Implementing dictionaries using hashing

Michiel Smid*

June 8, 2004

1 Introduction

These notes are based on Mehlhorn [5].

Many algorithms manipulate sets. For example, a compiler uses a symbol table to keep track of the identifiers that are used in a program. Identifiers are strings of letters and digits. With each identifier, associated information such as type, scope, and address, is stored. As another example, the catalogue of a library maintains, among other things, names of authors. With each author, associated information such as title and ISBN number is stored. Typical operations that have to be supported in such applications are:

access(x, S): if $x \in S$, then return the information associated with x. Otherwise, report that x is not an element of S.

insert(x, S): insert element x into S, i.e., set $S := S \cup \{x\}$.

delete(x, S): delete element x from S, i.e., set $S := S \setminus \{x\}$.

A data structure that supports these three operations is called a *dictionary*. In this chapter, we do not consider the associated information of keys. That is, we identify each element of the set with its key. There are basically two ways of implementing dictionaries.

*School of Computer Science, Carleton University, Ottawa, Ontario, Canada K1S 5B6. E-mail: michiel@scs.carleton.ca.

Hashing. This gives a very fast implementation. It works, however, only for sets that come from a finite universe, say $K = [0 \ldots N-1]$.

Search trees. These are slower, but work for arbitrary universes, and are more flexible. For example, if element x is not contained in S, then the operation access(x, s) returns the smallest element $y \in S$ such that $y > x$, or the largest element $z \in S$ such that $z < x$. Using hashing, we only get the message that x is not an element of S. In many applications, it is useful to find the successor or predecessor of x, in case $x \notin S$.

2 Ingredients of hashing

Let N be a positive integer. We want to maintain a subset S of the *universe* $K = [0 \ldots N-1]$ under the operations access, insert, and delete.

Here is a trivial implementation. We use an array $A[0 \ldots N-1]$, where $A[i] = 1$ if $i \in S$, and $A[i] = 0$ if $i \notin S$. Clearly, all three operations take $O(1)$ time. The amount of space used, however, is $\Theta(N)$. That is, the amount of space used is independent of the size of S. We want an implementation that uses $O(|S|)$ space, because in many applications, the size of S is much smaller than the size of the universe. (This is certainly true for the examples given in Section 1.)

In this chapter, we show how to achieve $O(|S|)$ space using *hashing*. For hashing, we need

1. a *hash table*, which is an array $T[0 \ldots m-1]$, and
2. a *hash function*, which is a function $h : K \to [0 \ldots m-1]$.

Given the hash table and hash function, we store a subset S of K by storing element $x \in S$ in $T[h(x)]$.

Example 1 Let $N = 100$, $K = [0 \ldots 99]$, $m = 7$, $h(x) = x \bmod 7$, and $S = \{3, 19, 22\}$. Then the hash table looks as follows.

0	
1	22
2	
3	3
4	
5	19
6	

The operation access(x, S) is easy: Given x, compute $h(x)$, and look in T at position $h(x)$, i.e., look up $T[h(x)]$. Then, $x \in S$ if and only if $T[h(x)] = x$.

How do we insert an element? Consider our example, and assume that we want to insert element 17. We see that $h(17) = 3$, and that position 3 of T is occupied already, by element 3. Hence, there is a problem if there are elements $x, y \in S$, $x \neq y$, such that $h(x) = h(y)$. Such an event is called a *collision*. There are three methods to deal with collisions.

Open addressing. Here, we use two hash functions h_1 and h_2. If we want to insert element x, then we try the table positions

$$h_1(x), (h_1(x) + h_2(x)) \bmod m, (h_1(x) + 2h_2(x)) \bmod m, \ldots,$$

until we find a position that is not occupied. (Of course, we may not find such a free position.) For details of this technique, see [1, 5].

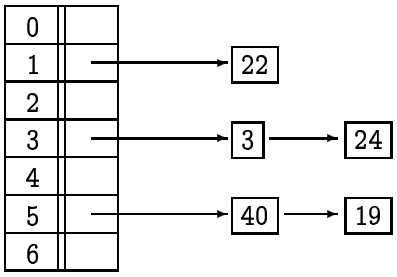
Chaining. This technique will be described in Section 3.

Perfect hashing. This technique will be presented in Section 4.

