Implicit Storage Schemes for Quick Retrieval

By

Simeon Naor

B.A. Technion (Israel Institute of Technology) 1985

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA at BERKELEY

Approved: Manuel Blum April 27, 1989
Chair Date
Richard M. Karp April 27, 1989
David Hodes April 27, 89

****

Implicit Storage Schemes for Quick Retrieval

Moni Naor

Abstract

We address problems of the form: given a collection of n records, each composed of k keys, arrange them in a table of size n so that locating a record with a given key value can be performed while accessing the table as few times as possible. We analyze the amount of additional memory required to achieve optimal search time in a few settings. Schemes that do not use any additional memory are called *implicit.*

When comparisons are the only basic operation performed on the records, the table can be arranged with no additional memory, so that searching can be performed in optimal logarithmic time. The preprocessing time is also optimal, $O(n \log n)$.

When the operations are not restricted to comparisons, e.g. hashing, the problem is analyzed with respect to the size of the domain from which the keys are drawn. It is shown that if the domain is of size at most exponential in the number of records, then there exists an implicit scheme which achieves searching in worst case $O(1)$ probes to the table. Conversely, lower bounds on the domain size for which no implicit $O(1)$ probe search schemes exist are given. These lower bounds improve the previously known bounds. The problem is considered both for the single-key case and the multi-key case. The general conclusion is that the two cases behave in a similar fashion.

The upper bounds for both the comparison and the unrestricted cases are in contrast to lower bounds that appeared in the literature for these models. The seeming contradiction is resolved by observing that our models allow the search algorithm to use information which is not directly related to the value being searched. This points out the delicacy in choosing an appropriate model.

The solutions are achieved by techniques that enable representation of up to $O(n \log n)$ arbitrary bits implicitly, by exploiting the order of the records, so that a word of length $O(\log n)$ bits can be retrieved by accessing the table a constant number of times. Thus, the general methodology in designing the schemes is as follows: first, give a space efficient algorithm that uses some additional memory, then show how to encode the additional memory implicitly.

To the memory of my father

לאבי ז"ל

Acknowledgements

It is a pleasure to thank all the people who have helped make this thesis possible.

My advisor Manuel Blum has been a constant source of enthusiasm, inspiration and encouragement. He gave me the freedom to pursue my research in diverse directions and I am grateful for that. By his example he has shown me how non traditional thinking can always reveal new facets even in the most familiar topic. Dick Karp will always be my symbol of intellectual integrity. I have benefited enormously from the classes he taught. My meetings with him always gave me new perspectives and new research directions. I wish to thank Dorit Hochbaum for her diligent reading of the thesis and her encouragement throughout my years at Berkeley. To all three I thank for serving on my Thesis and Qualifying Exam Committees.

Umesh Vazirani always had an open door and mind to listen to my ideas and helped crystallize many of them. I also wish to thank him for serving on my Qualifying Exam Committee.

Amos Fiat has been a true friend and a colleague. All the results in this thesis are joint with Amos. His profound thinking has had a lasting influence on my own. Alex Schaffer taught me what it means to be a true scholar. I have enjoyed working with him on searching and storing multi-key tables and other topics. I wish to thank Jeanette Schmidt and Alan Siegel for their cooperation on non-oblivious hashing.

Avi Wigderson has been very supportive during my crucial first steps at Berkeley. An early exposure to a researcher of such calibre was the best stimulator I could have hoped for.

Thanks to my other teachers at Berkeley, Gene Lawler, Raimund Seidel and Mike Sipser I discovered many different perceptions of the field.

I wish to thank my fellow students for making Berkeley such a unique place. During my years in Berkeley I had the opportunity to collaborate with some of them: Russell Impagliazzo, Sandy Irani, Sampath Kannan, Noam Nisan, Ronitt Rubinfeld and Steven Rudich. I have learned a lot from the experience.

I am lucky to have two family members in the trade. The collaboration with my brother Seffi is of a very special type. This is a unique relationship that only few people are fortunate to have. Finally, I thank my wife Dalit, without whom I doubt if I would have survived.

Contents

1 Introduction **1**

1.1 Problem Definition, Background and Results 4

1.1.1 Comparisons 4

1.1.2 Unrestricted Case 5

1.2 Techniques 8

1.3 organization 10

2 Rainbows **11**

2.1 Rainbows Provide Virtual Memory 12

2.2 Rainbow Construction 14

2.3 Bounds on Rainbows 19

2.4 Relation to Other Constructions 21

3 Many Good Orders and Cycle Chasing **25**

3.1 Permutations Supporting $O(1)$ Search 25

3.2 Cycle Chasing 28

4 Comparisons **32**

4.1 Finding a Good Sample 33

4.2 The Search 36

4.3 Encoding 39

4.4 Application to Memory Efficient Data Structures 41

5 Unrestricted Search **43**

5.1 Perfect Hash Functions 45

5.2 Implicit $O(1)$ Probe Search 46

5.2.1 The Scheme 47

5.2.2 Implicit $O(1)$ Probe Search Yields Rainbows 48

5.3 Coping with nonexistence of implicit schemes 50

5.4 Multi-key Tables and Successor Computation 52

5.4.1 Multi-key Tables 53

5.4.2 Successor Computation and Sorted Array Emulation 54

6 Further Research **57**

Chapter 1

Introduction

The search for a record with a given key value in a large collection of records is one of the most basic problems computer science addresses. In this thesis we try to determine what are the minimal resources that allow quick search.

The problems we consider are of the general form: a set of n records, each composed of k keys, is to be arranged in a table of size n, with one record per entry. Given a key value, a record with that value should be located in the table by accessing it as few times as possible. A typical solution to this problem consists of two parts: an arrangement scheme of the table, and a search algorithm for a record in the table that is arranged as suggested. The implementation of such a solution therefore consists of two stages: the *preprocessing stage* that arranges the table and the *search stage*.

A solution is characterized by the (worst case) run time of a search. If memory is for free, then usually the task is trivial. For instance, if each record consists of a single-key and the keys of the n records are actually a subset $S \subset \{1 \ldots m\}$ of size n, then an array A of size m such that $A[i] = 1$ if and only if $i \in S$ would yield an $O(1)$ search. Therefore, we analyze the *minimal* space requirements needed to

achieve quick search time.

We shall aim at achieving the best search time possible within the model, without using any additional memory apart from the table, or if that is not possible, we try to minimize the size of the additional memory. Schemes that do not use additional memory are called *implicit.*

Two reasons motivate us to investigate problems of this type. One is from a space-efficiency or data-compression point of view: fast access memory is a critical resource in many systems, there never seems to be enough of it. Also often its size is predetermined, and cannot be controlled by the designer. Hence, it is important to know what can be done with whatever resources we have. Thus, problems of this type have practical significance, and determining the space requirements is highly desirable.

The second and perhaps more important reason from our point of view is that in order to understand the inherent limitation of computation, we must realize how the different components of a system can pass information they have gained to other components. We can think of the preprocessing stage as trying to signal the search stage how to conduct quick search in the specific collection of records we have. The part of the memory that does not store the records, what we call the *additional memory*, is the obvious mean of passing this information. Indeed, most previous researchers had concentrated their efforts on this way of passing information. However, from our results it follows that most or all of the search information can be coded in the section of the memory that stores the records, by the relative order of the records. We therefore point out a neglected source of information flow in a system.

The general methodology we apply for designing implicit schemes is as follows: first, we suggest a scheme that uses a considerable amount of additional storage,

but less than the naive one; then we show how this additional memory can be represented implicitly. As this memory emulation provides us with the ability to store desired information without the need for physical memory, we call it a *virtual memory.*

The objective of the problem, as stated above, is to minimize the number of probes to the table. Thus, the complexity of coordinating the search, i.e. deciding which locations to probe, is not taken into account. We shall try not to take advantage of this hospitality and be as efficient as we can in that respect as well. Unless stated otherwise, all the results we describe are constructive, i.e. true algorithms that achieve the time bounds not only with respect to the number of probes.

In this thesis we consider a few variations of the general problem. To characterize a problem of the general format defined above, we have to specify the type of operation allowed in the search, and whether a record has a single-key or multiple keys. The operations on the records could be restricted to comparisons, i.e. in each step either two records are compared or a record is compared against the value being searched for, or they could be unrestricted, in which case the search for x can be tailored according to the specific value of x and the entries being probed. Hashing is the best known example of unrestricted search.

Another type of specification is the number of keys: records can either possess a single-key or be composed of many keys. Throughout the thesis we consider both versions. A somewhat surprising consequence of our results is that the multi-key case behaves essentially the same as the single-key case. The conditions that allow one-key search are similar to those that allow multi-key search.

The solution presented for the comparison-based multi-key table problem was developed with Amos Fiat and Alex Schäffer, and appeared in [FNSSS] and [FMNSSS]. The solution for the unrestricted search case was obtained in two stages: we first

gave an upper bound on the size of the additional memory enabling search in $O(1)$ time; this was done with Amos Fiat, Jeanette Schmidt and Alan Siegel, and appeared in [FNSS]. Rainbows and their connections to implicit $O(1)$ probe search were introduced later, and are a joint work with Amos Fiat [FN].

We start by giving a precise definition of the problems, background and a review of the literature related to the problems studied in the thesis along with a brief statement of the results. Then we give an overview of the techniques which are common to all the solutions, and finally a description of the content of the following chapters.

1.1 Problem Definition, Background and Results

1.1.1 Comparisons

In the comparison-based case, a collection of size n of k-key records is given. The records are to be arranged in a table of size n so that searching under any key can be executed quickly; no additional memory is allowed. The only operation allowed on a record is comparing two records according to the ith key, $1 \leq i \leq k$, or comparing the value being searched for against a record. A simple information theoretic argument implies that the best one can hope for is $\log n$ comparisons. If $k = 1$, i.e. there is single-key, life is easy: the table is arranged by sorting it according to the key and in order to locate a record, a binary search is conducted. Hence the more interesting case is when $k \geq 2$.

The problem was first considered formally by Munro in [Mun79]. The search time bounds achieved there are $O(n^{1-1/k})$. This time bound was conjectured to be the best possible. The conjecture was supported by a lower bound of $\Omega(n^{1-1/k})$ of

Alt Mehlorn and Munro [AMM], in case it is required that during the search all the comparisons are between the value being searched for and one of the records, i.e. no comparison between two records in the table is allowed. While this assumption might not seem restrictive, Munro in [Mun87] showed that $O(\log^k n \log\log^{k-1} n)$ search time is possible. His search algorithm does compare two records in the table. Our solution uses many techniques that were introduced in this paper.

We have completely resolved the comparison-based case. The records can be arranged so that searching under any key requires $O(\log n)$ comparisons. The preprocessing requires $O(n \log n)$ time, which is the best possible. The search itself does not involve any computation which cannot be performed within the claimed time bounds, i.e. deciding which keys to compare can be executed efficiently. The key values need not be distinct and all we have to assume is that no two records are identical. The keys can in fact intersect, i.e. be composed of overlapping fields in the record.

One appealing property of our solution is for $k = 1$, i.e. the single-key case, the solution defines the organization as a sorted array and the search as binary search. Thus, the claim that this is the k-dimensional generalization of binary search is justified.

1.1.2 Unrestricted Case

In the unrestricted case, where every possible operation can be performed on a key value, even the optimal solution for the single-key problem is not clear. We rephrase the one-key problem in this context. A set $S \subset \{1 \ldots m\}$ of size n is to be stored in a table T of size n where every table entry holds a single element of S. Given $x \in \{1 \ldots m\}$, the goal is to locate x in the table as quickly as possible.

Yao [Yao] has shown that if no storage is available in addition to the table T, then there is no table organization that enables an element to be located in less than $\log n$ probes. We refer to a table organization that requires no additional storage as an implicit scheme. Yao's proof assumes that the domain size m is much larger than the number of elements n. This immediately raises the following questions:

1. For what values of m (as a function of n) does an implicit $O(1)$ probe search scheme exist?

2. Given that an implicit scheme does not exist, how much additional storage is required to ensure $O(1)$ search?

3. If the set S is chosen uniformly at random, or if access to a random hash function is available, does an implicit $O(1)$ probe scheme exist with high probability?

As for the first question, what was known is that an implicit $O(1)$ probe search scheme exists when the domain size m is less than $2n - 2$ where n is the set size [Yao].

