Membership in Constant Time and Almost-Minimum Space

Article *in* SIAM Journal on Computing · January 1999

DOI: 10.1007/BFb0049398 · Source: DBLP

2 authors:

Andrej Brodnik
University of Ljubljana
117 PUBLICATIONS **1,636** CITATIONS

SEE PROFILE

J. Ian Munro
University of Waterloo
302 PUBLICATIONS **8,237** CITATIONS

SEE PROFILE

SIAM J. COMPUT.
Vol. 28, No. 5, pp. 1627–1640

MEMBERSHIP IN CONSTANT TIME AND ALMOST-MINIMUM SPACE*

ANDREJ BRODNIK† AND J. IAN MUNRO‡

Abstract. This paper deals with the problem of storing a subset of elements from the bounded universe $\mathcal{M} = \{0, \ldots, M-1\}$ so that membership queries can be performed efficiently. In particular, we introduce a data structure to represent a subset of N elements of $\mathcal{M}$ in a number of bits close to the information-theoretic minimum, $B = \left\lceil \lg \binom{M}{N} \right\rceil$, and use the structure to answer membership queries in constant time.

Key words. information retrieval, search strategy, data structures, minimum space, dictionary problem, efficient algorithms hashing, lower bound

AMS subject classifications. 68P05, 68P10, 68Q20

PII. S0097539795294165

1. Introduction.

A basic problem in computing is to store a finite set of elements so that one can quickly determine whether or not a query element is a *member* of this set. In this paper we study a version of the problem in which elements are drawn from the bounded universe $\mathcal{M} = \{0, \ldots, M-1\}$ using an extended random access machine (RAM) model that permits constant-time arithmetic and Boolean bitwise operations on these elements. Such a very realistic model enables us to decrease the space needed to store a set of N elements almost to the information-theoretic minimum of $B = \lceil \lg \binom{M}{N} \rceil$ bits, while answering queries in constant time.

Fich and Miltersen [12] have shown that, under a RAM model whose instruction set does not include division, $\Omega(\log N)$ operations are necessary to answer a membership query if the size of a data structure is at most M/N^{ϵ} words of $\lceil \lg M \rceil$ bits each. Thus a sorted array is optimal in that context. Our model includes integer division along with the other standard operations in its instruction set. This permits us to use perfect hash tables (functions) and bitmaps, both of which have constant-time worst-case behavior. However, hash tables generally require that key values be stored explicitly, and so are succinct only when relatively few elements are present. On the other hand, a bitmap is succinct only if about half of the elements are present. In this paper we focus primarily on the range in which N is at least M^{ϵ}, but still $o(M)$, with the goal of introducing a data structure whose size is within a lower-order term of the minimum.

In general terms, our basic approach is to use either perfect hashing or a bitmap whenever one of them achieves the optimum space bound; otherwise we split the

*Received by the editors November 5, 1995; accepted for publication (in revised form) April 10, 1998; published electronically May 6, 1999. This work was supported in part by the Natural Science and Engineering Research Council of Canada under grant A-8237 and the Information Technology Research Centre of Ontario and was done while the first author was a graduate student at the University of Waterloo. Some of the results of this work were announced in preliminary form, in *Membership in constant time and minimum space*, in Proceedings, 2nd European Symposium on Algorithms, Lecture Notes in Comput. Sci. 855, Springer-Verlag, Berlin, New York, 1994, pp. 72–81.
http://www.siam.org/journals/sicomp/28-5/29416.html

†Department of Theoretical Computer Science, Institute of Mathematics, Physics, and Mechanics, University of Ljubljana, Jadranska 11, 1111 Ljubljana, Slovenia, and Department of Computer Science, Luleå Technical University, SE-971 87 Luleå, Sweden (andrej.Brodnik@IMFM.Uni-Lj.SI).

‡Department of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada (imunro@uwaterloo.ca).

universe into subranges of equal size. We discover that, after a couple of careful iterations of this splitting, the subranges are small enough so that succinct indices into a single table of all possible configurations of these subranges (*table of small ranges*) permit the encoding in the minimal space bound. This is an example of what we call *word-size truncated recursion* (cf. [15, 16]). That is, the recursion continues only to a level of "small enough" subproblems, at which point indexing into a table of all solutions suffices. We can do this because at this level a single word in the machine model is large enough to encode a complete solution to each of these small problems.

We proceed with definitions, notation, and background literature. In section 3 we present a constant-time solution with space bound within a small constant factor of the minimum required. The solution has the merit of providing a reasonably practical implementation, and can be tuned to specific problem sizes as is illustrated in giving the space requirements for two specific examples. In section 4 the solution is further tuned to achieve the asymptotic space bound of $B + o(B)$. The results of sections 3 and 4 are extended in section 5 to the dynamic case.

2. Notation, definitions, and background.

2.1. The problem. We use lg to denote the logarithm base 2 and ln to denote the natural logarithm. $\lg^{(i)}$ indicates lg applied i times and $\lg^*$ indicates the number of times lg can be applied before reducing the parameter to at most 1.

DEFINITION 2.1. *Given a universe $\mathcal{M} = \{0, 1, \ldots, M-1\}$ with an arbitrary subset $\mathcal{N} = \{e_0, e_1, \ldots, e_{N-1}\}$, where N and M are known, the* static membership problem *is to determine whether a query value $x \in \mathcal{M}$ is in $\mathcal{N}$.*

This problem has an obvious dynamic extension leading to the following definition.

DEFINITION 2.2. *The* dynamic membership problem *is the static membership problem extended by two operations: insertion of an element $x \in \mathcal{M}$ into $\mathcal{N}$ (if it is not already in $\mathcal{N}$) and deletion of x from a set $\mathcal{N}$ (if it is in $\mathcal{N}$).*

Since solving either problem for $\mathcal{N}$ trivially gives a solution for $\overline{\mathcal{N}}$, we assume $0 \le N \le M/2$.

