*, 16(2):105–109, 1973.

[CFG+78] Larry Carter, Robert Floyd, John Gill, George Markowsky, and Mark Wegman. Exact and approximate membership testers. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing (STOC '78)*, pages 59–65. ACM Press, 1978.

[CGH+85] Benny Chor, Oded Goldreich, Johan Hastad, Joel Friedman, Steven Rudich, and Roman Smolensky. The bit extraction problem of t-resilient functions (preliminary version). In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science (FOCS '85)*, pages 396–407. IEEE Comput. Soc. Press, 1985.

[Chi94] Andrew Chin. Locality-preserving hash functions for general purpose parallel computation. *Algorithmica*, 12(2-3):170–181, 1994.

[CHM92] Zbigniew J. Czech, George Havas, and Bohdan S. Majewski. An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters*, 43(5):257–264, 1992.

[CHM97] Zbigniew J. Czech, George Havas, and Bohdan S. Majewski. Perfect hashing. *Theoretical Computer Science*, 182(1–2):1–143, 1997.

[CS97] Arthur Czumaj and Volker Stemann. Randomized allocation processes. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS '97)*, pages 194–203. IEEE Comput. Soc. Press, 1997.

[CSV84] Ashok K. Chandra, Larry Stockmeyer, and Uzi Vishkin. Constant depth reducibility. *SIAM J. Comput.*, 13(2):423–439, 1984.

[CW79] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *J. Comput. System Sci.*, 18(2):143–154, 1979.

[Df96] Martin Dietzfelbinger. Universal hashing and $k$-wise independent random variables via integer arithmetic without primes. In *Proceedings of the 13th Symposium on Theoretical Aspects of Computer Science (STACS '96)*, pages 569–580. Springer-Verlag, 1996.

[DfGMP92] Martin Dietzfelbinger, Joseph Gil, Yossi Matias, and Nicholas Pippenger. Polynomial hash functions are reliable (extended abstract). In *Proceedings of the 19th International Colloquium on Automata, Languages and Programming (ICALP '92)*, volume 623 of *Lecture Notes in Computer Science*, pages 235–246. Springer-Verlag, 1992.

[DfHKP97] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19–51, 1997.

[DfKM$^+$94] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994.

[DfMadH90] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming (ICALP '90)*, volume 443 of *Lecture Notes in Computer Science*, pages 6–19. Springer-Verlag, 1990.

[DH01] Martin Dietzfelbinger and Torben Hagerup. Simple minimal perfect hashing in less space. In *Proceedings of the 9th European Symposium on Algorithms (ESA '01)*, volume 2161 of *Lecture Notes in Computer Science*, pages 109–120. Springer-Verlag, 2001.

[DMadH92] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. High performance universal hashing, with applications to shared memory simulations. In *Data structures and efficient algorithms*, volume 594 of *Lecture Notes in Computer Science*, pages 250–269. Springer, 1992.

[Dum56] Arnold I. Dumey. Indexing for rapid random access memory systems. *Computers and Automation*, 5(12):6–9, 1956.

[EIS76] Shimon Even, Alon Itai, and Adi Shamir. On the complexity of timetable and multicommodity flow problems. *SIAM J. Comput.*, 5(4):691–703, 1976.

[FCH92] Edward A. Fox, Qi Fan Chen, and Lenwood S. Heath. A faster algorithm for constructing minimal perfect hash functions. In *Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Data Structures, pages 266–273. ACM Press, 1992.

[FHCD92] Edward A. Fox, Lenwood S. Heath, Qi Fan Chen, and Amjad M. Daoud. Practical minimal perfect hash functions for large databases. *Communications of the ACM*, 35(1):105–121, 1992.

[FK84] Michael L. Fredman and János Komlós. On the size of separating systems and families of perfect hash functions. *SIAM Journal on Algebraic and Discrete Methods*, 5(1):61–68, 1984.

[FKS84] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. Assoc. Comput. Mach.*, 31(3):538–544, 1984.

[FLPR99] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS '99)*, pages 285–297. IEEE Comput. Soc. Press, 1999.

[FM95] Faith Fich and Peter Bro Miltersen. Tables should be sorted (on random access machines). In *Proceedings of the the 4th Workshop on Algorithms and Data Structures (WADS '95)*, volume 955 of *Lecture Notes in Computer Science*, pages 482–493. Springer-Verlag, 1995.

[FN93] Amos Fiat and Moni Naor. Implicit $O(1)$ probe search. *SIAM J. Comput.*, 22(1):1–10, 1993.