3 Hashing with chaining

Let the universe be $K = [0 \ldots N-1]$, and let $h : K \to [0 \ldots m-1]$ be a hash function. In hashing with chaining, the hash table $T[0 \ldots m-1]$ is an array of linear lists. That is, $T[i]$ is a (pointer to a) list that contains all elements x of S for which $h(x) = i$.

Example 2 Let $N = 100$, $K = [0 \ldots 99]$, $m = 7$, $h(x) = x \bmod 7$, and $S = \{3, 19, 22, 24, 40\}$. Then the hash table looks as follows.



The operation access(x, S) is processed as follows:

1. Compute $h(x)$.
2. Search for x in the list $T[h(x)]$.

The operations insert(x, S) and delete(x, S) are implemented in a similar way. In case of an insertion, we first check if x occurs already in S, by calling access(x, S). If $x \notin S$, then we add x at the end of the list $T[h(x)]$. To delete element x, we find x in $T[h(x)]$, and then delete it from this list.

The time for one operation is equal to the time to compute $h(x)$ plus the time to search the list $T[h(x)]$. Clearly, the size of $T[h(x)]$ can be proportional to the size of S. Hence, the worst-case time for an operation is linear in the size of S. The average case behavior, however, is much better.

Assumption 1 *We analyze the complexity of a sequence of n insertions, starting with an empty table, under the following assumptions.*

1. *For each $x \in K$, $h(x)$ can be computed in $O(1)$ time.*
2. *The hash function h distributes the universe K uniformly over the hash table, i.e.,*

$$|h^{-1}(i)| = |K|/m \text{ for all } i,\ 0 \leq i < m.$$

3. *The elements are drawn uniformly and independently from K. That is, each element of K has a probability of $1/N$ for being chosen as the next element to be inserted.*

We will discuss these assumptions at the end of Section 3.3.

For elements x and y of K, we define

$$\delta_h(x, y) := \begin{cases} 1 & \text{if } x \neq y \text{ and } h(x) = h(y), \\ 0 & \text{otherwise.} \end{cases}$$

Moreover, we define

$$\delta_h(x, S) := \sum_{y \in S} \delta_h(x, y).$$

Hence, if the hash table T stores the set S, and $x \notin S$, then $\delta_h(x, S)$ is the length of $T[h(x)]$. Therefore, the insertion of a new element x takes $O(1 + \delta_h(x, S))$ time.

What is the expected time for n insertions? To answer this question, we choose elements $x_1, x_2, \ldots, x_n$ uniformly and independently from K, and compute the expected value of the random variable $\delta_h(x_{i+1}, \{x_1, \ldots, x_i\})$, for each i, $0 \leq i < n$.

For each $1 \leq j \leq n$, let $p_j := h(x_j)$. The following observation follows from the second and third items in Assumption 1.

Observation 1 *For each* j, $1 \leq j \leq n$, *the random variable* p_j *is uniformly distributed in* $[0 \ldots m-1]$, *i.e., it takes any value in* $[0 \ldots m-1]$ *with probability* $1/m$.

3.1 Analyzing the expected insertion time

Recall that

$$E(\delta_h(x_{i+1}, \{x_1, \ldots, x_i\})) = \sum_{l=0}^{\infty} l \cdot \Pr(\delta_h(x_{i+1}, \{x_1, \ldots, x_i\}) = l).$$

Let $0 \leq p < m$. What is the probability that among $p_1, p_2, \ldots, p_i$, there are exactly l elements that are equal to p? Note that there are $\binom{i}{l}$ ways of choosing l elements from $p_1, p_2, \ldots, p_i$. Assume we have chosen l elements. Then for each of them, say p_j, the probability that $p_j = p$ is equal to $1/m$. For each of the remaining $i - l$ elements p_k, the probability that $p_k \neq p$ is equal to $1 - 1/m$.

Therefore, the probability that among $p_1, p_2, \ldots, p_i$, exactly l elements are equal to p is

$$\binom{i}{l} (1/m)^l (1 - 1/m)^{i-l}.$$

From this, we obtain the expected value of $\delta_h(x_{i+1}, \{x_1, \ldots, x_i\})$:

$$\begin{aligned} & E(\delta_h(x_{i+1}, \{x_1, \ldots, x_i\})) \\ &= \sum_{l=0}^{\infty} l \cdot \Pr(\delta_h(x_{i+1}, \{x_1, \ldots, x_i\}) = l) \end{aligned}$$