Our model of computation is an extended version of the RAM machine model (cf. [1]; see also MBRAM in [9]). Memory consists of words of $m = \lceil \lg M \rceil$ bits, which means that one memory register (word) can be used to represent a single element of $\mathcal{M}$, specify an arbitrary subset of a set of m elements, refer to some portion of the data structure, or have some other role that is an m-bit blend of these. For convenience, we measure space in bits rather than in words. Our word size, then, matches the problem size, and so the model is *transdichotomous* in the sense of Fredman and Willard [14]. The usual operations, including integer multiplication, division, and bitwise Boolean operations, are performed in unit time.

We take as parameters of our problem M and N. Hence,

$$B = \left\lceil \lg \binom{M}{N} \right\rceil \tag{2.1}$$

is an information-theoretic lower bound on the number of bits required to describe any possible subset of N elements chosen from M elements. Since we are interested only in solutions that use $O(B)$ or $B + o(B)$ bits for a data structure, there is no need to pay attention to rounding errors, and so we can omit the ceiling and floor functions.

Using Stirling's approximation for the factorial function and Robbins' refinement for its error term (cf. [20, p. 184]), we compute from (2.1) a lower bound on the

number of bits required,

$$\begin{aligned} B &= \lg \binom{M}{N} = \lg M! - \lg N! - \lg(M-N)! \\ &\approx M \lg M - N \lg N - (M-N)\lg(M-N) \quad \text{error} \le \lg N + \mathrm{O}(1) \\ &= M \lg M - N \lg N \\ &\quad -(M-N)(\lg M + \lg(1 - N/M)) \\ &= N \lg(M/N) - (M-N)\lg(1-N/M). \end{aligned} \tag{2.2}$$

Defining the *relative sparseness* of the set $\mathcal{N}$ as

$$r = M/N \tag{2.3}$$

and observing that $2 \le r \le \infty$, we rewrite the second term of (2.2) as

$$N \le -N((r-1)\lg(1-r^{-1})) \le N/\ln 2 \approx 1.4427\ldots N. \tag{2.4}$$

Thus, for the purposes of much of this work, we can use

$$B \approx N \lg(M/N) \equiv N \lg r \tag{2.5}$$

with an error bounded of $\Theta(N)$ bits as given in (2.4). Note that the error term is positive and hence (2.5) is an underestimate.

An intuitive explanation of (2.5) is that $\mathcal{N}$ is fully described when each element in $\mathcal{N}$ "knows" its successor. Since there are N elements in $\mathcal{N}$, the average distance between them is $r = M/N$; to encode this distance we need $\lg r$ bits. Moreover, it is not hard to argue that the worst case, and indeed the average one, occurs when elements are fairly equally spaced. This is exactly what (2.5) says.

2.2. Some background literature. This paper deals with one of the most heavily studied problems in computing, in a context in which the exact model of computation is critical. Therefore, we suggest [9] and [18] as general background and focus on those papers that most heavily shaped the authors' approach. We address three aspects of the problem: the static and dynamic cases of storing a table with little auxiliary data and the information-theoretic trade-offs. In the first two cases, it is usually assumed that there is enough space to list those keys that are present (in a hash table or similar structure) or to list the answers to all queries (by using a bitmap). Here we deal with the situation in which we cannot always afford the space needed to use either structure directly. Nonetheless, we start with the idea of storing keys and little else.

Yao [24] extended the notion of an implicit data structure [21] to the domain of the bounded universe and addressed the problem of storing the value N and an array of N words, each containing a $\lg M$ bit data item. He showed that if no more information is stored, then there always exists some value of N and subset of size N that requires at least logarithmic search time. Adding almost any storage, however, changes the situation. For example, with one more register ($\lg M$ bits) Yao [24] showed that there exists a constant-time solution for $N \approx M$ or $N \le \frac{1}{4}\sqrt{\lg M}$, while Tarjan and Yao [23] presented a more general $\mathrm{O}(\lg M/\lg N)$ time, $\mathrm{O}(N \lg M)$ bit solution. Fredman, Komlós, and Szemerédi [13, sec. 4] developed a constant-time algorithm with a data structure of $N \lg M$ bits for the portion of the data, plus an additional $\mathrm{O}(N\sqrt{\lg N} + \lg^{(2)} M)$ bits. Fiat et al. [11] decreased the extra bits to

$6 \lg N + 3\lceil \lg^{(2)} M \rceil + O(1)$. Moreover, combining their result with Fiat and Naor's [10] construction of an implicit search scheme for $N = \Omega((\lg M)^p)$, they produced a scheme using fewer than $(1+p)\lceil \lg^{(2)} M \rceil + O(1)$ additional bits.

Mairson [17] took a different approach. He assumed all structures are implicit in Yao's sense and the additional storage represents the complexity of a searching program. Following a similar path, Schmidt and Siegel [22] proved a lower bound of $\Omega(N/(k^2 e^k) + \lg^{(2)} M)$ bits spatial complexity for k-probe oblivious hashing. In particular, for constant-time hashing this gives a spatial complexity of $\Theta(\lg^{(2)} M + N)$ bits.

For the dynamic case, Dietzfelbinger et al. [5] proved an $\Omega(\lg N)$ worst-case lower bound for a realistic class of hashing schemes. In the same paper they also presented a scheme which, using results of [13] and a standard doubling technique, achieved constant amortized expected time per operation. However, the worst-case time per operation (nonamortized) was $\Omega(N)$. Later Dietzfelbinger and Meyer auf der Heide [6] upgraded the scheme and achieved constant worst-case time per operation with a high probability. A similar result was also obtained by Dietzfelbinger et al. [4].