[FNSS92] Amos Fiat, Moni Naor, Jeanette P. Schmidt, and Alan Siegel. Nonoblivious hashing. *Journal of the ACM*, 39(4):764–782, 1992.

[Fre60] Edward Fredkin. Trie memory. *Comm. A.C.M.*, 3(9):490–499, 1960.

[FS89] Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing (STOC '89)*, pages 345–354. ACM Press, 1989.

[FW93] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. System Sci.*, 47:424–436, 1993.

[FW94] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. System Sci.*, 48(3):533–551, 1994.

[GG81] Ofer Gabber and Zvi Galil. Explicit constructions of linear-sized superconcentrators. *J. Comput. System Sci.*, 22(3):407–420, 1981.

[GM79] Gaston H. Gonnet and J. Ian Munro. Efficient ordering of hash tables. *SIAM J. Comput.*, 8(3):463–478, 1979.

[Gon84] Gaston Gonnet. *Handbook of Algorithms and Data Structures.* Addison-Wesley Publishing Co., 1984.

[GW97] Oded Goldreich and Avi Wigderson. Tiny families of functions with random properties: A quality-size trade-off for hashing. *Random Structures & Algorithms*, 11(4):315–343, 1997.

[Hag98a] Torben Hagerup. Simpler and faster dictionaries on the $AC^0$ RAM. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP '98)*, volume 1443 of *Lecture Notes in Computer Science*, pages 79–90. Springer-Verlag, 1998.

[Hag98b] Torben Hagerup. Sorting and searching on the word RAM. In *Proceedings of the 15th Symposium on Theoretical Aspects of Computer Science (STACS '98)*, volume 1373 of *Lecture Notes in Computer Science*, pages 366–398. Springer-Verlag, 1998.

[Hag99] Torben Hagerup. Fast deterministic construction of static dictionaries. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '99)*, pages 414–418. ACM Press, 1999.

[Hal35] Philip Hall. On representatives of subsets. *J. London Math. Soc.*, 10:26–30, 1935.

[Han02] Yijie Han. Deterministic sorting in $O(n \log\log n)$ time and linear space. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC '02)*. ACM Press, 2002.

[HBI79] David R. Heath-Brown and Henryk Iwaniec. On the difference between consecutive primes. *Invent. Math.*, 55(1):49–69, 1979.

[HM93] George Havas and Bohdan S. Majewski. Graph-theoretic obstacles to perfect hashing. In *Proceedings of the 24th Southeastern International Conference on Combinatorics, Graph Theory, and Computing*, volume 98, pages 81–93. Utilitas Mathematica, 1993.

[HMP01] Torben Hagerup, Peter Bro Miltersen, and Rasmus Pagh. Deterministic dictionaries. *J. Algorithms*, 41(1):69–85, 2001.

[HT01] Torben Hagerup and Torsten Tholey. Efficient minimal perfect hashing in nearly minimal space. In *Proceedings of the 18th Symposium on Theoretical Aspects of Computer Science (STACS '01)*, volume 2010 of *Lecture Notes in Computer Science*, pages 317–326. Springer-Verlag, 2001.

[IMRV99] Piotr Indyk, Rajeev Motwani, Prabhakar Raghavan, and Santosh Vempala. Locality-preserving hashing in multidimensional spaces. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC '97)*, pages 618–625. ACM Press, 1999.

[Jan93] Svante Janson. Large deviation inequalities for sums of indicator variables. Technical Report 34, Department of Mathematics, Uppsala University, 1993.

[Juk00] Stasys Jukna. *Extremal Combinatorics with Applications in Computer Science*. Springer-Verlag, 2000.

[KL96] Jyrki Katajainen and Michael Lykke. Experiments with universal hashing. Technical Report DIKU Technical Report 96/8, University of Copenhagen, 1996.

[KLMadH96] Richard M. Karp, Michael Luby, and Friedhelm Meyer auf der Heide. Efficient PRAM simulation on a distributed memory machine. *Algorithmica*, 16(4-5):517–542, 1996.

[Knu73] Donald E. Knuth. *The Art of Computer Programming*. Addison-Wesley Publishing Co., Reading, Mass., 1973. Volume 3. Sorting and searching.

[Knu98] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley Publishing Co., Reading, Mass., second edition, 1998.

[LS98] Nathan Linial and Ori Sasson. Non-expansive hashing. *Combinatorica*, 18(1):121–132, 1998.