We can answer for most functions f , when $m = f(n)$, whether there is an implicit $O(1)$ probe search scheme for m and n. If m is bounded by $2^{p(n)}$ for some polynomial p, then an implicit $O(1)$ probe search scheme exists. However, for that range we know of only a non-constructive scheme that is a based on a probabilistic construction, yet we have a concrete scheme when m is bounded by a polynomial in n. We also give a new proof of Yao's theorem that yields better lower bounds than Yao's: If m as a function of n grows quicker than a tower of powers of 2 of constant height, i.e.

$$m \geq 2^{2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}}} \Big\}O(1).$$

then no implicit $O(1)$ probe scheme exists.

Many researchers, among them Tarjan and Yao [TY], Yao [Yao], and Fredman, Komlós, and Szemerédi [FKS] used additional memory to achieve an $O(1)$ probe search. In particular, [FKS] describe how to generate a *perfect* hash function, which maps every key in S to a unique location in the table; the function's description requires a substantial amount of additional memory. The number of bits in the additional memory required to implement the method of [FKS] is $O(n\sqrt{\log n} + \log\log m)$.

An $\Omega(n + \log\log m)$ lower bound on the number of extra bits required for perfect hashing functions ([Meh84] Chapter 3, Theorem. 6, [Mai83], [FKS], [FK], [BBDOP]) is known.

Our solution uses considerably less storage than these bounds. In the cases where there is no implicit $O(1)$ probe search (or we do not know of one) an additional memory of size $O(\log n + \log\log m)$ bits suffices to achieve $O(1)$ probe search. This does not contradict the lower bound mentioned above, since that lower bound applies only to the special case of a perfect hash scheme.

Whenever an access to a random hash function is available "for free", no additional memory is required to achieve $O(1)$ probe search in the worst case, with high probability.

The multi-key case: In the case where we have several keys the same questions can be asked. Our results are similar. Whenever a solution is available in the single-key case, it is available in the multi-key case as well. No nontrivial upper bounds appeared in the literature before.

We also consider the question of finding, in case the value being searched for is not in the set, the closest element to the value. Ajtai [Ajt] showed that it is impossi-

ble to achieve $O(1)$ probes without a substantial amount of additional memory. We give logarithmic time bounds without additional memory. Since in a sorted table this kind of problem is easily solved, we call it sorted array emulation.

Mairson [Mai84] introduced a quantity which he calls the *program complexity*. The preprocessing can be regarded as producing a program for searching the specific collection of records. Intuitively, this program is assumed to reside in the additional memory. Thus, the size of the additional storage required to achieve an $O(1)$ probe search in the single-key case is treated as the bound on the program complexity. From our results it follows that this intuition is wrong. Most or all of the program complexity is absorbed in the order of the records and does not necessarily have to reside in the additional memory.

1.2 Techniques

In this section we shall try to gain some insight into the techniques that achieve our results.

A major source of power common to all of our search algorithms is the fact that they make use of information found in the table other than the information directly related to the value being searched for. How can we gain anything from "unrelated information"? Below, three possible techniques that make use of "unrelated information" are described

Virtual Memory: In order to emulate additional memory, we will let the records play an active role even when not searched for, instead of just lying around waiting for the day they will be searched for. With any sequence of t records we associate a color, which encodes some memory word we wish to preserve. To assure that all bit patterns of the word are possible we define and construct *rainbows*. A rainbow is a

coloring of sequences (without repetitions) of length t over a certain domain, so that for every large enough subset of that domain and every color there is a sequence of elements from the subset that is colored with that color. Prior techniques allowed representation of $O(n)$ bits. They were based on Munro's even-odd encoding - the order of two adjacent records determines a bit. Our rainbows allow representation of up to $O(n \log n)$ bits.

Flexibility of order: A fundamental observation due to Paul Feldman (appeared in [BFMUV]), without which none of results of this thesis would have been possible, is that there are many orders under which quick search is possible. In his scenario, the single-key comparison-based case, the order of the completely sorted array is not the only permutation that allows quick search. A table could be in one of a set of permutations of size $(\frac{n}{4})!$ and yet be searched in logarithmic time. This flexibility has been used by [Mun87] for the comparison case. We use it in conjunction with the rainbows, so as not to have a degradation in the search performance time, yet save memory by applying the rainbow technique for virtual memory.

Cycle Chasing: Another useful tool is the *cycle chasing* technique. For clarity, we will introduce the problem (which the cycle chasing technique is meant to solve) by describing an application of it in a context that slightly differs from the other problems in this thesis. Suppose there is an imaginary city where one night all the wives move to live with different husbands (not their own). By the pigeon hole principle, each husband receives a new wife. Suppose furthermore that one of the husbands wants to locate his original wife. He can do so by going to his new wife's previous house, query the new wife there about her former residence, and continue in such a manner until he finds his former wife. This might take very long. To help such husbands, we would like to put hints on their way to shorten the tour. Given that we can put less than n hints, is there anything useful we can do? In fact we

can do a lot. Given $c \cdot n$ hints, for any fixed $c < 1$, we can cut down the chase time of each husband to a constant (depending on c).

Chapters 2 and 3 describe these techniques, and chapters 4 and 5 show how to apply them for the problems at hand.

1.3 organization

The rest of the thesis is organized as follows:

- Chapter 2 defines, motivates, constructs and bounds rainbows
- Chapter 3 discusses the flexibility in ordering and the cycle chasing technique
- Chapter 4 solves the comparison-based case
- Chapter 5 deals with the unrestricted case
- Chapter 6 suggests further research.

Chapter 2

Rainbows

This chapter introduces a combinatorial structure that will turn out to have important applications for emulating a storage device - the *rainbow*.

Definition:

A (c, m, n, t)-*rainbow* is a coloring of all t-sequences (without repetitions) over $\{1 \ldots m\}$ with c colors so that for any set $S \subset \{1 \ldots m\}$, $|S| = n$, all c colors appear in the t-sequences over S.

Two simple examples of rainbows are:

Example 1: For any m, n and $t \leq n$ we can construct a $(c = t!, m, n, t)$-rainbow by associating a color with each of the $t!$ permutations. A t-sequence is colored with the color associated with the permutation on the natural order it defines.

Example 2: For any n, a $(c = n, m = n, n, t = 1)$-rainbow can be constructed. The color a sequence receives is just the value of its single member.

For the applications we have in mind, we are interested mainly in the case where $c = n$ and t is a constant. As we shall see, when the number of colors required is

larger than $t!$, than the existence of a (c, m, n, t)-rainbow depends on the relationship between m and n.

From their definition it is apparent that rainbows are related to Ramsey theory. Indeed, the impossibility results we have are derived from Ramsey theory, and are expressed in terms of Ramsey numbers. We also use constructions from Ramsey theory to reduce the problem of constructing rainbows to similar structures where the order does not matter.

Rainbows are used to emulate additional storage both in the comparison-based search and in the unrestricted search. However, in the latter case they are connected in the other direction as well. We will show in chapter 5 that given an implicit $O(1)$ probe search scheme, we can construct from it a rainbow with certain parameters. Thus we will be able to apply the impossibility results of rainbows to implicit schemes.

We first motivate rainbows, by showing how they can be used to provide virtual memory. The goal of Section 2.2 is to construct rainbows. An explicit construction of a (c, m, n, t)-rainbow when m is polynomial in n, $c = n$ and t is a constant is given and a probabilistic construction is shown to be good even when m is exponential in n. Section 2.3 deals with impossibility of certain rainbows; if m is much larger than n then such a construction is not possible. Finally, Section 2.4 briefly mentions relationship between rainbows and other combinatorial structure.

2.1 Rainbows Provide Virtual Memory

We now show how rainbows can be used to simulate additional memory. Consider the following:

Virtual Memory Problem

Given:

- A set $R = \{r_1, r_2, \ldots, r_{n'}\}$, where $1 \leq r_j \leq m$ for $1 \leq j \leq n'$.
- A series of values $v_1, v_2, \ldots, v_l$ where $0 \leq v_i \leq n-1$ for $1 \leq i \leq l$.

Arrange the elements of R in an array A of size n' (put each element of R in a different location) so that given $1 \leq j \leq l$, v_j can be reconstructed (decoded) quickly, via t accesses to A, where t is such that $n' - t(l-1) \geq n$.

Note that we do not require anything about locating elements of R, only that they will reside somewhere in A.

The next lemma shows the relationship between this problem and the existence of rainbows.

Lemma 2.1 *Given a $(c = n, m, n, t)$-rainbow $\mathcal{C}$, the virtual memory problem described above can be solved.*

Proof: Divide the first $t \cdot l$ locations of the array A into blocks of size t. The elements of R should be arranged in A so that the color assigned by $\mathcal{C}$ to the jth block, i.e. to the sequence $\langle A[jt+1], \ldots A[(j+1)t] \rangle$, is v_j. To achieve that, a greedy algorithm can be applied:

Greedy Encoding

- Set $U = R$
- For $j = 1$ to l
  - Find a sequence s colored v_j in U

- Put sequence s in order in the jth block of A
- $U \leftarrow U \setminus s$

- Arrange U in the rest of A arbitrarily

Throughout the execution of the loop the number of elements in U is $n' - jt \geq n$. Hence there is a sequence in U colored by $\mathcal{C}$, and the find step in the algorithm always succeeds.

This arrangement means that in order to reconstruct v_j, one has to determine the color of the jth block under $\mathcal{C}$ and this can be done via t accesses to A. This method is constructive if given a sequence its color under $\mathcal{C}$ can be determined effectively. ∎

A concrete example of the application of the greedy encoding using the rainbow of Lemma 2.2 is given following that lemma.

Throughout the thesis we say that $v_1, v_2, \ldots, v_l$ are in the virtual memory.

2.2 Rainbow Construction

This section provides an explicit construction of rainbows when the number of colors $c = n$ and the length of the sequence t is a constant. We start with a simple construction for the case $m \leq 2n-1$ (Lemma 2.2) which exemplifies the construction for a domain m which is quadratic in the number of elements n (Lemma 2.3). The ideas behind this construction are later used in showing how to reduce a problem with domain m to another problem with domain $\sqrt{m}$ (Lemma 2.4) . This yields an explicit recursive construction for all m which are polynomial in n (Theorem 2.1). We conclude the section by showing that a probabilistic construction is good even when m is exponential in n (Theorem 2.2).

Lemma 2.2 *For any n and $m \leq 2n-1$ there exists a $(c = m, m, n, 2)$-rainbow.*

Proof: color an ordered pair $\langle x_1, x_2 \rangle$ with the color $x_1 - x_2 \bmod m$. We have colored all the edges of the complete directed graph on m nodes. Let D_i be the directed subgraph (of the complete graph) induced by the edges colored i. For any color i, D_i is a collection of disjoint cycles, since addition mod$m$ is a group. For any set $S \subset \{1 \ldots m\}$ such that $|S| = n$ consider the subgraph of D_i induced by the elements of S. Any element missing from S can eliminate at most two edges from D_i. Since at most $n-1$ elements are missing from S, at least one edge in D_i survives. Therefore, for each color i there is a pair colored i in S. ∎

Example 3: We show how this $(m, 2n-1, n, 2)$-rainbow can be used for the purposes of virtual memory encoding explained in the previous section. The domain size $m = 24$, the set R contains $n' = 20$ elements; we wish to encode $l = 4$ values $v_1, \ldots, v_4$:

$$R = \{1, 2, 3, 4, 5, 6, 8, 10, 11, 12, 13, 14, 15, 17, 18, 19, 21, 22, 23, 24\},$$

$$v_1 = 16, \quad v_2 = 5, \quad v_3 = 19, \quad v_4 = 7.$$

The set R is ordered in the array A so that $A[2i-1] - A[2i] = v_i$, $1 \leq i \leq 4$:

$$A[1, \ldots, 20] = [17, 1, 19, 14, 21, 2, 18, 11, \ldots].$$

∎

Lemma 2.2 is sufficient to provide the encoding required in the comparison model, but not that for the unrestricted search case. For that we need better constructions.

Lemma 2.3 *For any prime p, there is an explicit construction of a $(c = n, m = p^2, n = p+1, t = 2)$-rainbow.*

Proof: Consider a 1-1 mapping from all elements $e \in \{1 \ldots m\}$ to pairs (x, y) such that $1 \leq x, y \leq p$. (For instance, $x = e \pmod p + 1$, $y = (e - x)/p + 1$.) Given an element in $\{1 \ldots m\}$, we will consider its value the value of the mapping.