In the data compression technique described by Choueka et al. [3], a bit vector is hierarchically compressed. First, the binary representations of elements stored in the dictionary are split into pieces of equal size. Then the elements with the same value as the most significant piece are put in the same bucket and the technique is recursively applied within each bucket. When the number of elements which fall in the same bucket becomes sufficiently small, the data are stored in a compressed form. The authors experimentally tested their ideas but did not formally analyze them. They claim their result gives a relative improvement of about 40% over similar methods.

An information-theoretic approach was taken by Elias [7] in addressing a more general version of the static membership problem which involved several different types of queries. For these queries he discussed a trade-off 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 $N \lg(M/N) + O(N)$ (using (2.5), $B + o(B)$) bits, which required an average of $(1+\epsilon)\lg N + 2$ bit probes to answer a query. However, in the worst case the method required N bit probes. Elias and Flower [8] further generalized the notion of a query into a database. They defined the set of data and a set of queries and, in a general setting, studied the relation between the size of the data structure and the number of bits probed, given the set of all possible queries. Later, the same arrangement was more rigorously studied by Miltersen [19].

3. Static solution using $O(B)$ space. Our solution breaks down to a number of cases, based on the relative sparseness of $\mathcal{N}$. As noted earlier, we can assume that at most half the elements are present, since the complementary problem could otherwise be solved. We are left with four cases as r ranges between 2 and ∞ (cf. Table 3.1). The crucial dividing point between the sparse and dense cases comes when r is in the range $\Theta(\lg M)$. For purposes of tuning the method, we find it convenient to define this *separation point* in terms of a parameter $\lambda(>1)$, namely,

$$r_{sep} = \log_\lambda M, \tag{3.1}$$

or the size of sets

$$N_{sep} = M/r_{sep} = M/\log_\lambda M. \tag{3.2}$$

TABLE 3.1
Cases considered for the static version of the problem.

Sparseness	Range of r			Range of N			Section
Very sparse	∞	to	M^ϵ	0	to	$M^{1-\epsilon}$	3.1
Moderately sparse	M^ϵ	to	$\log_\lambda M$	$M^{1-\epsilon}$	to	$M/\log_\lambda M$	3.2
Moderately dense	$\log_\lambda M$	to	$1/\alpha$	$M/\log_\lambda M$	to	αM	3.3
Very dense	$1/\alpha$	to	2	αM	to	$M/2$	3.1

The very sparse and very dense cases are rather straightforward, though their boundaries with the more difficult moderately sparse and moderately dense cases are subject to tuning as well. After handling these easy cases, we address the moderately sparse case and subsequently extend its solution to handle the moderately dense.

3.1. Very sparse and very dense cases. When $\mathcal{N}$ is very dense, i.e., $N \geq \alpha M$ for $0 < \alpha \leq 1/2$, we can afford to use a bitmap of size $M = \Theta(B)$ to represent it. When $\mathcal{N}$ is very sparse, i.e., $N \leq M^{1-\epsilon}$ for $0 < \epsilon \leq 1$, we are allowed $\Theta(N \log M)$ bits which is enough to list all the elements of $\mathcal{N}$. For $N \leq c = \mathrm{O}(1)$ we simply list them. Beyond this we use a perfect hashing function of some form (cf. [10, 11, 13]). Note that all of these structures allow us to answer a membership query in constant time and are, indeed, reasonably practical methods.

3.2. Moderately sparse case—indexing. The range in which $r \approx r_{sep}$ typifies the case in which neither the straightforward listing of the elements nor a bitmap minimizes the storage requirements. In this range, the $N \lg M$ bits needed to list all elements is of the same order as the M for a bitmap, but $B = \Theta(N \lg^{(2)} M)$. Indeed, thoughts of this specific case lead not only to a solution to the entire moderately sparse range, but also to the first step in the solution for the moderately dense case.

LEMMA 3.1. *If $N \leq N_{sep} = M/\log_\lambda M$ for $\lambda > 1$, then there is an algorithm which answers a membership query in constant time using an $\mathrm{O}(B)$ bit data structure.*

Proof. The idea is to split the universe, $\mathcal{M}$, into p buckets, where p is as large as we can make it without exceeding our space constraints. The data falling into individual buckets are then organized using perfect hashing. The buckets cover contiguous ranges of equal sizes, $M_1 = \lfloor M/p \rfloor$, so that a key $x \in \mathcal{M}$ falls into bucket $\lfloor x/M_1 \rfloor$. To reach individual buckets, we index through an array of pointers.

Each pointer occupies $\lceil \lg M \rceil$ bits. Hence, the total size of the *index* (the array of pointers) is $p \lceil \lg M \rceil$ bits. We store all elements that fall in the same bucket in a perfect hash table [10, 11, 13] for that bucket. Since the ranges of all buckets are of equal size, the space required to describe each element in a hash table is $\lceil \lg(M/p) \rceil$ bits, and so to describe all elements in all buckets we require only $N \lceil \lg(M/p) \rceil$ bits. We also need some extra space to describe individual hash tables. If we use the method of Fiat et al. [11], the additional space for bucket i is bounded by $6\lceil \lg N_i \rceil + 3\lceil \lg^{(2)} M_1 \rceil + \mathrm{O}(1)$, where N_i is the number of elements in bucket i. Thus, the additional space to describe all hash functions is bounded by $p(6 \lg N + 3 \lg^{(2)} M + \mathrm{O}(1))$. Putting the pieces together, we get the expression for the size of the structure:

$$S = p \lg M + N \lg(M/p) + p(6 \lg N + 3 \lg^{(2)} M + \mathrm{O}(1)).$$

Choosing p to minimize this value leads to a rather complex formula. However, a simple approximation is adequate, and so we take

$$p = N/\lg M. \tag{3.3}$$

This gives

$$\begin{aligned} S &= N + N(\lg M/N + \lg^{(2)} M) + \\ &\quad N(6\lg N + 3\lg^{(2)} M + \mathrm{O}(1))/\lg M \qquad \text{by (2.3)} \\ &\le N\lg r + (N\lg r)(\lg^{(2)} M/\lg r) + N + \\ &\quad N(6\lg N + 3\lg^{(2)} M + \mathrm{O}(1))/\lg M \qquad \text{by (2.5)} \\ &= B + B\lg^{(2)} M/\lg r + \mathrm{o}(B). \end{aligned} \tag{3.4}$$

Hence, for a moderately sparse subset, i.e., $r \ge r_{sep}$, the size of the structure is $\mathrm{O}(B)$ bits. It is also easy to see that the structure permits constant-time search. □

Note that if $r_{sep} \ge \lg M$ (i.e., in (3.1) $\lambda < 2$), the lead term of (3.4) is less than $2B$.