$$
\begin{aligned}
&= \sum_{l=0}^{\infty} l \cdot \Pr(\text{after } i \text{ insertions, } |T[h(x_{i+1})]| = l) \\
&= \sum_{l=0}^{\infty} l \cdot \Pr(\text{among } p_1, \ldots, p_i, \text{ exactly } l \text{ elements are equal to } p_{i+1}) \\
&= \sum_{l=0}^{\infty} l \cdot \binom{i}{l} (1/m)^l (1-1/m)^{i-l} \\
&= \frac{i}{m} \sum_{l=1}^{\infty} \frac{(i-1)!}{(l-1)!(i-l)!} (1/m)^{l-1} (1-1/m)^{i-l} \\
&= \frac{i}{m} \sum_{l=1}^{\infty} \binom{i-1}{l-1} (1/m)^{l-1} (1-1/m)^{i-1-(l-1)} \\
&= i/m, \qquad (1)
\end{aligned}
$$

because for $i \geq 1$,

$$
\begin{aligned}
&\sum_{l=1}^{\infty} \binom{i-1}{l-1} (1/m)^{l-1} (1-1/m)^{i-1-(l-1)} \\
&\quad = \sum_{k=0}^{\infty} \binom{i-1}{k} (1/m)^k (1-1/m)^{i-1-k} \\
&\quad = (1/m + (1-1/m))^{i-1} \\
&\quad = 1.
\end{aligned}
$$

It follows that the expected time for the $(i+1)$-st insertion is bounded by

$$O\left(1 + E(\delta_h(x_{i+1}, \{x_1, \ldots, x_i\}))\right) = O(1 + i/m).$$

This is also true for the first insertion, i.e., if $i = 0$. Hence, the total expected time for n insertions is bounded by[1]

$$O\left(\sum_{i=0}^{n-1} (1 + i/m)\right) = O\left(n + (1/m)\sum_{i=1}^{n-1} i\right) = O\left(n + n^2/m\right) = O(n + \beta n),$$

where $\beta := n/m$ is the *load factor* of the hash table. Note that β is the expected length of any list in this table.

The expected time for one insertion, say the $(i+1)$-st one, is bounded by

$$O(1 + i/m) = O(1 + (i/n) \cdot \beta) = O(1 + \beta).$$

[1] Here we use Lemma 1.

3.2 A simpler analysis of the expected insertion time

We now give an alternative and simpler proof of (1). For $0 \leq p < m$, let $L_p(i)$ be the length of the p-th list after i insertions. Then $L_p(i)$ is a random variable, and we showed above that

$$\Pr(L_p(i) = l) = \binom{i}{l}(1/m)^l(1-1/m)^{i-l}.$$

From this, we computed the expected value of

$$E(L_p(i)) = E(\delta_h(x_{i+1}, \{x_1, \ldots, x_i\})).$$

The following computation is much simpler. First observe that, by symmetry,

$$E(L_p(i)) = E(L_q(i)) \text{ for all } 0 \leq p < q \leq m-1.$$

Since $L_0(i) + L_1(i) + L_2(i) + \cdots + L_{m-1}(i) = i$, it follows that

$$\begin{aligned} i &= E(i) \\ &= E(L_0(i) + L_1(i) + \cdots + L_{m-1}(i)) \\ &= E(L_0(i)) + E(L_1(i)) + \cdots + E(L_{m-1}(i)). \end{aligned}$$

Since all terms in the latter summation are equal, we get $E(L_p(i)) = i/m$, which is exactly what we got in (1).

In this derivation, we used the so-called *linearity of expectation*:

Lemma 1 *Let X and Y be random variables, both having a finite expected value. Then*

$$E(X+Y) = E(X) + E(Y).$$

Proof. We will give the proof for the case when X and Y both take non-negative integer values. Recall that $E(X) = \sum_{i=0}^{\infty} i \cdot \Pr(X = i)$, and $E(Y) = \sum_{j=0}^{\infty} j \cdot \Pr(Y = j)$. Moreover

$$E(X+Y) = \sum_{k=0}^{\infty} k \cdot \Pr(X+Y = k).$$

Since

$$\Pr(X+Y = k) = \sum_{i=0}^{k} \Pr(X = i \text{ and } Y = k-i),$$

we get

$$\begin{aligned} E(X+Y) &= \sum_{k=0}^{\infty}\sum_{i=0}^{k}(i+k-i)\cdot \Pr(X=i \text{ and } Y=k-i) \\ &= \sum_{i=0}^{\infty}\sum_{k=i}^{\infty} i\cdot \Pr(X=i \text{ and } Y=k-i) \\ &\quad + \sum_{i=0}^{\infty}\sum_{k=i}^{\infty}(k-i)\cdot \Pr(X=i \text{ and } Y=k-i). \end{aligned}$$

