Membership in Constant Time and Minimum Space*

Andrej Brodnik and J. Ian Munro

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1
{ABrodnik,IMunro}@UWaterloo.CA

Abstract. We investigate the problem of storing a subset of the elements of a bounded universe so that searches can be performed in constant time and the space used is within a constant factor of the minimum required. Initially we focus on the static version of this problem and conclude with an enhancement that permits insertions and deletions.

1 Introduction

Given a universal set $\mathcal{M} = \{0, \ldots, M-1\}$ and any subset $\mathcal{N} = \{e_1, \ldots, e_N\}$ the membership problem is to determine whether given query element in $\mathcal{M}$ is an element of $\mathcal{N}$. There are two standard approaches to solve this problem: to list all elements of $\mathcal{N}$ (e.g. in a hash table) or to list all the answers (e.g. a bit map of size M). When N is small the former approach comes close to the information theoretic lower bound on the number of bits needed to represent an arbitrary subset of the given size (i.e. a function of both N and M, $\left\lceil \lg \binom{M}{N} \right\rceil$). Similarly, when N is large (say αM) the later approach is near optimal in space utilization. This paper focuses on the middle ground where both of these natural approaches require space nonlinear in the lower bound. The techique we introduce has the feature that not only is the space used within a constant factor of the lower bound and the query time is constant, but also that both of these constants are modest.

The basic approach is to use either perfect hashing or a bit map whenever they achieve the optimum space bound, and otherwise to partition the universe into subranges of equal size. We discover that, with care, after a couple of iterations of this splitting, the subranges are small enough so that succinct indices into a single table of all possible configurations of these small ranges (*table of small ranges*) permit the encoding in the minimal space bound. Our technique is, then, an example of what we call *word-size truncated recursion* (cf. [11, 12]). Namely, the recursion needs only to continue down to a level of "small enough" subproblems, at which point indexing into a table of all solutions suffices. This happens to a large extent because at this level a single word in the machine model is big enough to encode a complete solution to each of these small problems.

* This research was supported in part by the Natural Science and Engineering Research Council of Canada under grant number A-8237 and the Information Technology Research Centre of Ontario.

Our paper is organized as follows: in the next section we introduce the notation, some definitions and a short overview of a relevant literature. In Sect. 3 we present a complete static solution, which is followed by a solution of the dynamic problem in Sect. 4. In Sect. 5 we present a pair of examples of usage of our static structure and end the paper with final conclusions and remarks.

2 Notation, Definitions and Background

As noted above, our universal set, $\mathcal{M}$, is assumed to be $\{0, \ldots, M-1\}$, hence of size M. On the other hand, $\mathcal{N}$ is an arbitrary subset of $\mathcal{M}$ of cardinality N. Since solving the membership problem for $\mathcal{N}$ trivially gives a solution for $\overline{\mathcal{N}}$ we assume $0 \leq N \leq \frac{M}{2}$.

The membership problem is to answer a query whether a given element $x \in \mathcal{M}$ is in $\mathcal{N}$. The problem has the obvious dynamic extension which also includes the operations insert and delete.

We use the standard *cell probe model* (cf. [6, 10, 15]) with word size $\lceil \lg M \rceil$ bits.[2] In other words, one register can be used to represent a single element from the entire domain $\mathcal{M}$, specify all elements which are present in a set of a size $\lg M$ elements, refer to some portion of the data structure, or have some other role that is a ($\lceil \lg M \rceil$ bit) blend of these. For convenience we measure the space usage in bits (rather than in words). Furthermore, we assume that the machine can perform usual operations on words in one unit of time.

The information theoretic lower bound on the number of bits needed to represent an arbitrary set of size N from the universe of size M is

$$B \equiv \left\lceil \lg \binom{M}{N} \right\rceil \quad . \tag{1}$$

Using Stirling's approximation and defining $r = \frac{M}{N}$, this formula can be replaced by the approximation

$$B \approx N \lg \frac{M}{N} = N \lg r \quad . \tag{2}$$

The error in this estimate is less than $N \lg e$ and so (2) can be used for B throughout this paper.