3.3. Moderately dense case—word-size truncated recursion. In this section we consider sets of size N (or sparseness r) in the range

$$\begin{array}{rccccccc} N_{sep} = M/\log_\lambda M & \le & N & \le & \alpha M & \le & M/2, \\ r_{sep} = \log_\lambda M & \ge & r & \ge & 1/\alpha & \ge & 2. \end{array} \tag{3.5}$$

For such *moderately dense* $\mathcal{N}$ we apply the technique of Lemma 3.1—that is, split the universe $\mathcal{M}$ into equal-range buckets. However, this time the buckets remain too full 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. If its relative sparseness falls in the range defined by (3.5) (with respect to the size of its smaller universe) we recursively split it.

Such a straightforward strategy leads, in the worst case, to a $\Theta(\lg^* M)$ level structure and therefore to a $\Theta(\lg^* M)$ search time. However, we observe that at each level the number of buckets with the same range increases and ultimately there must be so many small subsets that not all can be different. Therefore we build a table of all possible subsets of universes of size up to a certain threshold. This *table of small ranges* (TSR) allows replacement of buckets in the main structure by pointers (indices) into the table. Although the approach is not new (cf. [15, 16]), it does not appear to have been given a name. We refer to the technique as *word-size truncated recursion*. In our structure the truncation occurs after two splittings. In fact, because all our second-level buckets have the same range, our TSR consists of all possible subsets of only a single small universe. In the rest of this section we give a detailed description of the structure and its analysis.

On the first split we partition the universe into

$$p = N_{sep}/\lg M = M/(\log_\lambda M \lg M) \tag{3.6}$$

buckets, each of which has a range $M_1 = M/p$. At the second level we have, again, relatively sparse and dense buckets which now separate at the relative sparseness

$$r'_{sep} = \log_\lambda M_1 = \log_\lambda(M/p) = \mathrm{O}(\lg^{(2)} M). \tag{3.7}$$

For sparse buckets we apply the solution of section 3.2, and for very dense ones with more than the fraction α of their elements present we use a bitmap. For the moderately dense buckets, with relative sparseness within the range defined in (3.5), we reapply the splitting. However, this time the number of buckets is (cf. (3.6))

$$p_1 = M_1/(r'_{sep}\lg M_1), \tag{3.8}$$

so that each of these smaller buckets has the same range,

$$M_2 = M_1/p_1 = \mathrm{O}((\lg^{(2)} M)^2), \tag{3.9}$$

because $\lg M_1 = \mathrm{O}(\lg^{(2)} M)$.

At this point we use the TSR. This table consists of bitmap representations of all subsets of the universe of size M_2. Thus we can replace buckets in the main structure with "indices" (pointers of varying sizes) into the table. We order the table first according to the number of elements in the subset and then lexicographically. We store a pointer in the TSR as a record consisting of two fields: ν, the number of elements in the bucket, which takes $\lceil \lg M_2 \rceil$ bits; and β, the rank of this bucket with respect to the lexicographic order among all buckets containing ν elements. To store β, by (2.1), takes $B_\nu = \lceil \lg \binom{M_2}{\nu} \rceil$ bits. The actual position (index) of the corresponding bitmap of the bucket in the TSR is thus

$$\sum_{i=1}^{\nu-1} \binom{M_2}{i} + \beta - 1. \tag{3.10}$$

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

This concludes the description of the data structure also presented in Algorithm 3.1. As demonstrated in Algorithm 3.2, the data structure allows constant-time membership queries, but it remains to be seen how much space it occupies. The algorithm uses functions `LookUpBM`—look up bitmap; `FindOL`—find in ordered list; and `FindHT`—find in hash table, whose descriptions are omitted. However, their particular implementation suggests the constants c, ϵ, λ, and α used for the fine tuning of Algorithm 3.2.

ALGORITHM 3.1. DATA STRUCTURE FOR THE SOLUTION OF THE STATIC PROBLEM.

```
TYPE
tCases=
  (eEmpty,                                              (* N = 0 *)
   eVerySparse1,                                        (* 0 < N ≤ c *)
   eVerySparse2,                                        (* c < N ≤ M^(1-ε) *)
   eModeratelySparse,                                   (* M^(1-ε) < N ≤ M / log_λ M *)
   eModeratelyDense,                                    (* M / log_λ M < N ≤ α M *)
   eVeryDense);                                         (* α M < N < M/2 *)
tSet= RECORD CASE BOOLEAN OF
TRUE:                                                   (* current universe is at most M2 *)
  ν, β;
FALSE:                                                  (* general case *)
 N;                                                     (* size of the set *)
  CASE tCases OF
   eEmpty: ;                                            (* nothing *)
   eVerySparse1:                                        (* (un)ordered list *)
     list: ARRAY [] OF tElement;
   eVerySparse2:                                        (* hash table *)
     hashTable: tHashTable;
   eModeratelySparse:                                   (* indexing *)
     index: ARRAY [] OF ^tHashTable;
   eModeratelyDense:                                    (* (word-size truncated) recursion *)
     subset: ARRAY [] OF ^tSet;
   eVeryDense:                                          (* bit map *)
```

```
    bitmap: ARRAY [] OF BOOLEAN;
  END;
END;
```