Color the sequence $\langle u, v \rangle$, $u = (x_1, y_1)$, $v = (x_2, y_2)$, with the color $(y_2 - y_1)/(x_2 - x_1) \pmod p$. If $x_2 = x_1$ then color the sequence $\langle u, v \rangle$ with the color p. We have colored all edges of the full directed graph on m vertices. Note that the sequence $\langle u, v \rangle$ is colored as the sequence $\langle v, u \rangle$, hence we can consider the coloring as that of a complete undirected graph. To prove that this is a good coloring we need the following:

Claim 2.1 *Consider the edge induced subgraph G_i obtained by choosing all edges of color i. G_i consists of p disjoint cliques of size p.*

Proof: First, note that every vertex $u = (x, y)$ has exactly $p - 1$ directed edges $(u, v_j = (x_j, y_j))$ colored i, for all $0 \leq i \leq p$. For $i = p$ these are simply pairs (x, y_j), $y_j \neq y$; for $i < p$ the x_j and y_j values are the $p - 1$ solutions to the equation $(y_j - y)/(x_j - x) = i \bmod p$.

To show that the undirected induced subgraph consists of cliques, assume that the $\langle u, v \rangle$ and $\langle v, w \rangle$ sequences are colored i: then the $\langle u, w \rangle$ sequence must also be colored i. If $u = (x_1, y_1)$, $v = (x_2, y_2)$ and $w = (x_3, y_3)$ either $i = p$ in which case $y_1 = y_2 = y_3$ and (u, w) is also colored p or $i < p$ in which case $(y_2 - y_1)/(x_2 - x_1) = (y_3 - y_2)/(x_3 - x_2) = i \pmod p$. It now follows that $(y_3 - y_1)/(x_3 - x_1) = i \pmod p$. ∎

Remark: Note that all vertices u_j, $u_j = (x_j, y_j)$, belonging to the same clique in G_i, have the same value $y_j - ix_j \pmod p$. This means that we can identify the G_i clique containing a vertex u.

We can now resume the proof of the lemma:

Given a set $S \subset \{1 \ldots m\}$ of size $n = p+1$, at least two elements $u, v \in S$ belong to the same monochromatic clique colored i, for all $0 \leq i \leq p$. This means that both sequences $\langle u, v \rangle$ and $\langle v, u \rangle$ are colored i. ∎

To construct a rainbow for m polynomial in n we use a recursive construction. We explain how to use the construction above to transform the problem from a domain of size m to a domain of size $\sqrt{m}$, by concatenating two elements to each sequence in the $\sqrt{m}$ domain.

Lemma 2.4 *Given a construction of a $(c = n, m = p, n-2, t)$-rainbow where p is a prime, a $(c = n, p^2, n, t+2)$-rainbow can be constructed.*

Proof: Let $\mathcal{C}$ be a $(p+1, p^2, p+1, 2)$-rainbow as described in Lemma 2.3. Our inductive assumption is that we have an $(n, p, n-2, t)$-rainbow. Our goal is to construct an $(n, p^2, n, t+2)$-rainbow. Given a set $S \subset \{1 \ldots m\}$, $|S| = n$, if all $p+1$ colors appear in the 2-sequences over S under $\mathcal{C}$ then we are essentially done: (in fact, the rainbow contains more colors than required) the first two elements in the sequence will function as an indicator; if they are in decreasing order, the interpretation is that they are colored by $\mathcal{C}$ and the rest of the sequence should be ignored.

Otherwise, at least one color is missing, but there is at least one color that appears (we assume $n \geq 2$). Therefore, there is a color i such that no pair in S is colored by $\mathcal{C}$ with i, but there exist $u, v \in S$ such that (u, v) is colored $i-1$ under $\mathcal{C}$.

Consider G_i, the edge induced graph induced by edges colored i and introduced above. Every element in S is in a different clique of G_i, else there would have been a pair colored i. The cliques of G_i can easily be indexed as described by the remark at the end of Lemma 2.3. To obtain a domain of size $\sqrt{m}$ every element $v \in \{1 \ldots m\}$

is mapped to the index of its clique in G_i. If the first two elements u, v of the $(t+2)$-sequence are in increasing order it indicates that such a mapping should be applied. Let $i-1$ denote the color associated with (u,v) under $\mathcal{C}$, the mapping applied to the remainder of the sequence replaces every value $w \in \{1 \ldots m\}$ by its clique index in G_i. By our inductive assumption we have an $(n, p^2, n, t+2)$ rainbow. ∎

Since for any integer x there is a prime in $(x, 2x)$ we can apply Lemma 2.4 recursively, each time reducing the domain from m to $2\sqrt{m}$. Using Lemma 2.3 as the base case provides us for any $d \geq 1$ with an explicit construction of an $(c = n, m = n^d, n, 2\lceil \log d \rceil + \lceil \log \log d \rceil)$-rainbow. Thus we have

Theorem 2.1 *For any domain m polynomial in the set size n there exists an $(n, m, n, O(1))$-rainbow. Given a sequence, its color can be determined in $O(1)$ time assuming modular arithmetic on $O(\log m)$ bits in unit-time.* ∎

Remark: Note that the proof implies that the existence of rainbows is a robust property, meaning that if p_1, p_2, p_3 are polynomials, and m is as a function of n such that a $(c, m, n, O(1))$-rainbow exists, then a $(p_1(c), p_2(m), p_3(n), O(1))$-rainbow exists as well.

Probabilistic Construction We now turn to the probabilistic construction for m *exponential* in n. Suppose $m = 2^{n^\ell}$ and consider a random coloring with n colors of all $\ell + 2$ sequences over $\{1 \ldots m\}$. For a set $S \subset \{1 \ldots m\}$, $|S| = n$, the probability that a specific color is missing in the $\ell + 2$ sequences over S is less than

$$(1 - 1/n)^{n(n-1)\ldots(n-\ell-1)}.$$

There are n colors and $\binom{m}{n}$ sets, hence the probability that there exists a set and a color such that the color is missing over the set is less than

$$\binom{m}{n} \cdot n \cdot (1 - 1/n)^{n(n-1)\ldots(n-t-1)} \leq 2^{n^{t+1}} \cdot n \cdot e^{-n^{t+1}} \cdot e^{t^2} \ll 1.$$

Therefore we have:

Theorem 2.2 *For any domain m exponential in the set size n there exists an $(n, m, n, O(1))$-rainbow.* ∎

2.3 Bounds on Rainbows

In this section we give bounds on the maximum m, as a function of n and t, for which a $(c = n, m, n, t)$-rainbow. We will do that by showing the connection between rainbows and colorings of the t-uniform hypergraph. Consider coloring of all t-subsets (subsets of size t) of $\{1 \ldots m\}$ with c colors. Ramsey theory tells us that there exists a function $R(n, t, c)$ such that if $m > R(n, t, c)$ then for any coloring of the t-subsets of $\{1 \ldots m\}$ with c colors there exists a set $S \subset \{1 \ldots m\}$ of size n such that all the t-subsets over S are colored with the same color. (See the book by Graham, Rothschild and Spencer [GRS] for details on Ramsey theory.)

Theorem 2.3 *If there exists a (c, m, n, t)-rainbow and $c > t!$ then*

$$m \leq R(n, t, t! + 1)$$

Proof: Given a (c, m, n, t)-rainbow, we define a coloring of the t-subsets of $\{1 \ldots m\}$: for each subset $H \subset \{1 \ldots m\}$ of size t consider all possible orderings of H. Each of the $t!$ possible orderings receives a color in the rainbow. Since there are more

than $t!$ colors in the rainbow we know that there is a color i, $1 \leq i \leq t! + 1$ which none of the orderings receives. Color H with such an i. From Ramsey theory it follows that if $m > R(n, t, t! + 1)$ then there will be a set $S \subset \{1 \ldots m\}$ of size n such that all of S subsets of size t are colored with the same color i. Hence in the rainbow all the t-sequences of S were not colored i, and thus it is not a rainbow. ∎

How fast does $R(n, t, t! + 1)$ grow? Let the tower functions $h_i(x)$ be defined as $h_1(x) = x$ and $h_{i+1}(x) = 2^{h_i(x)}$ for $i \geq 1$. That is

$$h_i(x) = 2^{2^{2^{\cdot^{\cdot^{\cdot^{2^x}}}}}\}i}.$$

The stepping up lemma in [GRS] page 91 yields the following: $h_{j-1}(c_1 \cdot n^2) \leq R(n, j, 2) \leq h_j(c_2 \cdot n)$ for some fixed c_1 and c_2. By the method of the proof of Ramsey theorem, increasing the number of colors from 2 to $t! + 1$ does not add more than $\log t! + 1$ to the height, i.e. $R(n, t, t! + 1) < h_{t+\lceil \log(t!+1) \rceil}(c_2 n)$. Hence we can conclude that for a $(c \geq t! + 1, m, n, t = O(1))$-rainbow to exist we must have

$$m \leq 2^{2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}}\}O(1)}.$$

Undirected Rainbows: We now show that the existence of rainbows is closely related to that of undirected rainbows defined as follows: A (c, m, n, t)-*undirected rainbow* is a coloring of all t-subsets over $\{1 \ldots m\}$ with c colors so that for any set $S \subset \{1 \ldots m\}$, $|S| = n$, all c colors appear in the t-subsets over S.

Since the order itself in directed rainbows can determine $t!$ different colors, as in example 1 in the introduction to the chapter, we know that $(c = t!, m, n, t)$-rainbows exist for any m and n. However, by Ramsey theory, this is not true for undirected rainbows. On the other hand, the next theorem shows that in order to give bounds on the maximum m for which $(c = n, m, n, O(1))$-rainbows exist, it is enough to consider undirected rainbows.

Theorem 2.4 *For every t there exists a constant b_t, dependent upon t, such that a construction for a $(c = n, m, n, t)$-rainbow yields a construction for a $(c = n, m, \lceil \log(t!) \rceil \cdot n, b_t)$-undirected rainbow.*

Proof: The idea is to provide enough information in the subset so as to simulate a directed set. If in addition to a t-subset, $\lceil \log(t!) \rceil$ bits are provided to determine the order in the t-subset, then the color of the subset will be the color of the corresponding t-sequence in the $(c = n, m, n, t)$-rainbow.

The additional bits are obtained as follows: Sort the b_t-subset and divide it into $\lceil \log(t!) \rceil + 1$ consecutive subsets. The subset of the largest elements should be of size t, all the rest are equal in size. Let t' denote $(b_t - t)/\lceil \log(t!) \rceil$. b_t is selected so that there exists a $(2, n, m, t')$ undirected rainbow. Such a b_t exists because the bounds on from theorem 3 we know that $m < R(n, t, t! + 1)$ and thus $mh_{t''}(c_2 n)$ for some t'' depending only on t. Hence, from the lower bound on $R(n, j, 2)$ of the stepping up Lemma there exists a $(2, n, m, t')$-rainbow for $t' = t''$.

Each of the $\lceil \log(t!) \rceil$ subsets will supply one bit under its 2-coloring. ∎

2.4 Relation to Other Constructions

In this section we briefly mention some relationships between rainbows and other constructions.

In [Sip] Sipser defined a certain kind of expander, and gave a probabilistic construction for it. We will show how to use such expanders to obtain an implicit $O(1)$ probe scheme for values of m and n for which such an expander exists, namely when m is $n^{\log n}$. To achieve this, we first show how to construct a rainbow with $\log n$ colors for such m and n, and then show how to apply it with a Sipser expander to

get an $(c = n, m, n, O(1))$-rainbow for those m and n.

Lemma 2.5 *There exists an explicit construction for a* $(c = \log n, m, n, O(1))$-*rainbow, if* m *is* $n^{polylog(n)}$

Proof: For $1 \leq x \leq m$ let x_i denote the ith bit of x. Consider the coloring of pairs that assigns the pair (x, y) $\min_{1 \leq i \leq \log m} x_i \neq y_i$, i.e. the first bit in which x and y differ.

Claim 2.2 *In any set* $S \subset \{1 \ldots m\}$ *of size* n, *the pairs must be colored with at least* $\log n$ *different colors.*

To see that the claim is true consider organizing the elements of S in a trie, i.e. in a binary tree where each element appears as a leaf and its value is determined by the path from the root. If a node in the ith level of the trie has two children, then there is a pair $\langle x, y \rangle$, where x is a decedent of the left child and y a decedent of the right child, that is colored i. There must be at least $\log n$ levels in the trie in which there is a node with 2 children, since each level can at most double the number of nodes from the previous one and there are n leaves.