[Mad80] J. A. T. Maddison. Fast lookup in hash tables with direct rehashing. *The Computer Journal*, 23(2):188–189, May 1980.

[Mal77] Efrem G. Mallach. Scatter storage techniques: A uniform viewpoint and a method for reducing retrieval times. *The Computer Journal*, 20(2):137–140, May 1977.

[Mar] George Marsaglia. The Marsaglia random number CDROM including the diehard battery of tests of randomness. http://stat.fsu.edu/pub/diehard/.

[McG] Catherine C. McGeoch. The fifth DIMACS challenge dictionaries. http://cs.amherst.edu/∼ccm/challenge5/dicto/.

[Meh84] Kurt Mehlhorn. *Data structures and algorithms. 1, Sorting and searching*. Springer-Verlag, 1984.

[Mil96] Peter Bro Miltersen. Lower bounds for static dictionaries on RAMs with bit operations but no multiplication. In *Proceedings of the 23rd International Colloquium on Automata, Languages and Programming (ICALP '96)*, volume 1099 of *Lecture Notes in Computer Science*, pages 442–453. Springer-Verlag, 1996.

[Mil98] Peter Bro Miltersen. Error correcting codes, perfect hashing circuits, and deterministic dynamic dictionaries. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '98)*, pages 556–563. ACM Press, 1998.

[MN99] Kurt Mehlhorn and Stefan Näher. *LEDA. A platform for combinatorial and geometric computing.* Cambridge University Press, 1999.

[MNSW98] Peter Bro Miltersen, Noam Nisan, Shmuel Safra, and Avi Wigderson. On data structures and asymmetric communication complexity. *J. Comput. System Sci.*, 57(1):37–49, 1998. Proceedings of the 27th Annual ACM Symposium on Theory of Computing (STOC '95).

[MP69] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry.* MIT Press, 1969.

[MS77] F. Jessie MacWilliams and Neil J. A. Sloane. *The Theory of Error-Correcting Codes. I.* North-Holland Publishing Co., 1977. North-Holland Mathematical Library, Vol. 16.

[MS80] J. Ian Munro and Hendra Suwanda. Implicit data structures for fast search and update. *J. Comput. System Sci.*, 21(2):236–250, 1980.

[MV84] Kurt Mehlhorn and Uzi Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Inform.*, 21(4):339–374, 1984.

[Nis96] Noam Nisan. Extracting randomness: How and why: A survey. In *Proceedings of the 11th Annual IEEE Conference on Computational Complexity (CCC '96)*, pages 44–58. IEEE Comput. Soc. Press, 1996.

[NN93] Joseph Naor and Moni Naor. Small-bias probability spaces: Efficient constructions and applications. *SIAM J. Comput.*, 22(4):838–856, 1993.

[ÖP02a] Anna Östlin and Rasmus Pagh. One-probe search. To appear in proceedings of ICALP 2002, 2002.

[ÖP02b] Anna Östlin and Rasmus Pagh. Simulating uniform hashing in constant time and optimal space. Research Series RS-02-27, BRICS, 2002.

[Ove83] Mark H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer-Verlag, 1983.

[OvL81a] Mark H. Overmars and Jan van Leeuwen. Dynamization of decomposable searching problems yielding good worst-case bounds. In *Proceedings of the 5th GI-Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 224–233. Springer-Verlag, 1981.

[OvL81b] Mark H. Overmars and Jan van Leeuwen. Worst-case optimal insertion and deletion methods for decomposable searching problems. *Inform. Process. Lett.*, 12(4):168–173, 1981.

[Pag99] Rasmus Pagh. Hash and Displace: Efficient Evaluation of Minimal Perfect Hash Functions. In *Proceedings of the 6th international Workshop on Algorithms and Data Structures (WADS '99)*, volume 1663 of *Lecture Notes in Computer Science*, pages 49–54. Springer-Verlag, 1999.

[Pag00a] Rasmus Pagh. Faster Deterministic Dictionaries. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '00)*, pages 487–493. ACM Press, 2000.

[Pag00b] Rasmus Pagh. Dispersing Hash Functions. In *Proceedings of the 4th International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM '00)*, volume 8 of *Proceedings in Informatics*, pages 53–67. Carleton Scientific, 2000.

[Pag00c] Rasmus Pagh. A trade-off for worst-case efficient dictionaries. *Nordic J. Comput.*, 7(3):151–163, 2000.