ALGORITHM 3.2. MEMBERSHIP QUERY IF elt IS IN $\mathcal{N} \subseteq \mathcal{M}$, WHERE $|\mathcal{M}| = M$.

```
PROCEDURE Member (M, 𝒩, elt): BOOLEAN;
  IF M ≤ M₂ THEN
    pointer:= binomials[𝒩.ν] + 𝒩.β -1;                (* pointer by (3.10), *)
    RETURN LookUpBM (TSR[pointer], elt);             (* bit map from the TSR *)
  ELSE
    IF 𝒩.N ≥ M/2 THEN negate:= FALSE; N:= 𝒩.N
    ELSE negate:= TRUE; N:= M-𝒩.N END;
    CASE N OF
                                                     (* How sparse the set 𝒩 is: *)
    N = 0: answer:= FALSE                                    (* EMPTY SET; *)
    N ≤ c: answer:= FindOL (𝒩.list, elt);                (* VERY SPARSE SET; *)
    N ≤ M^(1-ε): answer:= FindHT (𝒩.hashTable, elt);   (* STILL VERY SPARSE SET;
*)
    N ≤ M/log_λ(M):                                      (* MODERATELY SPARSE SET: *)
      M₁:= Floor ((M/N)*lg(M));    (* split into buckets of range M₁ by (3.3), *)
      answer:= FindHT (𝒩.index[elt DIV M₁], elt MOD M₁);  (* search bucket; *)
    N ≤ α*M:                                             (* MODERATELY DENSE SET: *)
      M₁:= Floor (log_λ(M)*lg(M));          (* split into subuniverses of size M₁ by
(3.6), *)
      answer:=                                           (* and recursively search it; *)
        Member (M₁, 𝒩.subset[elt DIV M₁]^, elt MOD M₁)
                                                         (* VERY DENSE SET; *)
    ENDCASE;
    IF negate THEN RETURN NOT answer
    ELSE RETURN answer ENDIF;
  ENDIF;
END Member;
```

In analyzing the space requirements, we are interested only in moderately dense subsets, as otherwise we use the structures of sections 3.1 and 3.2. First we analyze the main structure, i.e., the data structure without a TSR, and begin with the following lemma.

LEMMA 3.2. *Suppose we are given a subset of N elements from the universe M, and B is as defined in* (2.1). *If this universe is split into p buckets of ranges of sizes M_i containing N_i elements, respectively (now, using* (2.1), $B_i = \lceil \lg \binom{M_i}{N_i} \rceil$ *for $1 < i \le p$), then $B + p > \sum_{i=1}^{p} B_i$.*

Proof. If $\sum_{i=1}^{p} M_i = M$ and $\sum_{i=1}^{p} N_i = N$, we know that $0 < \prod_{i=1}^{p} \binom{M_i}{N_i} \le \binom{M}{N}$ and therefore $\sum_{i=1}^{p} \lg \binom{M_i}{N_i} \le \lg \binom{M}{N}$. On the other hand, from (2.1) we have $B_i = \lceil \lg \binom{M_i}{N_i} \rceil$ and therefore $B_i - 1 < \lg \binom{M_i}{N_i} \le B_i$. This gives us $\sum_{i=1}^{p} (B_i - 1) < B$ and finally $B + p > \sum_{i=1}^{p} B_i$. □

In simpler terms, Lemma 3.2 proves that if subbuckets are encoded at close to the information-theoretic bound, then the complete bucket also uses an amount of space close to the information-theoretic minimum, provided that the number of buckets is small enough ($p = o(B)$) and that the index does not take too much space.

We analyze the main structure itself from the top to the bottom. The first-level index consists of p pointers of $\lg M$ bits each. Therefore, using (3.6) and (3.2), the size of that complete index is

$$p \lg M = M/\log_\lambda M = N_{sep} = o(B). \tag{3.11}$$

For the sparse buckets on the second level we use the solution presented in section 3.2. For the very dense buckets ($r \leq 1/\alpha$) we use a bitmap. 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 moderately dense buckets, then, using Lemma 3.2 and (3.11), the complete main structure uses $O(B)$ bits. Note that we can apply Lemma 3.2 freely because the number of buckets, p, is $o(B)$.

Next we determine the size of the encoding of the second-level moderately dense buckets, that is, the encoding of buckets with sparseness in the range of (3.5). For this purpose we first consider the size of bottom-level pointers (indices) into the TSR. As mentioned, the pointers are records consisting of two fields. The first field, ν (number of elements in the bucket), occupies $\lceil \lg M_2 \rceil$ bits, and the second field takes at most B_ν. Since $B_\nu \geq \lceil \lg M_2 \rceil$, the complete pointer[1] 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 (3.11). Subsequently, this, together with Lemma 3.2, also limits the size of space needed to store representation of moderately dense buckets on the second level to be within a constant factor of the information-theoretic bound. This, in turn, limits the size of the complete main structure to $O(B)$ bits.

It remains to compute the size of the TSR. There are 2^{M_2} entries in the table and each of the entries is M_2 bits wide. By (3.9) $M_2 = O((\lg^{(2)} M)^2)$. This gives us the total size of the table

$$\begin{aligned} M_2 2^{M_2} &= O((\lg^{(2)} M)^2 (\log M)^{\lg^{(2)} M}) \\ &= O((\lg \lg M \lg M)(\lg^{(2)} M (\lg M)^{1+\lg^{(2)} M})) \\ &= o(\lg r_{sep} M / r_{sep}) \qquad \text{by (3.1)} \\ &= o(N_{sep} \lg r_{sep}) = o(B). \end{aligned} \tag{3.12}$$

Finally, this also bounds the size of the whole structure to $O(B)$ bits and hence in conjunction with Lemma 3.1 proves the following theorem.