2.1 A Short Stroll Through the Literature

The searching problem is one of central problems of computer science with an extensive literature. In this section we point out particular work that is essential to ours. There are three aspects which are addressed: the static case, the dynamic case, and the information theoretic tradeoffs. In the first two cases it has always been assumed that there is at least enough space to list all keys that are present (or to use a bit map). However, we are dealing with the situation in which we can not always afford that much space.

We start with the static case, and, in particular, with the implicit data structures in Yao's sense. His notion of implicit structure was that one has room to store only N data

[2] We use a notation $\lg x$ to describe a binary logarithm $\log_2 x$.

items, although these were not constrained to be those in the "logical" table. In other words, by the term implicit structure he meant a structure which uses only N registers ($N \lg M$ bits) and no additional storage.

For such data structures Yao ([15]) showed that in a bounded universe there always exists some subset for which any implicit data structure requires at least logarithmic search time. Later Fiat and Naor ([7]) put a better bound on the size of the subset N. They proved that for this awkward case, N has to be larger than a quantity which lies somewhere between $\lg^{(O(1))} M$ and $(\lg M)^\epsilon$ $(0 \leq \epsilon < 1)$.[3] In particular, they proved that there exists an implicit scheme for $N = \Omega((\log M)^\epsilon)$.

However, adding some storage changes the situation. For example, Yao ([15]) showed that there exists a constant time solution for $N \approx M$ or $N \leq \frac{1}{4}\sqrt{\lg M}$, while Tarjan and Yao ([14]) presented a constant time solution using $O(N \lg M)$ bits of space for $N = O(M^\epsilon)$ and $0 < \epsilon \leq 1$. Fredman, Komlós and Szemerédi ([9]) closed the gap developing a constant time algorithm which uses a data structure of size $N \lg M$ bits and $O(N\sqrt{\log N} + \log^{(2)} M)$ additional bits of storage, for any N and M. Fiat, Naor, Schmidt and Siegel in [8] decreased the number of additional bits to $\lceil \lg N \rceil + \left\lceil \log^{(2)} M \right\rceil + O(1)$. Moreover, combining their result with Fiat and Naor's ([7]) construction of an implicit search scheme for $N = \Omega((\log M)^p)$ we get a scheme which uses less than $(1+p)\left\lceil \log^{(2)} M \right\rceil + O(1)$ additional bits of memory.

Moving to the dynamic case, Dietzfelbinger et al. ([2]) proved an $\Omega(\log N)$ worst case lower bound for a class of realistic hashing schemes. In the same paper they also presented a scheme which, using results of [9] and a standard doubling technique, achieved a constant amortized expected time per operation with a high probability. However, the worst case time per operation (non-amortized) was $\Omega(N)$. Later Dietzfelbinger and Meyer auf der Heide in [3] upgraded the scheme and achieved a constant worst case time per operation with a high probability. A similar result was also obtained by Dietzfelbinger, Gil, Matias and Pippenger in [1].

Finally we turn to some information theoretic references. Elias ([4]) addressed a more general version of a static membership problem which involved several different types of queries. For these queries he discussed a tradeoff between the size of the data structure and the average number of bit probes required to answer the queries. In particular, for the set membership problem he described a data structure of a size $N \lg \frac{M}{N} + O(N)$ – that is, using (2), $B + o(B)$ – bits which required an average of $(1+\epsilon)\lg N + 2$ bit probes to answer a query. However, in the worst case, it was still necessary to probe N bits. Elias and Flower in [5] further generalized the notion of a query into a database. They defined the set of data and a set of queries, and in such a general setting they studied the relation between the size of the data structure and the number of bits probed, given the set of all possible queries. The same arrangement was later more rigorously studied by Miltersen in [13].