The claim shows that rather than having a set of size n out of a domain of size m, the problem can be reduced to that of a set of size $\log n$ from a domain of size $\log m$. If m is $n^{\text{polylog}(n)}$, then $\log m$ is polynomial in $\log n$, and hence the constructions of Theorem 2.1 can be applied to obtain the required rainbow. ∎

An (l, r, d, a, b)-expander is a bipartite graph with l nodes on the left side, each with degree d, r nodes on the right side with the property that every subset of a nodes in the left side is connected to at least b of the nodes of the right side.

[Sip] gives a probabilistic construction for an $(n^{\log n}, n, 2\log^2 n, n, n/2)$ -expander. Such expanders can be used to amplify rainbows, and construct $(c = n, m =$

$n^{\log n}, n, O(1))$-rainbows. An element in $\{1 \ldots m\}$ is considered as a node on the left side of such an expander. Colors will correspond to nodes on the right side. The construction of a $(c = 2\log n, m, n, O(1))$-rainbow, described above, can be used to specify neighbors of a given node. Since a set of n nodes on the left side is adjacent to at least half the nodes on the right side, it follows that one can specify at least half of the nodes on the right side. Using the construction of Lemma 2.2 it can be amplified to all the colors. Therefore, an explicit construction for a Sipser expander would yield an implicit $O(1)$ probe search scheme for $m = n^{\log n}$.

No explicit construction with parameters close to the ones given in [Sip] is known. The best explicit construction for such expanders is given in Ajtai, Komlós Szemerédi [AKS].

Extracting Random Bits and Rainbows We finish this section by noting the relationship with extracting random bits from slightly random sources, as defined by Chor and Goldreich [CG]. Consider a source whose output $\in \{1 \ldots m\}$ but is actually chosen at random from an unknown set $S \subset \{1 \ldots m\}$ of size n. (This is a special case of the sources Chor and Goldreich considered.) Given a few such sources we wish to convert them to a quasi-random source, i.e. a source whose output distribution is close to uniform. (Quasi-random and semi-random sources are defined in Umesh Vazirani's thesis [Vaz]). Such a source is a slightly random source of entropy $\log n$. No constructive way is known to extract random bits at a constant rate when the domain size m is super-polynomial in n.

We claim that a constructive way to extract $\log n$ random bits from a fixed number of such sources would also define a $(c = n, m, n, O(1))$-rainbow. Suppose that the extractor can produce $O(\log n)$ random bits from t such source. Consider a coloring of t-sequences over $\{1, \ldots, m\}$ where the color assigned to a t-sequence is the output of the extractor in case the t sources output the values of the sequence.

Let the t sources be such that all output from the same set S. For any set S of size n and for most numbers $x \in \{1, \ldots, n\}$, there must be a combination of the output of the sources that is interpreted by the coloring as x, since otherwise there is a zero probability that the extractor outputs x for sources defined by the set S (and the output is therefore not quasi-random). This is not a good rainbow yet, but it can be converted to one by applying the construction of Lemma 2.2.

Chapter 3

Many Good Orders and Cycle Chasing

This chapter shows why the "natural order" is not the only one that is suitable for quick search (Section 3.1), and how husbands can locate their estranged wives with few hints (Section 3.2). The mathematical analogy of the last problem is, given a permutation π on $\{1,\ldots,n\}$ with the property that computing $\pi(i)$ is easy, find a way to compute π^{-1} without using much memory. We develop a technique called *cycle chasing* for computing $\pi^{-1}(i)$ efficiently that for any fixed $c > 0$ uses only $cn \log n$ bits. Those techniques will play an important role for both the comparison case and the hashing case.

3.1 Permutations Supporting $O(1)$ Search

Suppose that a set S such that $|S| = n$ is given, along with an oracle to a function f, $f : S \mapsto \{1,\ldots,n\}$ which is 1-1. (By an oracle we simply mean that given $x \in S$ there is some unspecified way to compute $f(x)$) There is a natural way to arrange

S in a table T of size n so that searching T can be done quickly: assign every $x \in S$ to $T[f(x)]$.

However, there are many ways in which the table can be perturbed and still support search in $O(1)$ time. We now show a set of $(\frac{n}{2})!$ permutations such that if the table is in one of them, searching the table (without knowing in which specific permutation it is) can still be conducted in $O(1)$ time.

Let $S_1 = \{x \mid f(x) \leq n/2\}$ and $S_2 = S \setminus S_1$.

Now suppose that S_1 is arranged in the first $n/2$ entries of T according to some arbitrary permutation τ of its natural order under f. If we arrange S_2 in the second half of T by applying τ^{-1} to its order under f, then searching for an element $x \in S$ can still be done in $O(1)$ time:

First note that both τ and τ^{-1} can be computed efficiently: for $1 \leq i \leq n/2$ $\tau(i) = f(T[i + n/2]) - n/2$; and $\tau^{-1}(i) = f(T[i])$.

To search for $1 \leq x \leq m$, compute $f(x)$ and probe for the record in its complementary location:

- If $f(x) \leq n/2$ then probe $T[\tau(f(x))]$.
- If $f(x) > n/2$ then probe $T[\tau^{-1}(f(x) - n/2) + n/2]$.

Since every permutation on the first half of the table is acceptable, there are $(\frac{n}{2})!$ permutations of T on which we can search.

Note that since τ and τ^{-1} can be computed efficiently, reconstructing the original order of T, i.e. computing $f^{-1}(i) \cap S$ can be accomplished by $O(1)$ accesses to T.

Example : Given a set $S = \{6, 9, 11, 15, 17, 19, 28, 32\}$ and a function f such that

$$f(17) = 1, \quad f(6) = 2, \quad f(11) = 3, \quad f(28) = 4,$$

$$f(19) = 5, \quad f(32) = 6, \quad f(15) = 7, \quad f(9) = 8.$$

the *natural* order of S in the table T under f is

$$T[1,\ldots,8] = [17,6,11,28,19,32,15,9].$$

We apply the permutation $\tau = (2,3,4,1)$ as described above, (*i.e.*, $\tau(1) = 2,\ldots,\tau(4) = 1$), $\tau^{-1} = (4,1,2,3)$. The table T now contains the values:

$$T[1,\ldots,8] = [28,17,6,11,32,15,9,19].$$

To search for the value 28 we compute $f(28) = 4$, as $4 \leq n/2$ we compute $\tau(4)$ by computing $f(T[n/2 + f(28)]) - n/2 = f(T[8]) - 4 = f(19) - 4 = 1$, in fact $T[1] = 28$. ∎

This technique is based on Feldman's construction in [BFMUW]. He showed that a single-key table in the comparisons-based case could be organized in one of $(\frac{n}{4})!$ permutations and yet can still be searched with $O(\log n)$ comparisons. The permutations are obtained by keeping the even ranked elements in the correct (original) position and pairing and swapping the odd ranked elements. We shall use a similar construction in Section 4.3.

We now mention briefly the context in which this technique will be used.

How can we exploit the freedom of ordering the first part of T? S_1 and the first half of T play the role of R and the array A in Lemma 2.1. S_1 would be permuted by some permutation τ of its natural order to provide "virtual memory" . By the technique presented above this does not preclude efficient search if a function f is available.

What will f be? In Chapter 4 $f(x)$ would be the (approximate) rank of x in S. In Chapter 5 $f(x)$ would be a perfect hash function.

3.2 Cycle Chasing

Suppose that a permutation π on $\{1 \ldots n\}$ is given by an oracle and we wish to be able to compute π^{-1}.

If preprocessing is allowed and additional memory of $n \log n$ bits divided into words of size $\log n$ is provided, then the task is easy. Write down $\pi^{-1}(i)$ in the ith memory word.

Suppose that instead of n words of additional memory only $c \cdot n$ are provided, for some $c < 1$. This section shows how to use this additional memory without causing a serious degradation in the time it takes to compute $\pi^{-1}(i)$.

If the cycles of π were of length $\leq h$, then $\pi^{-1}(i)$ can be evaluated by at most $h - 1$ forward mappings. This would be acceptable if h were guaranteed to be bounded by a constant. Unfortunately we can make no such claim. The cycle length h can be $\Theta(n)$.

When the cycles of π are long, we store strategic "shortcuts" that enable us to skip forward most of the way around the cycle in one step (see Figure 1). We declare that a cycle is *long* if it is longer than some constant, ℓ, which depends only on c. These shortcuts will require $\Theta(n)$ pointers, but the constant factor hidden in the Θ can be made as small as we wish by increasing ℓ, which is essential since the virtual memory the rainbows provide contains $c'n \log n$, for some $c' < 1$.

An arbitrary starting point p is chosen for each cycle, and starting from p, every ℓ^{th} element along the cycle is given a shortcut pointer to ℓ places *back* along the cycle. Thus, $\pi^{\ell}(p)$ "remembers" the index p, $\pi^{2\ell}(p)$ remembers the value $\pi^{\ell}(p)$, and so on. Location p remembers the value of $\pi^{\hat{h}}(p)$ where $\hat{h}$ is the largest multiple of ℓ strictly less than the cycle length h.

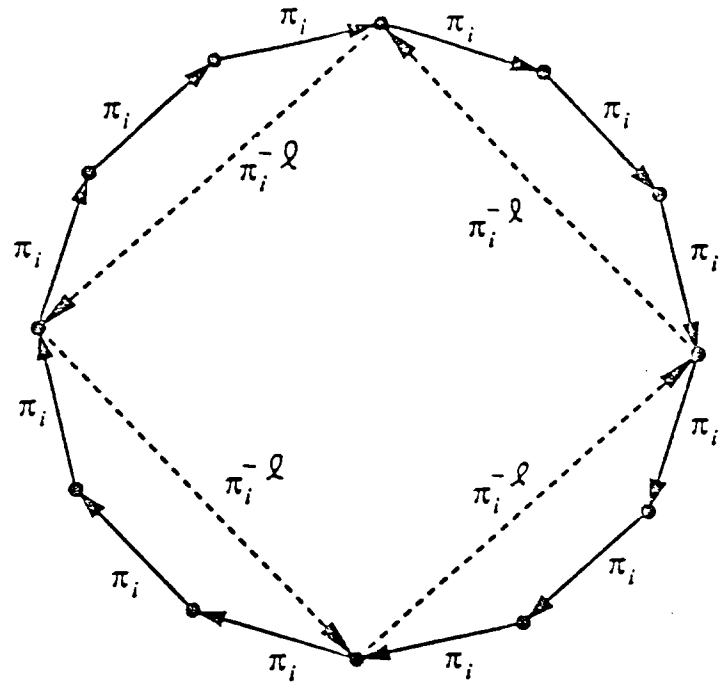

Figure 1. One can chase around a cycle of π going forward using a solid arrow or via a gateway using a dashed arrow.

These special locations, which are ℓ apart along the cycle, are called *gateways*. If $x \in \{1 \ldots n\}$ is a gateway, we use $P(x)$ to denote the location that x remembers. These gateways provide convenient paths to speed up the search at an aggregate cost of only "ϵn" pointers.

Let g be the number of gateways. Evidently, $g \leq \lfloor 2n/\ell \rfloor$. We are not done yet, since we must describe a way to know, when performing the cycle chasing, whether a gateway exists, and how to retrieve its value.

Number the gateways, x, associated with the permutation π (and our arbitrary choice of the p's above) with the values 1 through g, starting with the smallest gateway and in increasing order of x (regardless of which cycle they belong to).

A convenient way to store the values $P(x)$ is to use an array $Q[1..n]$ where $Q[x] = P(x)$ if x is a gateway and $Q[x] = 0$ otherwise. However, this violates our requirement that less than $n \log n$ bits would be used.

We now show how the array Q can be represented in a compact way. To obtain a space-efficient method use a bitmap $B[1..n]$ where $B[x] = 1$ if and only if x is a gateway, and an array $G[1..g]$ compressed to contain only the nonzero values of $Q[x]$ (in the order they would appear Q). To find the location of $Q(x)$ in G we use an array of counters, $C[0..\lfloor n/\lceil \log n \rceil \rfloor]$; each entry of C is a number between 0 and g inclusive. The entry $C[j]$ is the partial sum

$$\sum_{1 \le m \le j\lceil \log n \rceil} B[m].$$

It counts the number of gateways in the first $j\lceil \log n \rceil$ locations, If $B[x] = 1$ then $Q(x) = G[c]$ for some c. The index c can be computed as the sum of one C entry and at most $\log n$ consecutive B entries.