[Pag01a] Rasmus Pagh. On the Cell Probe Complexity of Membership and Perfect Hashing. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC '01)*, pages 425–432. ACM Press, 2001.

[Pag01b] Rasmus Pagh. Low redundancy in static dictionaries with constant query time. *SIAM J. Comput.*, 31(2):353–363, 2001.

[PM89] Patricio V. Poblete and J. Ian Munro. Last-come-first-served hashing. *J. Algorithms*, 10(2):228–248, 1989.

[PR01a] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *Proceedings of the 9th European Symposium on Algorithms (ESA '01)*, volume 2161 of *Lecture Notes in Computer Science*, pages 121–133. Springer-Verlag, 2001.

[PR01b] Rasmus Pagh and Flemming Friche Rodler. Lossy dictionaries. In *Proceedings of the 9th European Symposium on Algorithms (ESA '01)*, volume 2161 of *Lecture Notes in Computer Science*, pages 300–311. Springer-Verlag, 2001.

[Rad92] Jaikumar Radhakrishnan. Improved bounds for covering complete uniform hypergraphs. *Inform. Process. Lett.*, 41(4):203–207, 1992.

[Ram96] Rajeev Raman. Priority queues: Small, monotone and trans-dichotomous. In *Proceedings of the 4th European Symposium on Algorithms (ESA '96)*, volume 1136 of *Lecture Notes in Computer Science*, pages 121–137. Springer-Verlag, 1996.

[Riv78] Ronald L. Rivest. Optimal arrangement of keys in a hash table. *J. Assoc. Comput. Mach.*, 25(2):200–209, 1978.

[RR99] Venkatesh Raman and S. Srivinasa Rao. Static dictionaries supporting rank. In *Proceedings of the 10th International Symposium on Algorithms And Computation (ISAAC '99)*, volume 1741 of *Lecture Notes in Computer Science*, pages 18–26. Springer-Verlag, 1999.

[RRR01] Jaikumar Radhakrishnan, Venkatesh Raman, and S. Srinivasa Rao. Explicit deterministic constructions for membership in the bitprobe model. *Lecture Notes in Computer Science*, 2161:290–299, 2001.

[RRR02] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding $k$-ary trees and multisets. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '02)*, pages 233–242. ACM Press, 2002.

[RRV99a] Ran Raz, Omer Reingold, and Salil Vadhan. Error reduction for extractors. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS '99)*, pages 191–201. IEEE Comput. Soc. Press, 1999.

[RRV99b] Ran Raz, Omer Reingold, and Salil Vadhan. Extracting all the randomness and reducing the error in Trevisan's extractors. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing (STOC '99)*, pages 149–158. ACM Press, 1999.

[Sar80] Dilip V. Sarwate. A note on: "Universal classes of hash functions" [J. Comput. System Sci. **18** (1979), no. 2, 143–154; MR 80f:68110a] by J. L. Carter and M. N. Wegman. *Inform. Process. Lett.*, 10(1):41–45, 1980.

[Sie89] Alan Siegel. On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS '89)*, pages 20–25. IEEE Comput. Soc. Press, 1989.

[Sie95] Alan Siegel. On universal classes of extremely random constant time hash functions and their time-space tradeoff. Technical Report TR1995-684, New York University, 1995.

[Sil98] Craig Silverstein. A practical perfect hashing algorithm. Manuscript, 1998.

[SS89] Jeanette P. Schmidt and Alan Siegel. On aspects of universality and performance for closed hashing (extended abstract). In

*Proceedings of the 21st Annual ACM Symposium on Theory of Computing (STOC '89)*, pages 355–366. ACM Press, 1989.

[SS90a] Jeanette P. Schmidt and Alan Siegel. The analysis of closed hashing under limited randomness (extended abstract). In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing (STOC '90)*, pages 224–234. ACM Press, 1990.

[SS90b] Jeanette P. Schmidt and Alan Siegel. The spatial complexity of oblivious $k$-probe hash functions. *SIAM J. Comput.*, 19(5):775–786, 1990.

[SSS93] Jeanette P. Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff-Hoeffding bounds for applications with limited independence. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '93)*, pages 331–340. ACM Press, 1993.

[Sun93] Rajamani Sundar. A lower bound on the cell probe complexity of the dictionary problem. Manuscript, 1993.

[SV01] Peter Sanders and Berthold Vöcking, 2001. Personal communication.

[Tho00] Mikkel Thorup. Even strongly universal hashing is pretty fast. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '00)*, pages 496–497. ACM Press, 2000.