In the first summation, we introduce a new variable $j = k - i$. Then this summation becomes

$$\sum_{i=0}^{\infty} i \sum_{j=0}^{\infty} \Pr(X=i \text{ and } Y=j) = \sum_{i=0}^{\infty} i\cdot \Pr(X=i) = E(X).$$

In the second summation, we do the same, which gives

$$\begin{aligned} \sum_{i=0}^{\infty}\sum_{j=0}^{\infty} j\cdot \Pr(X=i \text{ and } Y=j) &= \sum_{j=0}^{\infty} j \sum_{i=0}^{\infty} \Pr(X=i \text{ and } Y=j) \\ &= \sum_{j=0}^{\infty} j\cdot \Pr(Y=j) \\ &= E(Y). \end{aligned}$$

This shows that $E(X+Y) = E(X) + E(Y)$. ∎

We summarize our results. If Assumption 1 holds, then the expected time for one insertion is $O(1+\beta)$, where $\beta = n/m$ is the load factor of the hash table, and n is the size of the set S. The same time bound can be proved for access and delete operations, under the assumption that we search or delete a random element. Hence, if β is bounded by a constant, say $\beta \leq 1$, then the expected time for one operation is $O(1)$.

What can we say about the space complexity? We assume that the hash function h takes $O(1)$ space to store. (For hashing with chaining, this is a reasonable assumption.) The hash table T, together with the linear lists, has size $O(n+m)$. Hence the entire data structure has size

$$O(1+n+m) = O(n+n/\beta).$$

If β is not too small, say $\beta \geq 1/4$, then the space bound is linear in the size of S, i.e., $O(n)$. This leads to the following problem.

Problem 1 *Can we guarantee that* $1/4 \leq \beta \leq 1$*?*

3.3 The halving-doubling method

If we insert or delete an element, then the value of n and, hence, that of β changes. What do we do if β becomes too small or too large? The solution is simple: We just build a new hash table, with a new value of m. Hence, we also choose a new hash function h. This technique is known as the *halving-doubling method*, and it can be applied in many other situations. The details are as follows.

For each $i \geq 1$, let $h_i : K \to [0 \ldots 2^i - 1]$ be a hash function. Let n be the current size of S, and let i be an integer such that $2^{i-2} < n < 2^i$. Then we store the set S in a hash table $T_i[0 \ldots 2^i - 1]$ using the hash function h_i. Note that the load factor indeed satisfies $1/4 \leq \beta \leq 1$: We have $\beta = n/2^i \leq 1$, and

$$\beta = \frac{n}{2^i} \geq \frac{2^{i-2}}{2^i} = \frac{1}{4}.$$

If the value of n becomes 2^i, then we *double* the size of the hash table. That is, we build a new table $T_{i+1}[0 \ldots 2^{i+1} - 1]$ using the hash function h_{i+1}. (Such a rebuilding is called a *rehash operation*.) Note that then $2^{i-1} < n < 2^{i+1}$. Hence, if we increase i by one, then the new hash table satisfies our constraint.

Now suppose that the value of n becomes 2^{i-2}. Then we rehash the elements into a new table $T_{i-1}[0 \ldots 2^{i-1} - 1]$ using the hash function h_{i-1}. That is, we *halve* the size of the hash table. Again, it is clear that now $2^{i-3} < n < 2^{i-1}$. Hence, we can decrease i by one.

It is clear that a rehash operation can be performed in $O(n)$ time. Hence, we have achieved the following.

1. If we do not rehash, then one operation takes $O(1)$ expected time.
2. A rehash operation takes $O(n)$ time.
3. The data structure uses $O(n)$ space.

During an access operation, we never rehash. Therefore, the expected time for each access operation is bounded by $O(1)$.

Let us analyze the *expected amortized* time for an insertion or deletion. Consider a rehash operation, and let T_i be the rebuilt hash table. At this

moment, $n = 2^{i-1}$. If we rehash again, n must have been decreased to 2^{i-2}, or increased to 2^i. Let k be the number of updates that have taken place between these two rehash operations. Then

$$k \geq \min\left(2^{i-1} - 2^{i-2}, 2^i - 2^{i-1}\right) = 2^{i-2} \geq n/4.$$

Hence the total expected time for the first rehash operation and the sequence of k updates until the next rehash operation is bounded by $O(n+k) = O(k)$. Hence, if we amortize over the length k of the sequence, then we see that the expected amortized time for an insertion or deletion is bounded by $O(1)$.