In order to compute this sum quickly we add another array D, such that for $1 \le i \le n$ if $d = \lfloor i/\lceil \log n \rceil \rfloor$ then $D(i) = \sum_{d\lceil \log n \rceil < m \le i} B[m]$. $D(i) \le \log n$ and hence representing D requires $n \log\log n$ bits. This is the dominating factor of the space requirements of all the additional arrays we used for solving the sparse array representation.

This method for representing sparse tables is actually a special case of the solution presented in Tarjan and Yao [TY]. We summarize by

Lemma 3.1 *Let T be a table of n entries, with g nonempty entries. T can be represented by a table of g entries together with an array of $O(n \log\log n)$ bits so that given i, $T[i]$ can be determined in $O(1)$ time.*

To conclude the description of the search algorithm, we introduce PI-INVERSE, a function that computes $\pi^{-1}(j)$.

PI-INVERSE(j)

Set nextindex := j;

Repeat

1. index := nextindex;
2. if B[index] = 1 (index is a gateway) then

   set $d := \lfloor index/\lceil \log n \rceil \rfloor$ set $c := C[d] + D[index]$;

   index := $G[c]$.
3. Set nextindex := $\pi(index)$;

Until nextindex = j;

Return(index);

Since the effective cycle length is bounded by ℓ, the loop in PI-INVERSE is repeated ℓ times at most. For any constant c, ℓ can be chosen so that the total memory requirements do not exceed $c \cdot n \log n$. Each iteration requires one call for π and constant amount of work otherwise.

We therefore conclude :

Theorem 3.1 *Suppose we are given an oracle for a permutation π and memory of $c \cdot n$ words, $\log n$ bits each. The memory can be arranged so that, given i, computing $\pi^{-1}(i)$ can be done by $O(1)$ calls to π and $O(1)$ time for other computations.*

Chapter 4

Comparisons

This chapter addresses the comparison-based version of the problem. A collection of n records, each composed of k keys, is to be arranged in a table T of size n, one record per entry. Given a value v and a key number i, a record having v in its ith key should be located, or an indication should be provided that there is no such record in the table.

As promised earlier, we can perform the search by $O(\log n)$ comparisons. The constant hidden in the big O is proportional to $k \log k$.

The general strategy is to partition the records into k subsets so that the ith subset is a good "sample" of the records when sorted under the order of the ith key. How can this be achieved is discussed in Section 4.1. Call the ith subset the *i-guides* and call the order of the records under the ith key $\prec_i$.

The members of the ith subset should be kept in their order under $\prec_i$, at least conceptually. Thus, they can function as an approximate oracle for the rank of a record under $\prec_i$. Under such an ordering, for each key i, the records are in some permutation π_i of the order $\prec_i$. The claim made in Section 4.2 is that this

permutation can be computed efficiently in one direction, but in order to retrieve a record, one needs to compute the permutation in the other direction. In other words, the location of a record in the permutation π_i can be computed efficiently, whereas what is really needed is to retrieve a record with a given location in permutation π_i. The cycle chasing technique of Section 3.2 is designed to handle exactly this type of situation.

Recall that the the cycle chasing technique requires $cn \log n$ bits for some fixed $c < 1$. Where would those bits come from? The rainbows, defined in chapter 2, will provide the encoding power, and the technique of Section 3.1 will be used so that the execution of the other task of the i-guides - being an approximate oracle - would not be disrupted. This is done in Section 4.3 and concludes the description of the algorithm. In Section 4.4 it is shown that the techniques developed in this section, in particular the good sampling, have applications outside the framework considered here. In a scenario which is not memory-tight the number of memory references can be halved in comparison with a more naive algorithm.

4.1 Finding a Good Sample

As suggested above, our first problem is to partition the records into k subsets so that the i^{th} subset consists of records that are "fairly evenly spaced" among the values under the i^{th} key. These subsets will, in fact, be of size n/k (we will assume that $8k$ divides n to simplify the presentation). Because of their role in the search procedure, we call the elements of the i^{th} subset the *i-guides.*

Let $T[1..n]$ be the table in which we store the records. The relation $\prec_i$ is the ordering of the records under the i^{th} key. We require that any two entire records be distinct, but the values along individual keys may be equal. We avoid having

to insist that key values be distinct by formally defining $\prec_i$ as the lexicographic (sorted) order on the records induced by concatenating the key values in cyclic order beginning with key i. L_i is used to denote the sequence of records sorted under relation $\prec_i$, i.e.

$$L_i[1] \prec_i L_i[2] \prec_i, \ldots, \prec_i L_i[n].$$

We take evaluation under $\prec_i$ to be a unit-time operation, even when fully lexicographic comparisons are necessary.

A fairly evenly spaced partition of the records is one where between any two successive (under $\prec_i$) elements in the i-guides there would be only a constant (depending only on k) number of elements in L_i. It is not clear a priori that such a partition even exists.

The scheme below shows how to associate each key with its n/k i-guides so that at most $2k-2$ keys fall between two consecutive i-guides under $\prec_i$. (See Figures 2 and 3)

1. Write the lists $L_1, L_2, \ldots, L_k$ as columns of an $n \times k$ matrix of records so that each record appears in each column.

2. Divide each column into n/k sets of k consecutive items. The matrix now contains n blocks, each of size k, and the blocks within each column are pairwise disjoint.

3. Choose one element from each set in such a way that each of the n records is selected exactly once.

Why can step 3 always be completed successfully? Consider the following bipartite graph: on one side it has n nodes, each representing a record, on the other

side $k \cdot n/k = n$ nodes each representing a block. Each record is connected to all the blocks that contain it. The graph is k-regular, and hence by P. Hall's theorem on "complete sets of distinct representatives" [Hal], it contains a perfect matching. Given such a perfect matching a record is considered to be chosen to the i-guides if it matched to a block in L_i. The matching problem can be solved rather quickly: in linear time if k is of the form 2^j by an algorithm in [Gab] and $O(kn \log n)$ time in general [CH].

The records are stored in T such that it is easy to search among the guides for any key. Our (arbitrary) choice is to place the i-guides in consecutive locations:

$$T[(i-1)(n/k)+1], \ldots, T[i(n/k)],$$

sorted by the $\prec_i$ order. We call these n/k locations the *i-cluster.*

Our goal of having the i-guides "fairly evenly spaced" has been achieved.

Remark The j^{th} guide for key i occurs in L_i somewhere between positions $k(j-1)+1$ and $kj-1$. In any list L_i, there are at most $2k-2$ records between consecutive i-guides.

Example: We now give an example of a collection of records and how they are partitioned. Let the collection be:

RECORD	NAME(1)	BIRTHPLACE(2)	YEAR OF BIRTH(3)
i	Carol	Honolulu	1960
ii	Frank	Denver	1955
iii	Alice	Boston	1948
iv	Gus	Atlanta	1958
v	David	Chicago	1960
vi	Iris	Denver	1949
vii	Bob	Philadelphia	1968
viii	Henry	Detroit	1963
ix	Eve	Miami	1958

Figure 2. Sample data for a subsequent figure; $k = 3$ and $n = 9$.

The figure on the next page shows the blocks and the partition.

4.2 The Search

The next stage in describing the data structure is to show how to search the table T with the records arranged as specified at the end of Section 4.1.

The reader ought to think of the *i-cluster* elements as being permanently sorted according to $\prec_i$. The final scheme, however, perturbs that order within each cluster for encoding purposes (see Section 4.3), but the guides for any particular key remain within their cluster in an order that supports a logarithmic search.

The remark above suggests the following partial search strategy for an item with i^{th} key value v:

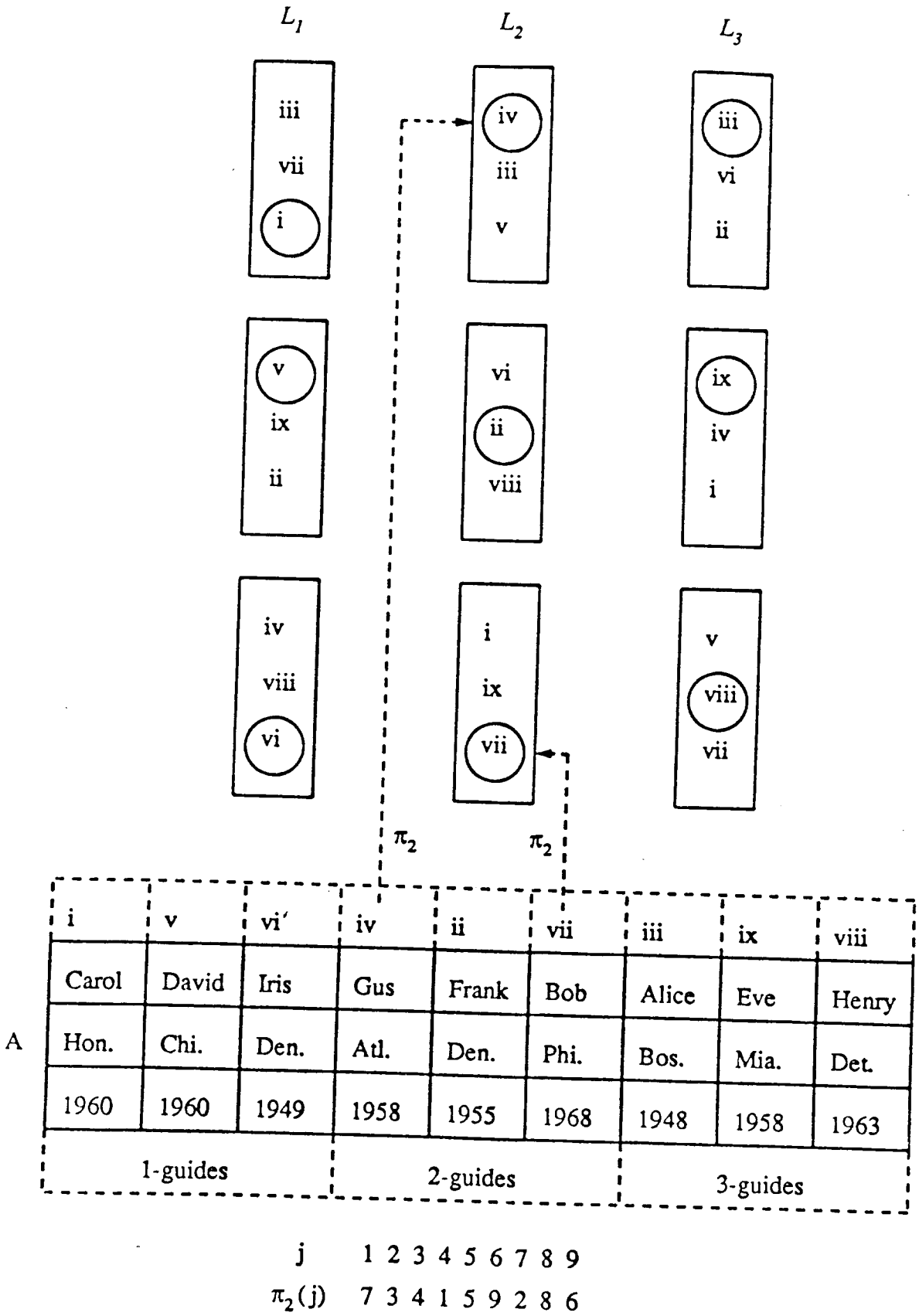

Figure 3. The use of Hall's Theorem; the records chosen as guides are circled. The initial placement of records is shown; Roman numerals are just record names and are not stored. The permutation π_2 is defined and its values at 4 and 6 are illustrated with arrows.

SEARCH-SKETCH(v)

1. Perform a binary search for v among the i-guides (in the i^{th} cluster). If v is not found, then we have determined that the value v may be found in one of $2k-2$ known positions in the (imaginary) list L_i.

2. We can thus restrict our attention to a constant number of positions in L_i. Our goal is to find the locations in T that contain the items from these positions in L_i.

We present the method that finds these items in two stages. In the first stage we use some extra pointers and counters that require $cn \log n$ bits for some fixed $c > 0$ in addition to the table T. In the second stage, deferred to Section 4.3, we show how to encode the $cn \log n$) additional bits of information into the table T, and modify the search algorithm accordingly.

Let $\pi_i : \{1, \ldots n\} \rightarrow \{1 \ldots n\}$ be the permutation that maps a record's index in T into the record's index when the records are sorted under the order $\prec_i$, i.e.,

$$T[\pi_i(1)] \prec_i T[\pi_i(2)] \prec_i \ldots \prec_i T[\pi_i(n)].$$

See Figure 3 for an example.