[3] $\log^{(p)} x$ denotes the iterated logarithm, namely the logarithm function applied p times to x.

3 A Solution for the Static Case

As mentioned earlier, if more than half the elements are present we can solve the complementary problem. In the solutions below it is implicit that if any subrange (bucket) has more than half its elements present we use a single bit flag to indicate that we are solving the complementary problem. The static solution is presented in two parts. In the first part we describe a solution for $N \leq \frac{M}{\log_\lambda M}$, that is for $r \geq \log_\lambda M$, while in the second part we deal with the case when $\frac{M}{\log_\lambda M} \leq N \leq \frac{M}{2}$, that is for $\log_\lambda M \geq r \geq \frac{1}{2}$. In both cases λ is some predefined constant greater than 1. We refer to the first case as *sparse* and to the second as *dense*.

Each of these cases has one extreme situation. The first is when $N \leq M^{1-\epsilon}$ for some $0 < \epsilon \leq 1$. In this case we need $\Omega(N \log M)$ bits which is enough to list all the keys. Moreover, this is also enough space to use a perfect hashing function in any of its flavours (cf. [7, 8, 9]) and so retain a constant response time in the worst case. The other extreme is when N is very large, that is $N \geq \alpha M$ for some $0 < \alpha \leq \frac{1}{2}$. The number of bits necessary in this case is $\Omega(M)$ and hence we can use a bit map of size M to represent the subset.

3.1 Indexing – Solution for $N \leq \log_\lambda M$

In this section we focus on case $N \leq \frac{M}{\log_\lambda M}$ for some $\lambda > 1$. The top end of this range, that is $N = \frac{M}{\log_\lambda M}$, typifies the case in which both simple approaches require too much space as $B = \Theta(N \log^{(2)} M)$. Furthermore, this solution suggests the first iteration of our general approach in the $N \geq \frac{M}{\log_\lambda M}$ case.

Lemma 1. *If $N \leq \frac{M}{\log_\lambda M}$ (i.e. $M \geq r \geq log_\lambda M$) for some $\lambda > 1$, then there is an algorithm which answers a membership query in constant time using $O(B)$ bits of space for a data structure.*

Proof. The idea is to split a universe $\mathcal{M}$ into p buckets, where p is to be determined later and will be as large as possible. The data falling into individual buckets is then organized using perfect hashing. The buckets are contiguous ranges of equal sizes, so that a key $x \in \mathcal{M}$ falls into bucket $\left\lfloor \frac{x}{p} \right\rfloor$ and to reach individual buckets, we index through an array of pointers.

To consider this in detail, let us assume that we split the universe into p buckets. We build an index of pointers to individual buckets, where each pointer occupies $\lceil \lg M \rceil$ bits. Hence, the total size of the index is $p \cdot \lceil \lg M \rceil$ bits.

We store all elements that fall in the same bucket in a perfect hash table ([7, 8, 9]) for that bucket. Since the ranges of all buckets are equal, the space required for these hash tables is $\beta = \left\lceil \lg \frac{M}{p} \right\rceil$ bits per element, and so, to describe all elements in all buckets we require only $N \cdot \beta$ bits.

If we allocate B bits for the index and about the same for the buckets, we achieve the desired space bound. Moreover, this structure also supports constant time queries.

It remains to show that $O(B)$ bits are enough to store elements in the buckets' perfect hash tables. From the limit on the size of the index and using (2) we get

$$p = \left\lfloor \frac{B}{\lceil \lg M \rceil} \right\rfloor = \Theta(\frac{M \log r}{r \log M})$$

buckets, each of which has a range $\frac{M}{p} = \Theta(\frac{r \log M}{\log r})$. The range of a bucket defines the number of bits used by each element in a hash table $\beta = \Theta(\log^{(2)} M + \log r - \log^{(2)} r)$, which, in turn, gives the total size of all hash tables $\Theta(N(\log^{(2)} M + \log r - \log^{(2)} r))$. Moreover, this order matches information theoretic lower bound in (2) for $r \geq \log_\lambda M$. $\square$