[TS] Amnon Ta-Shma. Storing information with extractors. To appear in Information Processing Letters.

[TSUZ01] Amnon Ta-Shma, Christopher Umans, and David Zuckerman. Loss-less condensers, unbalanced expanders, and extractors. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC '01)*, pages 143–152. ACM Press, 2001.

[TY79] Robert E. Tarjan and Andrew C.-C. Yao. Storing a sparse table. *Communications of the ACM*, 22(11):606–611, 1979.

[vEB75] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science (FOCS '75)*, pages 75–84. IEEE Comput. Soc. Press, 1975.

[Vöc99] Berthold Vöcking. How asymmetry helps load balancing. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS '99)*, pages 131–141. IEEE Comput. Soc. Press, 1999.

[WC81] Mark N. Wegman and J. Lawrence Carter. New hash functions and their use in authentication and set equality. *J. Comput. System Sci.*, 22(3):265–279, 1981.

[Wen92] Michael Wenzel. Wörterbücher für ein beschränktes Universum. Diplomarbeit, Fachbereich Informatik, Universität des Saarlandes, 1992.

[Wil83] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Inform. Process. Lett.*, 17(2):81–84, 1983.

[Wil00] Dan E. Willard. Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion tree. *SIAM J. Comput.*, 29(3):1030–1049, 2000.

[Woe99] Philipp Woelfel. Efficient strongly universal and optimally universal hashing. In *Proceedings of the 24th Mathematical Foundations of Computer Sciences*, volume 1672 of *Lecture Notes in Computer Science*, pages 262–272. Springer, 1999.

[Yao81] Andrew C.-C. Yao. Should tables be sorted? *J. Assoc. Comput. Mach.*, 28(3):615–628, 1981.

[Yao85] Andrew C.-C. Yao. Uniform hashing is optimal. *J. Assoc. Comput. Mach.*, 32(3):687–693, 1985.

## Recent BRICS Dissertation Series Publications

**DS-02-5** **Rasmus Pagh.** ***Hashing, Randomness and Dictionaries*****. October 2002. PhD thesis. x+167 pp.**

**DS-02-4** **Anders Møller.** ***Program Verification with Monadic Second-Order Logic & Languages for Web Service Development*****. September 2002. PhD thesis. xvi+337 pp.**

**DS-02-3** **Riko Jacob.** ***Dynamic Planar Convex hull*****. May 2002. PhD thesis. xiv+110 pp.**

**DS-02-2** **Stefan Dantchev.** ***On Resolution Complexity of Matching Principles*****. May 2002. PhD thesis. xii+70 pp.**

**DS-02-1** **M. Oliver Möller.** ***Structure and Hierarchy in Real-Time Systems*****. April 2002. PhD thesis. xvi+228 pp.**

**DS-01-10** **Mikkel T. Jensen.** ***Robust and Flexible Scheduling with Evolutionary Computation*****. November 2001. PhD thesis. xii+299 pp.**

**DS-01-9** **Flemming Friche Rodler.** ***Compression with Fast Random Access*****. November 2001. PhD thesis. xiv+124 pp.**

**DS-01-8** **Niels Damgaard.** ***Using Theory to Make Better Tools*****. October 2001. PhD thesis.**

**DS-01-7** **Lasse R. Nielsen.** ***A Study of Defunctionalization and Continuation-Passing Style*****. August 2001. PhD thesis. iv+280 pp.**

**DS-01-6** **Bernd Grobauer.** ***Topics in Semantics-based Program Manipulation*****. August 2001. PhD thesis. ii+x+186 pp.**

**DS-01-5** **Daniel Damian.** ***On Static and Dynamic Control-Flow Information in Program Analysis and Transformation*****. August 2001. PhD thesis. xii+111 pp.**

**DS-01-4** **Morten Rhiger.** ***Higher-Order Program Generation.*** **August 2001. PhD thesis. xiv+144 pp.**

**DS-01-3** **Thomas S. Hune.** ***Analyzing Real-Time Systems: Theory and Tools.*** **March 2001. PhD thesis. xii+265 pp.**

**DS-01-2** **Jakob Pagter.** ***Time-Space Trade-Offs*****. March 2001. PhD thesis. xii+83 pp.**

**DS-01-1** **Stefan Dziembowski.** ***Multiparty Computations — Information-Theoretically Secure Against an Adaptive Adversary*****. January 2001. PhD thesis. 109 pp.**