THEOREM 3.3. *There is an algorithm which solves the static membership problem in $O(1)$ time using a data structure of size $O(B)$ bits.*

Note the constants in order notation of Theorem 3.3 are relatively small. Algorithm 3.2 performs at most two recursive calls of `Member` and eight probes of the data structure:

- two probes in the first call of `Member`: one to get N and one to compute M_1;
- two probes in the second call of `Member`: same as above; and
- four probes in the last call of `Member`: two probes to get the number of elements in the bucket, ν, and the lexicographic order of the bucket, β; the next probe to get the sum in (3.10) by lookup in table `binomials`; and the final probe into the `TSR`.

[1] Note that the size of a pointer depends on the number of elements that fall into the bucket.

TABLE 3.2
Space usage for sets of primes and SINs for various data structures.

Example	M	N	B	ours	hash	bit map
Primes	$1.0 \cdot 2^{32}$	$1.4 \cdot 2^{27}$	$1.6 \cdot 2^{29}$	$1.9 \cdot 2^{30}$	$1.4 \cdot 2^{32}$	$1.0 \cdot 2^{32}$
SINs	$1.9 \cdot 2^{29}$	$1.7 \cdot 2^{24}$	$1.1 \cdot 2^{27}$	$1.2 \cdot 2^{28}$	$1.6 \cdot 2^{29}$	$1.9 \cdot 2^{29}$

If perfect hashing is used in one of the steps, the number of probes remains comparable.

It is easy to see that by setting $\alpha = 1/2$ and $\epsilon = 1$, thereby eliminating the two extreme cases, at most $2B + o(B)$ bits are required for the structure. In the next section we reduce this bound to $B + o(B)$ bits while retaining the constant query time. However, in practice the $o(B)$ term can be as much of a concern as the factor of 2. Indeed the reader of the next section is justified in questioning the notion of $(\lg^{(2)} M_1 - 5)/6$ becoming large in practice. We therefore first illustrate the tuning of the method to specific values of M and N with two examples.

The first is the set of primes that fit in a single 32-bit word, so $M = 2^{32}$ and N is of size approximately $M/\ln M$. We pretend that the set of primes is random and that we are to store them in a structure to support the query of whether a given number is prime. Clearly, we could use some kind of compression (e.g., implicitly omit the even numbers or sieve more carefully), but for the purpose of this example we will not do so. In the second example we consider Canadian Social Insurance Numbers (SINs) allocated to each individual. Canada has approximately 28 million people and each person has a nine-digit SIN. 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 $N = 28 \cdot 10^6$. One of the digits is a check digit, but we will ignore this issue.

Both examples deal with moderately sparse sets and we can use the method of section 3.2 directly, using buckets and a perfect hashing function described in [11]. On the other hand, no special features of the data are used, which makes our space calculations slightly pessimistic. Using an argument similar to that of Lemma 3.2, we observe that the worst-case distribution occurs when all buckets are equally sparse, and therefore we can assume that in each bucket there are N/p elements. Table 3.2 contains the sizes of data structures for both examples comparing a hash function, a bitmap, and a tuned version of our structure (computed from (3.4)) with the information-theoretic bound.

4. Static solution using $B + o(B)$ space. We now return to the tuning of our technique for asymptotically large sets, achieving a $B + o(B)$ bit space bound.

First, we observe that for very dense sets ($r \leq 1/\alpha$) we cannot afford to use a bitmap because it always takes $B + \Theta(B)$ bits. Similarly we cannot afford to use hash tables for very sparse sets (i.e., $r \geq M^{1-\epsilon}$). Therefore, we categorize sets *only* as sparse or dense (and not moderately dense). The key point in decreasing the space bound, however, is redefining the separation point between sparse and dense set to

$$r_{sep} = (\lg M)^{\lg^{(2)} M}, \tag{4.1}$$

and so

$$N_{sep} = M/(\lg M)^{\lg^{(2)} M}. \tag{4.2}$$

While we intend B to indicate the exact value from (2.1), for sparse sets we can

still use the approximation $N \lg r$ from (2.5) since the error in (2.4) is bounded by $\Theta(N) = o(B)$.

4.1. Sparse subsets. Again, sparse subsets are those whose relative sparseness is greater than r_{sep}. For such subsets we *always* apply the two-level indexing of section 3.2. All equations from section 3.2, and in particular (3.4), still hold. However, the second term of (3.4) can be tightened to $o(B)$, because now the relative sparseness, r, is at least r_{sep} defined in (4.1). This proves the following lemma.

LEMMA 4.1. *If $\infty > r \geq r_{sep}$ as in* (4.1) *(i.e., $N \leq N_{sep}$ as in* (4.2)*), then there is an algorithm to answer membership queries in constant time using a $B + o(B)$ bit data structure.*

4.2. Dense subsets. Dense subsets are treated in the same way as moderately dense subsets were treated in section 3.3. Thus most of the analysis can be taken from that section with the appropriate changes of r_{sep} (cf. (3.1)) and r'_{sep} (cf. (3.7)). To compute the size of the main structure, we first bound the size of pointers into the TSR. Recall that each pointer consists of two fields: the number of elements in the bucket, ν, and the rank (in lexicographic order) of the bucket in question among all buckets with ν elements, β. Although the number of bits needed to describe ν can be as large as the information-theoretic minimum for some buckets, this is not true on the average. By Lemma 3.2, all pointers together occupy no more than $B + o(B)$ bits, where B is the exact one from (2.1). Furthermore, the indices are small enough so that all of them together occupy $o(B)$ bits (cf. (3.11)). As a result we conclude that the main structure occupies $B + o(B)$ bits of space. It remains to bound the size of the TSR at the redefined separation points.