Exercise 1 Prove the bound on the expected amortized update time using a potential function.

We have proved the following result.

Theorem 1 *If Assumption 1 holds, then hashing with chaining achieves*

1. $O(1)$ *expected time for an access operation, and*
2. $O(1)$ *expected amortized time for an insertion or deletion,*
3. *using* $O(n)$ *space.*

Is Assumption 1 reasonable? Suppose that m divides N. Then the hash function $h(x) = x \bmod m$ satisfies the first and second item. If m does not divide N, but N is much larger than m, then the second item is "almost" satisfied for $h(x) = x \bmod m$. The third item is more critical, because it postulates a certain behavior of the user of the hash table. In general, the exact behavior is not known at the moment when the hash table is constructed. Therefore, one has to be very careful when applying hashing with chaining.

3.4 The expected length of a longest list

We have seen that the expected length of an arbitrary list in the hash table $T[0 \ldots m-1]$, when storing n elements, is equal to $n/m = \beta$. What can we say about the expected length of a *longest* list? This length is a measure for the expected worst-case time for an operation. As before, our analysis uses Assumption 1.

Let $S \subseteq K$ be a random subset of size n. Assume that $\beta = n/m \leq 1$. For $0 \leq p \leq m-1$, let L_p be the number of elements of S that are stored in the p-th list of the hash table T. Then we are interested in the expected value

$$E\left(\max_{0 \leq p \leq m-1} L_p\right)$$

of the random variable $\max_{0 \leq p \leq m-1} L_p$.

The following lemma turns out to be useful.

Lemma 2 *Let X be a random variable that takes non-negative integer values. Then*

$$E(X) = \sum_{k=1}^{\infty} \Pr(X \geq k).$$

Proof. Since

$$\Pr(X = k) = \Pr(X \geq k) - \Pr(X \geq k+1),$$

we have

$$\begin{aligned} E(X) &= \sum_{k=0}^{\infty} k \cdot \Pr(X = k) \\ &= \sum_{k=1}^{\infty} k \cdot (\Pr(X \geq k) - \Pr(X \geq k+1)) \\ &= \sum_{k=1}^{\infty} k \cdot \Pr(X \geq k) - \sum_{j=2}^{\infty} (j-1) \cdot \Pr(X \geq j) \\ &= \sum_{k=1}^{\infty} \Pr(X \geq k). \end{aligned}$$

■

Let $k \geq 1$ be an integer. Then

$$\Pr(L_p \geq k) \leq \binom{n}{k} (1/m)^k.$$

Combining this with

$$\Pr\left(\max_{0 \leq p \leq m-1} L_p \geq k\right) \leq \sum_{p=0}^{m-1} \Pr(L_p \geq k),$$

we get

$$\begin{aligned}
\Pr\left(\max_{0\le p\le m-1} L_p \ge k\right) &\le m\binom{n}{k}(1/m)^k \\
&= \frac{n(n-1)(n-2)\cdots(n-k+1)}{k!m^{k-1}} \\
&\le \frac{n^k}{k!m^{k-1}} \\
&= n\frac{(n/m)^{k-1}}{k!} \\
&\le n/k!,
\end{aligned}$$

where the last inequality follows from the fact that $\beta \le 1$. Using Lemma 2, we get

$$\begin{aligned}
E\left(\max_{0\le p\le m-1} L_p\right) &= \sum_{k=1}^{\infty} \Pr\left(\max_{0\le p\le m-1} L_p \ge k\right) \\
&\le \sum_{k=1}^{\infty} \min(1, n/k!). \qquad (2)
\end{aligned}$$

Theorem 2 *If Assumption 1 holds, and if $\beta \le 1$, then the expected length of a longest list in the hash table is bounded by $O(\log n/\log\log n)$.*