Step 1 of SEARCH-SKETCH restricts v to at most $2k-2$ known locations in L_i, say locations $q_1 \leq j \leq q_2$. Since $T[\pi_i^{-1}(j)] = L_i[j]$, v can, in principle, be found through a binary search among the $2k-2$ records:

$$T[\pi_i^{-1}(q_1)], T[\pi_i^{-1}(q_1+1)], \ldots, T[\pi_i^{-1}(q_2)]$$

provided we can compute π_i^{-1}. In fact, it turns out that the relative ease of computing π_i is the crucial starting point for computing its inverse. Computing $\pi_i(j)$

is tantamount to finding the rank of $T[j]$ in the order $\prec_i$. To compute $\pi_i(j)$ approximately, we search for the record $T[j]$ in the i-cluster (under the order $\prec_i$). As in step 1 of SEARCH-SKETCH, this gives a range of $2k-2$ possible candidates for $\pi_i(j)$. To compute $\pi_i(j)$ exactly, we associate a vector $F_i[1..n]$ with every key $1 \leq i \leq k$, where $F_i[j] \in \{1,\ldots,2k-2\}$ gives $\pi_i(j)$'s index within the range of $2k-2$ candidates. Note that each array F_i only requires $n \log 2k$ bits.

From the description above we can conclude that the conditions are similar to the ones Section 3.2 deals with. We have an oracle for π_i. Calling it costs $O(\log n)$ comparisons, but since the cycle chasing solution requires only $O(1)$ calls, it does not exceed our time bounds. The solution requires for each key i to have arrays B_i, C_i, D_i, F_i, G_i, of which the size of G_i is dominant. The next section shows how to use the virtual memory supplied by rainbows to satisfy the additional memory requirements of the cycle chasing.

4.3 Encoding

In this section, we show how to perturb T so that the structures $B_i, C_i, D_i F_i$, and G_i, associated with each key $1 \leq i \leq k$, can be implicitly encoded in T. Recall from Section 3.2 that B_i and C_i require $O(n)$ bits, D_i requires $n \log\log n$ bits, and F_i requires $O(n \log k)$ bits. Since ℓ , the effective cycle length, can be chosen independently of n, the storage requirements will be dominated by the G_i's, which might contain $\Theta(n/\ell)$ $\lceil \log n \rceil$ bit words each.

The arrays associated with each key are encoded in the appropriate cluster independently of the others. We can, therefore, view any cluster as an array M that contains m elements in a sorted order; the elements are the i-guides. We phrase the problem we are faced with as follows:

Encoding Problem:

Find a constant $0 \leq c \leq 1$ and an algorithm that encodes $c \cdot m \log m$ bits into a previously sorted table, M, of m distinct elements while attaining the following:

1. Searching for a value in the perturbed table takes time $O(\log m)$.
2. Finding the record that would be in the h^{th} location if the table were sorted takes time $O(\log m)$.
3. The $c \cdot m \log m$ bits are divided into logical words of $\log m - 2$ bits each in such a way that random access to the j^{th} logical word requires $O(\log m)$ comparisons.

We give a solution to the encoding problem: The even positions will remain unchanged. Thus, when given an element x, by $\log m$ comparisons its ranking in the sorted table can be found. We will denote this by $rank(x)$. This will be used for three purposes: the ranking can be used as the function f needed for the technique presented in Section 3.1; it can also provide the values to be manipulated in the rainbows and finally it fulfills requirement 2 of the problem.

Let $S = \{x | x \text{ is in odd location } \}$ and let M' be the table you get by considering only the odd locations in M. Together with the ranking this is the scenario of Section 3.1. the only difference is that calling the oracle requires $\log m$ time, but this is within our means. Following section 3.1, $S_1 = \{x \in S | x \text{ is in location } < m/2\}$ and it is to be organized in the first half of M' in some permutation of it natural order. $S_2 = S \setminus S_1$ will be organized in the inverse permutation.

Let $n' = m/8$. We now implement the greedy encoding algorithm of Section 2.1 via the $(c = 2n' - 1, 2n' - 1, n', t = 2)$-rainbow described in Lemma 2.2. For any $x \in S_1$ its value for rainbow purposes is $(rank(x) + 1)/2$. Thus, the set U in the

algorithm Greedy Encoding is initially $\{1 \ldots m/4\}$ and we can proceed for $m/16$ stages, as long as $|U| \geq m/8$. The number of memory words encoded is $m/8$, so $c = 1/16$.

To summarize

Lemma 4.1 *The encoding problem can be solved with* $c = 1/16$. ∎

When using this for solving our problem we have $m = n/k$. Thus, the amount of virtual memory that can be supplied for each key is $1/16k \cdot n \log(n/4k)$. G_i, the biggest consumer of memory, must be bounded accordingly, by taking the ℓ , the effective cycle length, to be $16k$. Thus computing π_i^{-1} can be done by $O(\log n)$ comparisons, where the constant is proportional to k. Since we are performing a binary search over

$$T[\pi_i^{-1}(q_1)], T[\pi_i^{-1}(q_1+1)], \ldots, T[\pi_i^{-1}(q_2)]$$

π_i^{-1} needs to be computed $\log(2k)$ times.

We have

Theorem 4.1 *A search in an Implicit* k*--key table search can be done in* $O(\log n)$ *time. The constant is proportional to* $k \log k$ *and the preprocessing can be preformed in* $O(n \log n)$ *time.*

4.4 Application to Memory Efficient Data Structures

In this section, we sketch a potentially practical scheme to organize a memory-efficient data structure for solving the k-key table search. We use the theoretical

tools developed in Section 4.1 to suggest a scheme that uses $(k-1)n$ pointers, but can be searched with about half of the number of memory references that would be needed by the obvious kn-pointer solutions.

One obvious solution is to keep k copies of the table, each sorted under a different key. This is clearly wasteful in storage and has other problems as well. A second solution is to keep sorted tables of the form (key-value, pointer). This at least doubles the basic storage requirement (if keys are composed of overlapping fields then it can waste more space) and requires an additional $kn \log n$ bits for pointers. To avoid duplicating the record values, one can store an array of pointers, sorted under the record value they address. This memory-efficient scheme requires 2 memory accesses per comparison, one for the pointer and one for the record's key, and altogether $2 \log n$ memory accesses per search.

To cut down the search time by $1/2$, we partition the the records into i-guides. The basic organization is exactly as above (Section 4.1), i.e. sorted clusters of i-guides in one table. For each key, we keep an array of $n - n/k$ pointers, giving the sorted order of the non-guide records under that key. To search under the i^{th} key, we perform a binary search among the i-guides. If no match is found, the search can nevertheless be limited to $2k-2$ pointers. This method requires at most $\log (n/k) + 2 \log (2k-2) \leq \log n + \log k + 2$ memory references per search and no additional computation.

Chapter 5

Unrestricted Search

In this chapter we consider the version of the problem where there are no restrictions on the the operations performed on the records. From Yao's result mentioned in Chapter 1, we cannot analyze the complexity of this problem independent of the size of the domain from which the keys are drawn.

We first concentrate our efforts on implicit $O(1)$ probe search schemes for the single-key case: A set $S \subset \{1, \ldots, m\}$ of size n is to be stored in a table T of size n, where every table entry stores a single element of S. Given $x \in \{1, \ldots, m\}$, the goal is to locate x in the table while probing T only $O(1)$ times. No additional memory is available.

When does an implicit $O(1)$ probe search scheme exist? It turns out that rainbows play a significant role in answering this question. The relationship between rainbows and implicit $O(1)$ probe search schemes is specified by the following theorems:

Theorem 5.1 *For m and n, let $c = \max(n, \log m)$. The existence of a $(c, m, n, t = O(1))$-rainbow yields an implicit $O(1)$ probe search scheme for n elements from the*

domain $\{1, \ldots, m\}$.

Theorem 5.2 *Given an implicit* $O(1)$ *probe search scheme for* n *elements chosen from the domain* $\{1, \ldots, m\}$, *we can construct an* $(n, m, n, t = O(1))$*-rainbow.*

We can use Theorem 5.1 in combination with theorems 2.1 and 2.2 (that deal with the existence of rainbows) to derive the following:

Corollary 5.1 *For any domain size* m, *polynomial in the set size* n, *there is an implicit* $O(1)$ *probe search scheme for which search requires* $O(1)$ *time assuming modular arithmetic on* $O(\log m)$ *in unit-time. For any domain size* m, *exponential in the set size* n *there is an implicit* $O(1)$ *probe search scheme.*

From Theorem 5.2 in combination with theorem 2.3 (which deals with the impossibility of certain rainbows) we can conclude:

Corollary 5.2 *If there is an implicit* t*-probe search scheme for* m *and* n, *then* $m \leq R(n, t, t! + 1)$.

Theorem 5.1 and 5.2 are proved in Section 5.2

The intellectual efforts we have invested in the implicit case pay off for the other questions presented in the introduction. Showing that an additional memory of $O(\log n + \log \log m)$ bits suffices to assure $O(1)$ probe search for every m and n follows pretty directly from the implicit (zero additional memory) case. Similarly, so does showing that if a probabilistic assumption is made on the input, then no additional memory is required to assure success with high probability. These results are shown in Section 5.3.

The multi-key case is solved by combining the solution to the single-key case with the solution to the multi-key comparison-based case. This is described in Section

5.4. In that section we give another application of our techniques, emulation of an array in sorted order, but with $O(1)$ search time.

Before describing how to get these bounds we must discuss *perfect hash functions.* Perfect hash functions are the basic component for all of our solutions. Instead of storing their description explicitly, which is the previous approach, they will be stored in the virtual memory provided by the rainbows. Thus, we start with describing perfect hash functions in the next section.

5.1 Perfect Hash Functions

Definition:

A function $f : \{1,\ldots,m\} \mapsto \{1,\ldots,n\}$ is a perfect hash function for $S \subset \{1,\ldots,m\}$, $|S| = n$, if it is one to one on S.

Definition:

A family of functions F is (m,n) perfect if for all $S \subset \{1,\ldots,m\}$, $|S| = n$, there is a function $f \in F$ which is perfect on S.

Perfect hash functions are applied to the search problem: given a set $S \subset \{1,\ldots,m\}$, a perfect hash function $f \in F$ is found. S is arranged in T by assigning each $x \in S$ to $T[f(x)]$. The description of f is written in the additional memory. To search for x, $f(x)$ is computed and $T[f(x)]$ is accessed.

It is clear that the length of the description of f is at least $\log|F|$.

The minimum size of F has been thoroughly investigated . The conclusion is that $\log|F|$ must be at least $\Omega(n + \log\log m)$ ([BBDOP], [FKS], [FK], [Mai83], [Meh83]).

For a perfect hash function to be useful it should be efficiently computable and in addition, given x, $f(x)$ should be computable by accessing only a small part of the description of f.

The scheme suggested by Fredman Komlós and Szemerédi has all these properties.

To emulate the perfect hash function f we require the following properties:

- f can be described in $o(n \log n) + O(\log \log m)$ bits.
- The description consists of $o(n)$ words: $v_1, v_2, \ldots, v_h$, where each word v_i is $\max(\log n, \log \log m)$ bits in length.
- To compute f only $O(1)$ of the v_i's need be accessed.

Lemma 5.1 *The scheme described in [FKS] can be implemented with these properties. (In fact, the total memory requirements are* $O(n\sqrt{\log n} + \log \log m)$ *bits.)*

∎

Given a set S, finding the perfect hash function of [FKS] can be done efficiently. The expected number of arithmetic operations is linear. See [DKMMRT].

5.2 Implicit $O(1)$ Probe Search

The close relationship between rainbows an implicit $O(1)$ probe schemes is discussed in this section. We show how to emulate a perfect hash scheme with the properties mentioned above. The memory needed for the perfect hash function will not be stored explicitly, but encoded in virtual memory provided by the rainbows. Section 5.2.1 shows how to use the rainbows so as to eliminate the additional memory requirements (Theorem 5.1 in the introduction of this chapter). Section 5.2.2 shows

how to construct a rainbow given an implicit $O(1)$ probe scheme, and thus provides lower bounds for the existence of implicit $O(1)$ probes (Theorem 5.2 in the introduction of this chapter).

5.2.1 The Scheme

The basis of the scheme is a perfect hash function with the properties mentioned in Lemma 5.1. Rather than storing the block B explicitly, B will be in a virtual memory provided by the rainbows. We use the Greedy Encoding method of section 2.1.