With the redefinition of r_{sep} in (4.1), (3.6) now gives

$$p = M/(r_{sep} \lg M) = M/(\lg M)^{1+\lg^{(2)} M} \tag{4.3}$$

buckets on the first level. Each of these has a range of

$$M_1 = M/p = r_{sep} \lg M = (\lg M)^{1+\lg^{(2)} M}. \tag{4.4}$$

To simplify further analysis we set the redefined separation point between first-level sparse and dense buckets (cf. (3.7)) to

$$r'_{sep} = (\lg M_1)^{(\lg^{(2)} M_1 - 5)/6}, \tag{4.5}$$

which is adequate to keep the space requirement of the sparse buckets to $o(B)$ (cf. (3.4)). The position of this separation point r'_{sep} is further bounded by

$$\begin{aligned}
r'_{sep} &= (\lg(r_{sep} \lg M))^{(\lg^{(2)}(r_{sep} \lg M)-5)/6} && \text{using (4.4)} \\
&< (2 \lg r_{sep})^{(\lg(2 \lg r_{sep})-5)/6} && \text{since } r_{sep} > \lg M \text{ by (4.1)} \\
&< (2(\lg^{(2)} M)^2)^{(\lg((\lg^{(2)} M)^2)-4)/6} && \text{again using (4.1)} \\
&< ((\lg^{(2)} M)^3)^{(} \lg^{(3)} M - 2)/3 && \text{since } 2 < \lg^{(2)} M \\
&< (\lg^{(2)} M)^{\lg^{(3)} M - 1}/3 && \text{since } (\lg^{(2)} M)^{-1} < 1/3. \qquad (4.6)
\end{aligned}$$

Next, the first-level dense buckets are further split into p_1 (cf. (3.8)) subbuckets, each of range $M_2 = M_1/p_1 = r'_{sep} \lg M_1$. Finally, since M_2 is also the range of buckets in the TSR, the size of the table is

$$
\begin{aligned}
M_2 2^{M_2} &= r'_{sep}(\lg M_1) M_1^{r'_{sep}} \\
&= r'_{sep} \lg(r_{sep} \lg M)(r_{sep} \lg M)^{r'_{sep}} && \text{by (4.4)} \\
&< 2(\lg r_{sep}) r'_{sep} {r_{sep}}^{2r'_{sep}} && \text{since } r_{sep} = (\lg M)^{\omega(1)} \\
&< (\lg r_{sep}) {r_{sep}}^{3r'_{sep}-1} \\
&< \lg r_{sep}((\lg M)^{\lg^{(2)} M})^{(\lg^{(2)} M)^{\lg^{(3)} M - 1}}/r_{sep} && \text{by (4.6) and (4.1)} \\
&< \lg r_{sep}(\lg M)^{(\lg^{(2)} M)^{\lg^{(3)} M}}/r_{sep} \\
&= o(M \lg r_{sep}/r_{sep}) = o(N_{sep} \lg r_{sep}) \\
&= o(B)
\end{aligned}
$$

for $r \le r_{sep}$. This brings us to the main asymptotic result.

THEOREM 4.2. *There is an algorithm which solves the static membership problem in* $O(1)$ *time using data structure of size* $B + o(B)$ *bits.*

Proof. The discussion above dealt only with the space bound. However, since the structure is more or less the same as that of section 3, the time bound can be drawn from Theorem 3.3. □

With Theorem 4.2 we proved only that the second term in space complexity is $o(B)$. In fact, using a very rough estimate from the second term of sparse first-level buckets we get the bound $O(B/\lg^{(3)} M)$. To improve the bound one would have to refine values r_{sep} and r'_{sep}.

5. Dynamic version. There are several options for converting our ideas for a static structure into one that supports insertions and deletions. In the interest of simplicity we sketch and demonstrate only one.

THEOREM 5.1. *There exists a data structure requiring* $O(B)$ *bits which supports searches in constant time and insertions and deletions in constant expected amortized time.*

The basic approach is simple, and we make no attempt here to minimize the space bound other than to retain the $O(B)$ requirement. We use the method of section 3 but substitute dynamic perfect hashing [5] for the static perfect hashing scheme used there.

A key observation is that, given a set of N elements, our structure can be built in expected time $O(N)$ and $O(B)$ space if we use dynamic perfect hashing for the hashing aspect. Indeed, any of our substructures can be built in linear time and in space within a constant factor of that suggested in the preceding section. This also applies to the TSR in that it can be created in time linear in its size.

Like dynamic perfect hashing itself, our dynamic scheme operates in phases. At the beginning of a phase, a structure of N_0 elements is built, but each dynamic perfect hashing structure is given $1 + \kappa$ times as much space as it requires. Here κ is an arbitrary positive constant. In addition the entire space allocated for the structure is increased globally by another factor of $1 + \kappa$.

As insertions and deletions are made, two critical conditions can arise. The number of elements in the table may drop, say to $N_0/(1 + \kappa)$, in which case we obtain a new block of space appropriate for a table of the new, reduced, size. A new table is constructed in the new space and the old table is released. The other condition is that we run out of space either in one of the hash tables or in the structure as a whole. It is unlikely that we will run out of space in a single dynamic perfect hashing structure until a number of updates proportional to its original size is made. However,

with our multilevel structure we could have a large number of insertions fall into the same bucket, which could cause a dynamic perfect hashing structure to overflow after a rather small number of updates relative to the size of the entire structure. If this happens, we simply rebuild this subtable using the extra space allocated globally.

6. Discussion and conclusions. In this paper we have presented a solution to a static membership problem. Our initial solution answers queries in constant time and uses space within a small constant factor of the minimum required by the information-theoretic lower bound. Subsequently, we improved the solution reducing the required amount of space to the information-theoretic lower bound plus a lower-order term. We also addressed the dynamic problem and proposed a solution based on a standard doubling technique.