Proof. We have to show an upper bound of $O(\log n/\log\log n)$ on the summation (2). Let i be the largest positive integer such that $n/i! \ge 1$. Since

$$i! \ge \left(\frac{i}{e}\right)^i, \qquad (3)$$

it follows that $n \ge (i/e)^i$. Hence,

$$i = O(\log n/\log\log n). \qquad (4)$$

We conclude that

$$\begin{aligned}
&\sum_{k=1}^{\infty} \min(1, n/k!) \\
&\quad = i + \sum_{k=i+1}^{\infty} n/k!
\end{aligned}$$

$$
\begin{aligned}
&\leq i + \sum_{k=i+1}^{\infty} (i+1)!/k! \\
&= i + 1 + \frac{1}{i+2} + \frac{1}{(i+2)(i+3)} + \frac{1}{(i+2)(i+3)(i+4)} + \cdots \\
&\leq i + 1 + 1/2 + 1/2^2 + 1/2^3 + \cdots \\
&= i + \sum_{k=0}^{\infty} (1/2)^k \\
&= i + 2 \\
&= O(\log n / \log\log n).
\end{aligned}
$$

■

Exercise 2 Prove (3) and (4). *Hint:* To prove (3), take logarithms, and estimate the sum by an integral. To prove (4), assume that $i > c \log n / \log\log n$ for an appropriate constant c. Then show that $i \log(i/e) > \log n$, which is a contradiction.

Remark 1 We have proved an upper bound on the expected length of a longest list in the hash table. Gonnet [4] has shown that, if $1/4 \leq \beta \leq 1$, this expected length is in fact $\Theta(\log n / \log\log n)$.

We can interpret this result as follows. Assume we have n balls and n buckets. We throw these balls, randomly and independently, in the buckets. Then any fixed bucket contains, expected, exactly one ball. On the other hand, the bucket with the largest number of balls contains, again expected, $\Theta(\log n / \log\log n)$ balls.

4 Perfect hashing

In the previous sections, we have seen a hashing scheme, which is efficient provided Assumption 1 holds. In this section, we will design a hashing scheme that is efficient for *any* set S.

Recall the ingredients of hashing. We have a universe $K = [0 \ldots N-1]$ and a hash function $h : K \to [0 \ldots m-1]$. A subset S of K, having size n, is stored in a hash table $T[0 \ldots m-1]$ by storing element x of S in $T[h(x)]$. In order to guarantee a data structure of size $O(n)$, the load factor $\beta = n/m$ must be bounded from below by a constant. A problem arises, if there are

two distinct elements x and y in S such that $h(x) = h(y)$. Can we avoid such collisions?

Assume that $n \leq m$. Then, if the hash function h is *injective* on S, there are no collisions. Clearly, such an injective function exists. However,

1. how much space do we need to specify such a function?
2. how much time do we need to find such a function?
3. given an element x, how much time do we need to compute $h(x)$?

In this section, we will show that an injective function h can be computed in $O(n)$ expected time (or $O(nN)$ deterministic time), that needs $O(n)$ space to be stored, and that can be evaluated in $O(1)$ time.

We start with the static case, i.e., there are only access operations. The construction is due to Fredman, Komlós and Szemerédi [3].

4.1 The main lemma and its consequences

Let N be a prime number, and S a subset of the universe $K = [0 \ldots N-1]$, having size n. For each k, $1 \leq k \leq N-1$, we define

$$h_k(x) := ((kx) \bmod N) \bmod m.$$

Furthermore, for each k, $1 \leq k \leq N-1$, and i, $0 \leq i \leq m-1$, we define

$$b_i^k := |\{x \in S : h_k(x) = i\}|.$$

The final injective hash function will be based on the functions h_k. Note that b_i^k is equal to the number of elements of S that are mapped, by h_k, to i. We want a value of k, such that b_i^k is "small" for all i, $0 \leq i \leq m-1$. The following lemma is the basis for the fact that such a k exists and can be found efficiently.

Lemma 3 *We have*

$$\sum_{k=1}^{N-1} \sum_{i=0}^{m-1} b_i^k (b_i^k - 1) \leq 2 \frac{n(n-1)}{m} (N-1).$$

Before we prove this lemma, let us see what it says. First note that $\sum_{i=0}^{m-1} b_i^k = n$. Imagine the b_i^k as arbitrary numbers such that $\sum_{i=0}^{m-1} b_i^k = n$. The sum $\sum_{i=0}^{m-1} b_i^k(b_i^k - 1)$ is minimal if all b_i^k are equal, i.e., equal to n/m. In this case, the double sum in the lemma is equal to

$$\sum_{k=1}^{N-1} \sum_{i=0}^{m-1} \frac{n}{m}\left(\frac{n}{m} - 1\right) = (N-1)m\,\frac{n}{m}\left(\frac{n}{m} - 1\right) \sim \frac{n^2}{m}(N-1).$$

Hence, the lemma says that the double sum is (approximately) at most twice its minimal possible value.