Assume that $\mathcal{C}$ is a $(c = max(n, \log m), m, n/4, t)$-rainbow.

Given the set $S = \{x_1, x_2 \ldots x_n\} \subset \{1, \ldots, m\}$ our goal is to arrange the elements in a table T of size n, without any additional memory, but to enable $O(1)$ worst case search.

The first step is to find a suitable $f : \{1, \ldots, m\} \mapsto \{1, \ldots, n\}$, which is a perfect hash function for S. Order the elements of S in T using the natural order defined by f, i.e., $T[f(x)] := x$ for all $x \in S$. Let $v_1, v_2, \ldots v_h$ be the words of the block B associated with the description of f. Since t is supposed to be $O(1)$, we assume that $h < n/4t$. Let $S_1 = \{T[1], T[2], \ldots, T[n/2]\}$ and $S_2 = S - S_1$.

Second, using the elements of S_1 in the first half of the table, encode $v_1, v_2, \ldots, v_h$ via $\mathcal{C}$ by the greedy encoding algorithm of Section 2.1. Since $h < n/4t$ this is possible. Decoding v_i can be done by accessing the array t times.

The first half of T is now in some permutation τ with respect to the original order. We reorder the second half of T (that contains S_2) by τ^{-1}.

Given a value w to be searched, we run a search algorithm similar to the one

described in Section 3.2, which requires computing f twice, once for the value being searched for and once for a value we find in the table. In Section 3.3 we ignored the question of where f's description is stored. To compute $f(x)$, we have to decode some constant number of v_i's, but that is done by computing the color C associated with a constant number of sequences of entries in T. Overall, this gives us $O(1)$ search time.

We have thus proved theorem 5.1. ∎

5.2.2 Implicit $O(1)$ Probe Search Yields Rainbows

In this section we show that rainbows and $O(1)$ probe search schemes relate in the other direction as well; that is given a search scheme we show how to construct a rainbow. More specifically, we prove a refined version of theorem 2:

Theorem 5.3 *Given an implicit t-probe search scheme for n elements from the domain $\{1,\ldots,m\}$, an $(n,m,n,t+2\lceil\log t\rceil)$-rainbow can be constructed.*

Proof: The sequences are assigned colors based on simulating a search scheme. The idea is that in a t-sequence there is enough information to simulate a t probe search, i.e. given a sequence $v_1, v_2, \ldots v_t$ we simulate a search for v_1 where v_{i+1}, $1 \leq i \leq t-1$, is the element probed at step i. Since the location probed at step i is determined by the the search value and the elements probed in steps 1 through $i-1$, we know the location in the imaginary array at each step of the simulation. The color assigned to the sequence is the last location we are to probe.

The only problem with this description is that v_1 might be probed at any one of the t steps, not necessarily the last, but our sequences do not have repetitions. Hence we need $2\lceil\log t\rceil$ bits to indicate the step number at which v_1 is probed.

This can be done by even odd encoding, in which to encode $1 \leq j \leq t$ a pair of elements is allocated for each bit of j. If the elements are in order they encode 0, otherwise 1. We assume that these elements are at the end of the sequence, that is $v_{t+1}, v_{t+2}, \ldots, v_{t+2\lceil \log t \rceil}$.

To summarize, the color assigned to the sequence $v_1, v_2, \ldots, v_t, v_{t+1}, \ldots, v_{t+2\lceil \log t \rceil}$ is the location of v_1 in the array for which the search is being simulated, where v_1 is encountered in the step encoded by $v_{t+1}, \ldots, v_{t+2\lceil \log t \rceil}$.

Claim 5.1 *Given a set $S \subset \{1, \ldots, m\}$ of size n, and any color $1 \leq i \leq n$, there is a $t + 2\lceil \log t \rceil$-sequence over S which is colored i.*

Proof: Assume that the set S is arranged in the table T so that an implicit t-probe search is possible. For any color i, consider the sequence consisting of the elements probed in T when searching for $T[i]$. concatenate it with $2\lceil \log t \rceil$ elements in S that do not appear in the probe sequence whose order encodes the step number at which cell i is probed. This sequence is colored i, and consists only of elements in S. ∎

Applying theorem 2.3, on the impossibility of the existence of rainbows for certain m and n we get that an implicit k-probe scheme can exist only if $m < R(n, t', t'! + 1)$ where $t' = t + 2\log t$ and $R(n, t', t'! + 1)$ is the Ramsey number defined in Chapter 2. Thus, for an implicit $O(1)$ probe search to exist we must have

$$m \leq 2^{2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}}\}O(1)}.$$

This constitutes a new proof of Yao's theorem [Yao] with better bounds. His bounds imply that $m < R(2n-1, n, n!)$, which grows at least as fast as a tower of powers of 2 of height n. Yao's proof has the advantage that it implies that whenever $m \geq R(2n-1, n, n!)$, the lower bound on the search time is $\lceil \log n \rceil$. Our

proof cannot give better bounds than $\Omega(\log n/\log\log n))$, since $t! + 1$ must be less than n.

Any improvement on the lower bounds for rainbows would yield a better lower bound for implicit $O(1)$ probe search. Conversely, constructive implicit $O(1)$ probe search schemes for higher bounds imply better rainbow constructions. The reader can interpret this as either an optimistic or a pessimistic statement, at his or her choice.

5.3 Coping with nonexistence of implicit schemes

What to do when m and n are such that no implicit $O(1)$ probe search scheme exists?

We will consider two approaches. One assumes some additional memory, and the goal is to minimize its size while still attaining $O(1)$ search time. The other is to make some probabilistic assumption on the input or about the availability of a truly random hash function.

Our solutions in both of these approaches are based on the solution to the implicit case. We reduce the problem to a scenario where an implicit scheme does exist. The extra power granted by the assumption (i.e additional memory or probabilistic assumption) will be used for the reduction.

We start by analyzing the amount of additional memory required to achieve $O(1)$ probe search.

We want to map $S \subset \{1,\ldots,m\}$ to a smaller range, and construct an implicit scheme on the smaller range. Such a function can be taken from [FKS]. Corollary 2 in their paper shows that for any set $S \subset \{1,\ldots,m\}$ of size n, there exists a prime

$p < n^2 \log m$ and a $k < p$ such that the function $g(x) = (k \cdot x \bmod p) \bmod n^2$ is 1-1 when restricted to S. (Alternatively, we could have used universal hash functions of Carter and Wegman [CW]).

Given a set S, the function $g(x)$ which is 1-1 on S is found. The values p and k will be stored explicitly in the additional memory. We then apply the implicit $O(1)$ probe scheme of Section 5.2 but regard each $x \in S$ as $g(x)$. Since $g(x) \leq n^2$ such a scheme is possible. To search for x compute $g(x)$ and search for it. If there is an entry y such that $g(x) = g(y)$ it will be found and then x can be compared to y.

The size of the additional memory required is $\log p + logk$ which is $O(\log n) + O(\log \log m)$. So we get:

Theorem 5.4 *Given a set S of n distinct elements in the range $\{1, \ldots, m\}$, it can be stored in a table T of size n plus an additional memory of size $O(\log n) + O(\log \log m)$, so that searching an element can be done in $O(1)$ time in the worst case.*

The Probabilistic Approach: The approach assumes an idealized random hash function $f(x, i)$, where x is the key value and i is the probe sequence number. Constructions have been given, among others, by Rivest [Riv78], Gonnet and Munro [GM], Schmidt and Shamir [SS], and Celis, Larson and Munro [CLM]. In all these works the elements of S are ordered in T, so that given x the search is performed by accessing $T[f(x,1)], T[f(x,2)], \ldots$ until x is found or until the number of probes exceeds some bound and it can be concluded that x is not in the table. All of these methods have $O(1)$ expected average behavior and $\Omega(\log n)$ expected worst case. Consequently, unsuccessful search requires $\Omega(\log n)$ probes. In fact, Gonnet [Gon] has shown that under these assumptions, the longest probe sequence in any such non adaptive scheme must be $\Omega(\log n)$ with high probability.

Our results are in sharp contrast to that. By assuming either a random hash function $f : \{1,\ldots,m\} \mapsto \{1,\ldots n^3\}$, or that S is chosen uniformly at random from $\{1,\ldots,m\}$, we can construct a scheme that has a worst case search $O(1)$ with high probability.

The idea is simply to use the randomness to get to a situation where the rainbows can work. First note that f is 1-1 on S with probability $> 1 - 1/n$, and similarly if S is chosen uniformly at random, then with high probability the function $f(x) = x \bmod n^3$ is 1-1. Hence, as before, we can apply the implicit $O(1)$ probe scheme. Each $x \in S$ will be regarded as $f(x)$. To search for x, $f(x)$ is computed and searched for under the implicit scheme. If there exists an entry y such $f(y) = f(x)$, then x is compared to y. This scheme does not use any additional memory.

A different method of obtaining these bounds was presented in [FNSS], where the scheme of Schmidt and Shamir [SS] is used directly. This was needed since no good rainbow constructions were known at that time.

We conclude

Theorem 5.5 *A random hash function can be used to store any set S of n distinct elements from the range $\{1,\ldots,m\}$ in a table T of size n, giving worst case search time $O(1)$ with probability at least $1 - 1/n$.*

5.4 Multi-key Tables and Successor Computation

We show in this section two more applications of the techniques introduced in Chapters 2 and 3. The application will be similar in nature to those in chapter 4. Section 5.4.1 gives the multi-key application and Section 5.4.2 gives the application to successor computation.

5.4.1 Multi-key Tables

A collection of n multi-key records are to be organized in a table T of size n, so that search can be performed on any one of the k keys, in $O(1)$ worst case time. For simplicity, assume that all k keys in a record are numbers in the range $\{1,\ldots,m\}$ and that for each key i, $1 \leq i \leq k$, there do not exist two records with the same value in key i.

We first assume that m and n are such that an implicit $O(1)$ probe search scheme of Section 5.2.1 works. If this assumption is not true, then we can revert to the methods of the previous section.

Let S_i be the set of key values under the ith key. Let t be such that an $(c = n, m, n/4, t)$-rainbow exists.

The multi-key scheme: For each key i construct a perfect hash function for the set S_i. Let T_i be an imaginary table that is arranged according to the natural order f_i induces. We initially take T to be T_1. For each table T_i, $1 < i \leq k$, there is some permutation τ_i that maps T_i to T_1. Given access to f_i, $\tau_i(j)$ can be computed efficiently by evaluating $f_i(T[j])$. However, to locate keys according to the ith key we need to be able compute τ_i^{-1}, since $T[\tau_i^{-1}(f_i(x))]$ is where the record that contains x in its ith key is located, if such a record exists.

Sounds familiar? Yes, because these are exactly the conditions of the hypothesis of Theorem 3.1. Similarly to Section 4.2, if we allocate for each key $1/16tk \cdot n \log n$ bits, then the problem of computing τ_i^{-1} can be solved in $O(1)$ time.

What are the memory requirements so far? for each key we need to store the description of f_i, which is $O(n\sqrt{\log n}+\log\log m)$, and the $1/16k \cdot n \log n$ bits required in the solution of theorem 3.1. Altogether this is less than $1/4tn \log n$ bits, at least for large enough n, which is the bound on our virtual memory. We store this in

the virtual memory that can be obtained from f_1 exactly as in Section 5.2.1: a permutation of T_1 will encode the memory words via the rainbow, yet the original order can be reconstructed in $O(1)$ time.

As in Section 5.3, in case there is no rainbow we can use either additional storage or a probabilistic assumption to reduce the scenario to a one where rainbows exist.

To summarize:

Theorem 5.6 *Any set S of n records, each record consisting of k different keys, where m is a bound on the maximal key size, can be stored in a table T of size n where:*

1. *If a $(c = n, m, n, t)$-rainbow with t a constant exists, then without any additional memory we can assure search under any key with worst case time $O(1)$*

2. *An additional $O(\log n + \log\log m)$ bits guarantee that records can be searched under any key in worst case time of $O(1)$.*

3. *A random hash function gives worst case $O(1)$ search time, under any key, with high probability.*

5.4.2 Successor Computation and Sorted Array Emulation

Hashing in general is a method that allows $O(1)$ access when the value being searched for is known exactly. Hashing is problematic when it comes to searching for the next entry greater than the search value. This problem has been addressed by Ajtai, Fredman and Komlós [AFK] who give a solution requiring an additional $n \log\log m$ bits of storage. This storage is used to hold a trie and search can be performed by accessing $O(\log m)$ nodes. If m is sufficiently large then the trie can be

stored in one extra word of $\log m$ bits. Therefore, if we assume that the additional memory has words of size $O(\log m)$, then implementing it requires $O(1)$ probes when m is exponential in n, though it is not clear how to conduct the computation in $O(1)$ time.