The cases covered so far are summarized in Table 1. The first column of the table holds ranges of N as an increasing function of M. The second column contains approximations of B on the top end of the range in the first column and was obtained using (2), while the third column defines the data structure which should be used for N in the range defined in the first column. In the table we can find solutions for small or large N, while as yet we have no solution for

$$\begin{array}{c} \frac{M}{\log_\lambda M} \leq N \leq \alpha M \leq \frac{M}{2} \\ \log_\lambda M \geq \ r \ \geq \ \frac{1}{\alpha} \ \geq 2 \ . \end{array} \tag{3}$$

Next we will assume that r and N lie in this awkward range, that is that subset at hand is dense.

Table 1. Ranges of N and structures used in them to represent subset $\mathcal{N}$.

range of N	B	structure
0	1	single bit
1 to c	$\Theta(\log M)$	unordered list
c to $M^{1-\epsilon}$	$\Theta(N \log M)$	hash table
$M^{1-\epsilon}$ to $\frac{M}{\log_\lambda M}$	$\Theta(N \log^{(2)} M)$	indexing
$\frac{M}{\log_\lambda M}$ to αM	$\Theta(N \log r)$	?
αM to $\frac{1}{2} M$	$\Theta(M)$	bit map

3.2 A Complete Solution

We return to Table 1 and in particular to subsets $\mathcal{N}$ whose sizes lie in the range given in (3). For such dense $\mathcal{N}$ we apply the technique of Lemma 1, that is split the universe $\mathcal{M}$ into equal-range buckets (using at most B bits for the pointers). However, this time the ranges of buckets remain too big to use hash tables, and therefore we apply the splitting scheme again. In particular, we treat each bucket as a new, separate, but smaller universe and if the number of elements falling into the bucket falls in the range defined by (3) (with respect to the size of its smaller universe) we can recursively split it.

Such a straightforward strategy leads, in the worst case, to an $O(\log^* M)$ level structure and therefore to an $O(\log^* M)$ search time and to an $O((\log^* M)\cdot B)$ space requirement. However, we observe that at each level the number of buckets with the same range increases and, furthermore, their number is becoming so large that not all of them can be different. Therefore we could build a table of all possible subsets of universes of size up to a certain threshold. That is, we could build a *table of small ranges* and replace buckets in the main structure by pointers (indices) into the table (see [11, 12] for example). We refer to this technique as *word-size truncated recursion*. In our structure the truncation occurs after two splittings. In the rest of this section we give a detailed description of the structure and its analysis.

Returning to the problem at hand, on the first split we partition the universe into $p = \frac{M}{(\log_\lambda M)^2}$ buckets each of which has a range $M_1 = \frac{M}{p} = (\log_\lambda M)^2$. This gives three kinds of buckets at the second level: sparse, dense and very dense ones. For sparse buckets we apply the solution from Subsect. 3.1 and for very dense ones with more than the fraction α of their elements present we use a bit map. For the remaining dense buckets (they have a number of elements within the range defined in (3)) we re-apply the splitting. However, this time the number of buckets is $p_1 = \frac{M_1}{(\log_\lambda M_1)^2}$ so that each of these smaller buckets has range $M_2 = \frac{M_1}{p_1} = 4(\log_\lambda^{(2)} M)^2$.

At this point we build the table of small ranges, which consists of a bit map representations of all possible subsets chosen from the universe of size $M_2 = \sigma$, and replace buckets in the main structure with "indices" (of varying sizes) into the table. We order the table first according to the number of elements in the subset and then lexicographically. If we store, as the representation of a pointer to the table of small ranges, a record consisting of two fields: the number of elements in the bucket, ν (this takes $\lceil \lg \sigma \rceil$ bits), and the lexicographic order of the bucket in question among all buckets containing ν elements, η (from (1) this is $\lceil \lg \binom{\sigma}{\nu} \rceil$ bits), then we can compute the actual position of the corresponding bit map of the bucket using

$$\sum_{i=1}^{\nu-1} \binom{\sigma}{i} + \eta - 1 \ . \tag{4}$$

The sum is found by table lookup and so a search is performed in constant time.