Proof. Since

$$b_i^k(b_i^k - 1) = |\{(x,y) : x,y \in S, x \neq y, h_k(x) = h_k(y) = i\}|,$$

we have

$$\begin{aligned}
&\sum_{k=1}^{N-1} \sum_{i=0}^{m-1} b_i^k(b_i^k - 1) \\
&= \sum_{k=1}^{N-1} \sum_{i=0}^{m-1} |\{(x,y) : x,y \in S, x \neq y, h_k(x) = h_k(y) = i\}| \\
&= \sum_{k=1}^{N-1} |\{(x,y) : x,y \in S, x \neq y, h_k(x) = h_k(y)\}| \\
&= \sum_{(x,y)\in S\times S, x\neq y} |\{k : 1 \leq k \leq N-1, h_k(x) = h_k(y)\}|.
\end{aligned}$$

Let $x,y \in S$, $x \neq y$, be fixed. How many k, $1 \leq k \leq N-1$, are there such that $h_k(x) = h_k(y)$? Note that

$$h_k(x) = h_k(y) \text{ if and only if } ((kx) \bmod N - (ky) \bmod N) \bmod m = 0.$$

Define $g(k) := (kx) \bmod N - (ky) \bmod N$. Then

$$g(k) \bmod m = 0 \text{ if and only if } g(k) = jm \text{ for some integer } j.$$

Since $0 \leq kx \bmod N \leq N-1$ and $0 \leq ky \bmod N \leq N-1$, we have

$$-N+1 \leq g(k) \leq N-1.$$

Moreover, $g(k) \neq 0$. (For, if $g(k) = 0$, then $k(x-y) \bmod N = 0$. Hence, since N is a prime, we have $k \bmod N = 0$ or $(x-y) \bmod N = 0$. But,

$1 \leq k \leq N-1$, $0 \leq x \leq N-1$, $0 \leq y \leq N-1$, and $x \neq y$. Hence, $k \bmod N \neq 0$ and $(x-y) \bmod N \neq 0$. This is a contradiction.)

It follows that $g(k) \bmod m = 0$ if and only if $g(k) = jm$ for some integer $j \neq 0$ that satisfies $(-N+1)/m \leq j \leq (N-1)/m$. We claim that for each j, there is at most one $k \in [1 \ldots N-1]$ such that $g(k) = jm$.

To prove this, let k_1 and k_2 be elements from $[1 \ldots N-1]$ such that $g(k_1) = g(k_2)$. Then

$$(k_1 x) \bmod N - (k_1 y) \bmod N = (k_2 x) \bmod N - (k_2 y) \bmod N,$$

which can be rewritten as

$$(k_1 - k_2)(x-y) \bmod N = 0.$$

Since N is a prime, $(k_1 - k_2) \bmod N = 0$ or $(x-y) \bmod N = 0$. We saw already that $(x-y) \bmod N \neq 0$. Therefore, we must have $(k_1 - k_2) \bmod N = 0$. Since $1 \leq k_1 \leq N-1$ and $1 \leq k_2 \leq N-1$, it follows that $k_1 = k_2$. This proves the claim.

We conclude that the number of k, $1 \leq k \leq N-1$, such that $h_k(x) = h_k(y)$ is at most equal to the number of integers $j \neq 0$ such that $(-N+1)/m \leq j \leq (N-1)/m$. Therefore,

$$\begin{aligned}
\sum_{k=1}^{N-1} \sum_{i=0}^{m-1} b_i^k (b_i^k - 1) &= \sum_{(x,y) \in S \times S, x \neq y} |\{k : 1 \leq k \leq N-1, h_k(x) = h_k(y)\}| \\
&\leq \sum_{(x,y) \in S \times S, x \neq y} 2\,\frac{N-1}{m} \\
&= 2\,\frac{n(n-1)}{m}(N-1).
\end{aligned}$$

This completes the proof. ∎

For $1 \leq k \leq N-1$, let

$$B_k := \sum_{i=0}^{m-1} b_i^k (b_i^k - 1).$$

Then, Lemma 3 says that

$$\sum_{k=1}^{N-1} B_k \leq 2\,\frac{n(n-1)}{m}(N-1).$$

Corollary 1 *1. There exists a* k, $1 \le k \le N-1$, *such that* $B_k \le 2n(n-1)/m$.