On the other hand, Ajtai [Ajt] has shown that for certain values of m and n this problem cannot be solved in a constant number of probes even if the number of cells available is as large as any polynomial in n.

We would like to solve the successor problem, while attaining the quick search retrieval property and without using additional memory. We can also solve the related problems of determining the rank of a value and finding the i^{th} largest element. Because these are the properties of a sorted array we call it *sorted array emulation.*

We will give the solutions in the cases for which we know an implicit $O(1)$ probe search scheme. To solve these problems we use the $\epsilon n \log n$ virtual bits at our disposal from the encoding, for some fixed $\epsilon > 0$. For some constant d such that $c < 1/\epsilon$, construct an array of n/d virtual pointers, so pointer i points to the $(d \cdot i)^{\text{th}}$ element in rank. We can decode such a pointer in time $O(1)$ and can thus perform a binary search on these elements. This lets us compute the rank of a value approximately, within a region of uncertainty of size d. This uncertainty can be removed for elements in the hash table by holding a virtual vector of size n with $\log d$ bits per entry, giving the appropriate offset for every element by index in the *unpermuted* hash table (before the current encoding). This lets us compute the rank for an element in the table in $O(\log n)$ probes.

Consider the rank permutation which transforms the unpermuted table ordered under the natural perfect hash order to a sorted table ordered by rank. This permutation can be computed using the rank function in time $O(\log n)$. These are exactly

the conditions under which we can apply the cycle chasing technique of Section 3.2. Thus, the rank permutation can be computed in time $O(\log n)$. If the effective cycle length in the cycle chasing is taken to be sufficiently large, then all the additional memory required for implementing the cycle chasing can be encoded in the virtual memory available. This lets the i^{th} largest element be found in time $O(\log n)$.

Finally, the successor to a value (that need not be present in the table) can be found by performing an initial binary search on the n/d elements whose pointers are encoded directly (in time $O(\log n)$). This gives us an interval of d ranks, with inverses that can be binary searched in an aggregate $O(\log d \log n) = O(\log n)$ time.

To summarize:

Theorem 5.7 *Computing the successor and rank functions and finding the ith largest element, can be performed on a full table in time* $O(\log n)$.

Chapter 6

Further Research

Defining an abstract model for a computational problem is a delicate art; as we have seen, seemingly minor changes within reasonable models of computation result in significant change in the performance. Thus, when defining a model, one should differentiate between restrictions that are inherent within the problem and those that are imposed for the sake of the analysis.

Researchers in concrete complexity have the advantage that, unlike their peers in abstract complexity, they can show non-trivial lower bounds. On the other hand, the models in which the lower bounds are shown are not robust. Among problems of the type considered in this thesis, we are aware of only one result that can be regarded as robust, at least in one respect. This is Ajtai's lower bound for the successor function mention in section 5.4. However it is unsatisfactory in the time bounds.

We suggest the following problem, known as the partial-match retrieval problem [Riv76], as a candidate for which robust bounds might be shown.

Partial-Match Retrieval Problem:

Input to the preprocessor: a set S of n elements in $\{1, \ldots, 2^l\}$

Query: a string $s \in \{0, 1, \star\}^l$.

Find an element in S that corresponds to s in the bit positions with 0 and 1 and has arbitrary value for those with a $\star$, or indicate that none exists.

The goal is to organize S in memory of size polynomial in n and l, so that answering a query can be done in time polylogarithmic in n and l. We conjecture that this is impossible, at least for n and l of a certain relationship.

The results in this thesis suggest several open problems. First, we know that implicit $O(1)$ probe search for the single-key unrestricted search is possible when the domain size m is exponential in the set size n. However, it is based on the probabilistic construction of the rainbows. Can this construction be made explicit?

There is a gap between the existence and impossibility results on implicit $O(1)$ probe search. Can it be tightened?

Research in data compression has primarily focused on global compression, i.e. an object is compressed as a whole and then the only way to access a specific part of it is by uncompressing the entire object. Some of ideas developed in this thesis where used in [Nao] to show how to represent a general unlabeled graph with an optimal number of bits. However, this is a global compression. Can the techniques presented in this thesis, namely the rainbows be useful for providing a representation which allows to uncompress specific parts of the graph, or in general to provide random access compression?

This thesis did not consider at all the dynamic case, where the collection of records is modified by insertions and deletions. Progress in that area has been made ([Mun86], [FG], [DKMMRT]), however the analysis is far from being complete. Even for the single-key comparison-based case no tight bounds are known.

Bibliography

[Ajt] M. Ajtai, A Lower Bound for Finding Predecessors in Yao's Cell Probe Model, IBM RJ 4867 (51297) 10/3/85, Computer Science.

[AFK] M. Ajtai, M. Fredman and J. Komlós, Hash Functions for Priority Queues, the 24*th* Annual Symposium on Foundations of Computer Science, 1983, pp. 299 – 303.

[AKS] M. Ajtai, J. Komlós, E. Szemerédi, Deterministic Simulation in LOGSPACE, Proc. 19*th* ACM Symposium on Theory of Computing, 1987 pp. 132-140.

[AMM] H. Alt, K. Mehlhorn, and J. I. Munro, Partial Match Retrieval in Implicit Data Structures, *Information Processing Letters* 19(1984), pp. 61–65.

[BBDOP] F. Berman, M.E. Bock, E. Dittert, M.J. O'Donnell and D. Plank, Collections of Functions for Perfect Hashing, *SIAM Journal on Computing*, 15(1986), pp. 604 – 618. July 1984, pp. 538–544.

[BFMUW] A. Borodin, F. E. Fich, F. Meyer auf der Heide, E. Upfal, and A. Wigderson, Tradeoff Between Search and Update Time for the Implicit Dictionary Problem, Proc. 13th Inter. Colloq. Automata, Lang., and

Prog., Rennes, France *Lecture Notes in Computer Science 226*, Springer-Verlag, 1986, pp. 50–59; *Theoretical Computer Science* 58(1988), pp. 57–68.

[CW] J.L. Carter and M.N. Wegman, Universal Classes of Hash Functions, *Journal of Computer and Systems Sciences* 18, No. 2 (April 79), 143 –154.

[CLM] P. Celis, P. Larson and J. Ian Munro, Robin Hood Hashing, Proceedings of the 26*th* Annual Symposium on Foundations of Computer Science, 1985, pp. 281 – 288.

[CG] B. Chor and O. Goldriech, Unbiased Bits From Sources of Weak Randomness and Probabilistic Communication Complexity, *SIAM Journal on Computing.* 17(1988), pp. 230-261.

[CH] R. Cole and J. Hopcroft, On Edge Coloring Bipartite Graphs, *SIAM Journal on Computing* 11(1982), pp. 540–546.

[DKMMRT] M. Ditzfelbinger, A. Karlin, K. Melhorn, F. Meyer auf der Heide, H. Rohnert and R.E. Tarjan, Dynamic Perfect Hashing: Upper and Lower Bounds, the 29*th* Annual Symposium on Foundations of Computer Science, 1988, pp. 524 – 531.

[FN] A. Fiat and M. Naor, Implicit $O(1)$ Probe Search, Proc. 21st ACM Symposium on Theory of Computing, Seatle, (1989), pp. 336–344.

[FMNSSS] A. Fiat, I. Munro, M. Naor, A. Schäffer, J. P. Schmidt and A. Siegel, An Implicit Data Structure for Searching a Multikey Table in Logarithmic Time, manuscript, (to appear, Journal of Computer and System Sciences).

[FNSSS] A. Fiat, M. Naor, A. A. Schäffer, J. P. Schmidt, and A. Siegel, Storing and Searching a Multikey Table, Extended Abstract, Proc. 20th ACM Symposium on Theory of Computing, Chicago, (1988), pp. 344–353.

[FNSS] A. Fiat, M. Naor, J. P. Schmidt, and A. Siegel, Non-Oblivious Hashing, Extended Abstract, Proc. 20th ACM Symp. on Theory of Computing, Chicago, (1988), pp. 367–376.

[FG] M.L. Fredman and D.L. Goldsmith, Three Stacks, the 29*th* Annual Symposium on Foundations of Computer Science, 1988, pp. 514 – 523.

[FK] M.L. Fredman and J. Komlós, On The Size Of Separating Systems And Families Of Perfect Hash Functions, *SIAM Journal of Algebraic and Discrete Methods* , Vol 5, No. 1, March 1984, pp. 61–68.

[FKS] M.L. Fredman, J. Komlós and E. Szemerédi, Storing a Sparse Table with $O(1)$ Worst Case Access Time, *Journal of the Association for Computing Machinery* , Vol 31, No. 3, July 1984, pp. 538–544.

[Gab] H. N. Gabow, Using Euler Partitions to Edge Color Bipartite Multigraphs, *Int. J. Comp. Inf. Sci.* 5(1976), pp. 345–355.

[Gon] G.H. Gonnet, Expected Length of the Longest Probe Sequence in Hash Code Searching, *Journal of the Association for Computing Machinery*, Vol 28, No. 2, April 1981, pp. 289–304.

[GM] G.H. Gonnet and J. Ian Munro, Efficient Ordering of Hash Tables, *SIAM Journal on Computing*, Vol 8, No. 3, August 1979, pp. 463–478. July 1984, pp. 538–544.

[GRS] R. L. Graham, B. L. Rothschild and J. H. Spencer, **Ramsey Theory**, Willey 1980.

[Hal] P. Hall, On Representations of Subsets, *J. London Math. Soc.* 10(1935), pp. 26–30.

[JV] C.T.M. Jacobs and P. Van Emde Boas, Two Results on Tables, *Information Processing Letters* 22 (1986), 43–48.

[Mai83] H. G. Mairson, The Program Complexity of Searching a Table, 24th Symposium on Foundation of Computer Science, November, 1983

[Mai84] H. G. Mairson, The Program Complexity of Searching a Table, PhD Dissertation, Stanford, 1984, STAN-CS-83-988.

[Meh82] K. Mehlhorn. On the Program size of Perfect and Universal Hash functions, Proc. 23*rd* Ann. Symp. on Foundations of Computer Science, 1982, pp. 170–175.

[Meh84] K. Mehlhorn, **Data Structures and Algorithms 1: Sorting and Searching**, Springer-Verlag, Berlin Heidelberg, 1984.

[Mun79] J. Ian Munro, A Multikey Search Problem, Proc. 17th Allerton Conference on Communication, Control, and Computing, Monticello, Illinois, (1979), pp. 241–244.

[Mun86] J. Ian Munro, An Implicit Data Structure Supporting Insertion, Deletion, and Search in $O(\log^2 n)$ Time, *Journal of Computer and Systems Sciences* 33(1986), pp. 66–74.

[Mun87] J. Ian Munro, Searching a Two Key Table Under a Single Key, Proc. 19th ACM Symp. on Theory of Computing, New York City (1987), pp. 383–387.

[MS] J. Ian Munro and Hendra Suwanda, Implicit Data Structures for Fast Search and Update, *Journal of Computer and Systems Sciences*, 21(1980), pp. 236–250.

[Nao] M. Naor, Succinct Representation of General Unlabeled Graphs, Manuscript.

[Riv76] R.L. Rivest, Partial-Match Retrieval Algorithms, *Siam Journal on Computing* , Vol 5, No. 1, March 1976, pp. 19-50.

[Riv78] R.L. Rivest, Optimal Arrangement of Keys in a Hash Table, *Journal of the Association for Computing Machinery*, Vol 25, No. 2, April 1978, pp. 200–209.

[SS] J.P. Schmidt and E. Shamir, An Improved Program for Constructing Open Hash Tables, ICALP 80, Proceedings, pp. 569 – 581.

[Sip] M. Sipser, Expanders, Randomness, or Time versus Space, Structure in Complexity, June 1986, pp. 325-330.

[TY] R.E. Tarjan and A.C. Yao, Storing a Sparse Table, *Communications of the ACM*, Vol 22, No. 11, November 1979, pp. 606–611.

[Vaz] U. V. Vazirani, Randomness, Adversaries and Computation, Phd Dissertation, University of California, Berkeley 1986.

[Yao] A.C. Yao, Should Tables Be Sorted?, *Journal of the Association for Computing Machinery*, Vol 28, No. 3, July 1981, pp. 615–628.