This concludes the description of our data structure. Obviously, the structure allows constant time membership queries, but it remains to be seen how much space it occupies. In the analysis we are interested only in dense subsets, that is in the range defined in (3), as otherwise we use structure presented in Subsect. 3.1.

First we analyze the main structure and in order to bind later results together we will need the following lemma:

Lemma 2. *Suppose we are given a subset of N elements from the universe M and B defined as in (1). If this universe is partitioned into p buckets of sizes M_i containing N_i elements (now, using (1), $B_i = \left\lceil \lg \binom{M_i}{N_i} \right\rceil$) then $B \geq \sum_{i=1}^{p} B_i - p$.*

Proof sketch. We observe that $B_i < \lg \binom{M_i}{N_i} + 1$ and that $\prod_{i=1}^{p} \binom{M_i}{N_i} \leq \binom{M}{N}$ where $\sum_{i=1}^{p} M_i = M$ and $\sum_{i=1}^{p} N_i = N$. □

The main structure itself is analyzed from the top to the bottom. The first level index consists of p pointers each of which is of size $\beta = \lg M$ bits. Therefore the size of that complete index is

$$p \cdot \beta = \frac{M}{(\log_\lambda M)^2} \cdot \lg M = o(\frac{M}{\log M}(\log^{(2)} M)) = o(B) \tag{5}$$

for the range defined in (3).

On the second level we have three kinds of buckets: sparse, dense, and very dense. For the sparse buckets we use solution presented in Subsect. 3.1 and for the very dense we use a bit map. Both of these structures guarantee space requirements within a constant factor of the information theoretic bound on the number of bits. If the same also holds for the dense buckets, then, using Lemma 2 and (5), the complete main structure uses $O(B)$ bits. Note, that we can apply Lemma 2 freely because the number of buckets p is, by (5), $o(B)$.

It remains to see how large the second level dense buckets can be. For this purpose we first consider the size of bottom level pointers (indices) into the table of small ranges. As mentioned, the pointers are records consisting of two fields, where the first field occupies $\lceil \lg \sigma \rceil$ bits and the second field the information theoretic bound on the number of bits required, B_ν, as defined in (1). Since $B_\nu \geq \lceil \lg \sigma \rceil$ the complete pointer[4] takes at most twice the information theoretic bound on the number of bits, B_ν. On the other hand, the size of an index is bounded using an expression similar to (3). Subsequently, this, together with Lemma 2, also limits the size of dense second level buckets to be within a constant factor of the information theoretic necessary number of bits. This, in turn, limits the size of a complete main structure to $O(B)$ bits.

It remains to compute the size of the table of small ranges. There are 2^σ entries in the table and each of the entries is σ bits wide, where $\sigma = 4(\log_\lambda^{(2)} M)^2$. This gives us the total size of the table

$$T_\sigma = \sigma \cdot 2^\sigma = o(B) \ . \tag{6}$$

Moreover, this also limits the size of the whole structure to $O(B)$ bits and, hence, proves the theorem:

Theorem 3. *There is an algorithm which solves the static membership problem in $O(1)$ time using data structure of size $O(B)$ bits.*

Let us note that the constants in order notation are relatively small: 7 for time – 2 per level in recursion and a final probe in the table of small ranges – and certainly less than 2 for space. For fine tuning of the structure one can adjust the constants ϵ, λ, and α mentioned in Table 1.

---

[4] Note, that the size of a pointer depends on the number of elements that fall into the bucket, and the more elements there are in the bucket the bigger the pointer is.

4 A Dynamic Version

The solution presented in the previous section deals with a fixed set. It is natural to ask how we can incorporate updates while maintaining the time and space bound.