Data structures used in solutions consist of three major substructures which are used in different ranges depending on the relative sparseness of the set at hand (that is, depending on the ratio between the size of the set and the universe). When the set is relatively sparse we use a perfect hashing; when the set is relatively dense we use a bitmap; and in the range between we use recursive splitting (indexing). The depth of the recursion is bounded by the use of *word-size truncation* and in our case it is 2.

The feasibility of the data structure was addressed through a couple of examples. However, to make the structure more practical one would need to tune the parameters c, ϵ, λ, and α mentioned in Algorithm 3.2. Moreover, for practical purposes it is helpful to increase the depth of recursive splitting to cancel out the effect of a constant hidden in the order notation and, in particular, to decrease the size of the TSR below the information-theoretic minimum defined by N and M at hand. For example, in the case of currently common 64- and 32-bit architectures, the depths can be increased to 4 and 5, respectively.

There are several open problems. One may be able to reduce the space requirement of the dynamic structure to $B+o(B)$ by first reexamining the details of dynamic perfect hashing and reducing its storage requirements to $N + o(N)$ words, assuming the universe is large relative to N. Another intriguing problem is to decrease the second-order term in the space complexity as there is still a substantial gap between our result, $B + O(B/\lg^{(3)} M)$, and the information-theoretic minimum, B. But do we need a more powerful machine model to close this gap?

Acknowledgments. We thank Martin Dietzfelbinger and the referees for many very helpful comments that improved this paper.

REFERENCES

[1] A. Brodnik, *Searching in Constant Time and Minimum Space* (*Minimæ Res Magni Momenti Sunt*), Ph.D. thesis, available as Technical report CS-95-41, University of Waterloo, Waterloo, ON, Canada, 1995.

[2] A. Brodnik and J. I. Munro, *Membership in constant time and minimum space*, in Proceedings, Second European Symposium on Algorithms, Lecture Notes in Comput. Sci. 855, Springer-Verlag, 1994, pp. 72–81.

[3] Y. Choueka, A. Fraenkel, S. Klein, and E. Segal, *Improved hierarchical bit-vector compression in document retrieval systems*, in 9th International ACM SIGIR Conference on Research and Development in Information Retrieval, ACM, 1986, pp. 88–96.

[4] M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger, *Polynomial hash functions are reliable*, in Proceedings, 19th International Colloquium on Automata, Languages and Programming, Lecture Notes in Comput. Sci. 623, Springer-Verlag, 1992, pp. 235–246.

[5] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. Tarjan, *Dynamic perfect hashing: Upper and lower bounds*, SIAM J. Comput., 23 (1994), pp. 738–761.
[6] M. Dietzfelbinger and F. Meyer auf der Heide, *A new universal class of hash functions and dynamic hashing in real time*, in Proceedings, 17th International Colloquium on Automata, Languages and Programming, Lecture Notes in Comput. Sci. 443, Springer-Verlag, New York, 1990, pp. 6–19.
[7] P. Elias, *Efficient storage retrieval by content and address of static files*, J. ACM, 21 (1974), pp. 246–260.
[8] P. Elias and R. Flower, *The complexity of some simple retrieval problems*, J. ACM, 22 (1975), pp. 367–379.
[9] P. van Emde Boas, *Machine models and simulations*, in Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity, J. van Leeuwen, ed., Elsevier, Amsterdam, 1990, pp. 1–66.
[10] A. Fiat and M. Naor, *Implicit $O(1)$ probe search*, SIAM J. Comput., 22 (1993), pp. 1–10.
[11] A. Fiat, M. Naor, J. Schmidt, and A. Siegel, *Nonoblivious hashing*, J. ACM, 39 (1992), pp. 764–782.
[12] F. Fich and P. Miltersen, *Tables should be sorted (on random access machines)*, in Proceedings, Fourth Workshop on Algorithms and Data Structures, Lecture Notes in Comput. Sci. 955, Springer-Verlag, New York, 1995, pp. 482–493.
[13] M. Fredman, J. Komlós, and E. Szemerédi, *Storing a sparse table with $O(1)$ worst case access time*, J. ACM, 31 (1984), pp. 538–544.
[14] M. Fredman and D. Willard., *Trans-dichotomous algorithms for minimum spanning trees and shortest paths*, J. ACM, 31 (1984), pp. 538–544.
[15] H. Gabow and R. Tarjan, *A linear-time algorithm for a special case of disjoint set union*, J. Comput. System Sci. 30 (1985), pp. 209–221.
[16] T. Hagerup, K. Mehlhorn, and J. I. Munro, *Optimal algorithms for generating discrete random variables with changing distributions*, in Proceedings, 20th International Colloquium on Automata, Languages and Programming, Lecture Notes in Comput. Sci. 700, Springer-Verlag, New York, 1993, pp. 253–264.
[17] H. Mairson, *The program complexity of searching a table*, in 24th IEEE Symposium on Foundations of Computer Science, 1983, pp. 40–47.
[18] K. Mehlhorn and A. Tsakalidis, *Data structures*, in Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity, J. van Leeuwen, ed., Elsevier, Amsterdam, The Netherlands, 1990, pp. 301–334.
[19] P. Miltersen, *The bit probe complexity measure revisited*, in Proceedings, 10th Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci. 665, Springer-Verlag, New York, 1993, pp. 662–671.
[20] D. Mitrinović, *Analytic Inequalities*, Grundlehren Math. Wiss. 165, Springer-Verlag, Berlin, 1970.
[21] J. I. Munro and H. Suwanda, *Implicit data structures for fast retrieval and update*, J. Comput. System Sci., 21 (1980), pp. 236–250.
[22] J. Schmidt and A. Siegel, *The spatial complexity of oblivious k-probe hash functions*, SIAM J. Comput., 19 (1990), pp. 775–786.
[23] R. Tarjan and A. Yao, *Storing a sparse table*, Comm. ACM, 22 (1979), pp. 606–611.
[24] A.-C. Yao, *Should tables be sorted?*, J. ACM, 28 (1981), pp. 614–628.