Our dynamic solution retains the space bound of $O(B)$ bits, but the time per operation of $O(1)$ is an expected bound with a high probability. The solution uses the standard technique of "doubling". The core of this method is to maintain, at any given time, the most appropriate data structure and when the situation changes, to switch to some other better structure. Moreover, the new structure is not built when a certain threshold is reached, but *smoothly* through a sufficient number of preceding operations. Since the additional work due to build-up adds only a constant amount of work per operation, this retains our $O(1)$ time bound.

It remains to be shown that we can apply the doubling technique to our static structure. The structure employs a number of different substructures each of which supports the smooth version of doubling technique including hashing (cf. dynamic perfect hashing [1, 3]) and construction of table of small ranges (when appropriate extend the table by doubling the range).

However, there is a slight memory management problem with indexing. Namely, when the size of a bucket remains the same, but the size of one of its sub-buckets grows to the level that it has to be doubled, we have to find space for the outgrown sub-bucket inside the big bucket.[5] The obvious solution is to allocate more space when the big bucket is created and then copy the outgrown sub-bucket into the first sufficiently big empty slot. Since the copying can be done using a smooth version of doubling technique it does not affect our time and space bounds.

Still, we can construct a case when this technique requires initial allocation of more than $O(B)$ bits of space for the big bucket. To avoid such a situation we use a background daemon that squeezes sub-bucket together. Roughly, in time proportional to the size of a big bucket it goes through all sub-buckets squeezes out all empty slots and compresses sub-buckets together. It is not hard to see that the daemon does not require more than constant additional time per operation, which leaves total time per operation at $O(1)$ worst case with the high probability and total space at $O(B)$ bits.

5 Two "Natural" Examples

There are many situations in dealing with a subset chosen from the bounded universe, in which the size of the subset is relatively big but not big enough to use a bit map representation. Here we mention only two such examples. The first one is a set of all primes smaller then some number M, for which it is known that there are approximately $\frac{M}{\ln M}$ such primes. We pretend that the set of primes is random and that we are to store primes in some kind of structure in order to allow quick answering on a query whether given number is prime. Clearly, we could use some kind of compression (e.g. in a bit map representation we could leave out all even numbers, sieve more carefully etc.), but for the purpose of this example we will not do so.

[5] This happens when, for example, elements from one sub-bucket are shuffled into another sub-bucket.

The second example we consider is population of Canada and Social Insurance Numbers (S.I.N.'s) allocated to each individual person. Canada has approximately 28 million people and each person has a 9 digit Social Insurance Number. One may want to determine whether or not a given number is allocated. This query is in fact a membership query in the universe of size $M = 10^9$ with a subset of size $N = 28 \cdot 10^6$.

In both of these examples N is approximately $\frac{M}{\log_\lambda M}$ and we can use the method presented in Subsect. 3.1 directly. Furthermore, no special features of data are used which makes our space calculations slightly pessimistic. Still, as is typically the case in applying complex data structures to real world data, some tuning is very helpful. In the original version we allocated B bits of memory for the index, but in our examples we allocated ζB bits for some constant ζ.

The perfect function used for the buckets is the one described in [8]. Using an argument similar to that of Lemma 2, we observe that the worst case distribution occurs when all buckets are equally full and, therefore, we can assume that in each bucket there are $\frac{N}{p}$ elements.

In total we use ζB bits of memory for the index, while for buckets we use $N \lg \frac{M}{p}$ bits to list all elements, $p \lg \frac{N}{p}$ bits to describe the number of elements in individual buckets, and $\lg^{(2)} \frac{M}{p}$ bits to describe the range of buckets. Table 2 gives the sizes of data structures for both examples for the standard approaches using a hash function or bit map, and the size of our structure for $\zeta = 0.29$. The table also includes the size of the universe, the size of the subset we are dealing with, and the information theoretic lower bound on the number of necessary bits which is computed using (2).

Table 2. Space usage for subsets of primes and SINs for different data structures.

Example	M	N	B	ours	hash	bitmap
primes	1.0 2^{32}	1.4 2^{27}	1.6 2^{29}	1.9 2^{30}	1.4 2^{32}	1.0 2^{32}
SINs	1.9 2^{29}	1.7 2^{24}	1.1 2^{27}	1.2 2^{28}	1.6 2^{29}	1.8 2^{29}

6 Discussion and Conclusions

In this paper we have presented a solution to a static membership problem. Our solution answers queries in constant time and uses space within a small constant factor of the minimum required by the information theoretic lower bound.

The data structure consists of three major sub-structures which are used in different ranges depending on the ratio of the size of the subset N to universe M. When the size of the subset is at most $M^{1-\epsilon}$ for some fixed $0 \leq \epsilon < 1$ we use perfect hash tables; when the size of the subset is at least αM for some fixed $\alpha \leq \frac{1}{2}$ we use bit map; and when the size of the subset is in the range between these two values we use recursive splitting. The depth of the recursion is bounded by the use of *word-size truncation* and in our case it is 2. We also addressed the dynamic problem and proposed the solution based on a standard doubling technique.

Acknowledgments

We thank Martin Dietzfelbinger for several very helpful comments including pointing out the relevance of non-oblivious hashing to our work. We also thank to anonymous referee for a suggestion of a direct proof of Lemma 2.

References

1. M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial hash functions are reliable. In *Proceedings 19*$^{\text{th}}$ *International Colloquium on Automata, Languages and Programming*, volume 623 of *Lecture Notes in Computer Science*, pages 235–246. Springer-Verlag, 1992.
2. M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R.E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. In *29*$^{\text{th}}$ *IEEE Symposium on Foundations of Computer Science*, pages 524–531, 1988.
3. M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proceedings 17*$^{\text{th}}$ *International Colloquium on Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 6–19. Springer-Verlag, 1990.
4. P. Elias. Efficient storage retrieval by content and address of static files. *Journal of the ACM*, 21(2):246–260, April 1974.
5. P. Elias and R.A. Flower. The complexity of some simple retrieval problems. *Journal of the ACM*, 22(3):367–379, July 1975.
6. P. van Emde Boas. Machine models and simulations. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, chapter 1, pages 1 – 66. Elsevier, Amsterdam, Holland, 1990.
7. A. Fiat and M. Naor. Implicit $O(1)$ probe search. *SIAM Journal on Computing*, 22(1):1 – 10, 1993.
8. A. Fiat, M. Naor, J.P. Schmidt, and A. Siegel. Nonoblivious hashing. *Journal of the ACM*, 39(4):764–782, October 1992.
9. M.L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, July 1984.
10. M.L. Fredman and M.E. Saks. The cell probe complexity of dynamic data structures. In *21*$^{\text{st}}$ *ACM Symposium on Theory of Computing*, pages 345–354, Seattle, Washington, 1989.
11. H.N. Gabow and R.E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30:209–221, 1985.
12. T. Hagerup, K. Mehlhorn, and J.I. Munro. Optimal algorithms for generating discrete random variables with changing distributions. In *Proceedings 20*$^{\text{th}}$ *International Colloquium on Automata, Languages and Programming*, volume 700 of *Lecture Notes in Computer Science*, pages 253 – 264. Springer-Verlag, 1993.
13. P.B. Miltersen. The bit probe complexity measure revisited. In *Proceedings 10*$^{\text{th}}$ *Symposium on Theoretical Aspects of Computer Science*, volume 665 of *Lecture Notes in Computer Science*, pages 662–671. Springer-Verlag, 1993.
14. R.E. Tarjan and A.C. Yao. Storing a sparse table. *Communications of the ACM*, 22(11):606–611, November 1979.
15. A.C.-C. Yao. Should tables be sorted? *Journal of the ACM*, 28(3):614–628, July 1981.