2. There are at least $(N-1)/2$ *values of* k, $1 \le k \le N-1$, *such that* $B_k \le 4n(n-1)/m$.

Proof. The first claim is equivalent to the fact that there is at least one B_k which is at most equal to the average value of all B_k's.

To prove the second claim, first note that $B_k \ge 0$ for all k. The second claim is equivalent to saying that at least 50% of the B_k's are at most twice the average value of all B_k's. We prove the latter statement. Let

$$A := \{k : 1 \le k \le N-1, B_k \le 4n(n-1)/m\}.$$

Then

$$\sum_{k=1}^{N-1} B_k \ge \sum_{k \notin A} B_k \ge (N-1-|A|)\,4n(n-1)/m.$$

On the other hand, Lemma 3 says that

$$\sum_{k=1}^{N-1} B_k \le 2\,\frac{n(n-1)}{m}(N-1).$$

Combining these inequalities gives

$$(N-1-|A|)\,4\,\frac{n(n-1)}{m} \le 2\,\frac{n(n-1)}{m}(N-1),$$

which can be rewritten as $|A| \ge (N-1)/2$. This is exactly what we wanted to show. ∎

We now show that a value of k as in Corollary 1 can be computed efficiently.

Corollary 2 *1. In* $O(m+nN)$ *deterministic time, we can compute a* k, $1 \le k \le N-1$, *such that* $B_k \le 2n(n-1)/m$.

2. In $O(m+n)$ *expected time, we can compute a* k, $1 \le k \le N-1$, *such that* $B_k \le 4n(n-1)/m$.

Proof. By Corollary 1, there exists a value of k such that $B_k \le 2n(n-1)/m$. The following exhaustive search algorithm finds such a k:

Algorithm *findk*:
(* this algorithm finds a value of k such that $B_k \leq 2n(n-1)/m$ *)
for $i := 0$ **to** $m-1$ **do** $T[i] := 0$ **endfor**;
for $k := 1$ **to** $N-1$ **do** *test*$(k, 2n(n-1)/m)$ **endfor**

The procedure *test*$(k, upper)$ is defined below. If this procedure is called, then there is a table $T[0 \ldots m-1]$ such that $T[i] = 0$ for all i. The procedure tests if $B_k \leq upper$. If so, it returns k. Otherwise, it resets all $T[i]$ to zero.

Algorithm *test*$(k, upper)$:
$B := 0$;
for all $x \in S$
do $i := h_k(x)$;
  $T[i] := T[i] + 1$;
  $B := B + T[i] \cdot (T[i] - 1) - (T[i] - 1) \cdot (T[i] - 2)$
endfor;
if $B \leq upper$
then return k
else for all $x \in S$ **do** $T[h_k(x)] := 0$ **endfor**
endif

The procedure *test* has a running time of $O(n)$. The initialization of algorithm *findk*, i.e., the first for-loop, takes $O(m)$ time. Since there are at most $N-1$ calls to *test*, the total running time is bounded by $O(m+nN)$. This proves the first claim.

We know from Corollary 1, that there are at least $(N-1)/2$ values of k such that $B_k \leq 4n(n-1)/m$. The following randomized algorithm finds such a k:

Algorithm *randomfindk*:
(* this randomized algorithm finds a value of k such that
  $B_k \leq 4n(n-1)/m$ *)
for $i := 0$ **to** $m-1$ **do** $T[i] := 0$ **endfor**;
while *true*
do choose a random k in $[1 \ldots N-1]$;
  test$(k, 4n(n-1)/m)$
endwhile

In one iteration of the while-loop, the probability that the algorithm chooses a "good" value of k is at least $1/2$. Therefore, the expected number of iterations is at most 2. Hence, the expected running time of the algorithm is bounded by $O(m+n)$. ∎

Remark 2 The running time of algorithm *randomfindk* depends on coin flips made by the algorithm. It does not, however, depend on the set S. That is, no distribution for S is assumed. For *any* set S, the expected running time is $O(m+n)$.

Until now, we did not specify the value of m. In the next two corollaries, we consider two cases, $m \sim n^2$ and $m = n$.

Corollary 3 *1. Let $m = n(n-1)+1$. In $O(n^2+nN)$ deterministic time, we can compute a k, $1 \le k \le N-1$, such that the function $h_k(x) = ((kx) \bmod N) \bmod m$ is injective on S.*

2. Let $m = 2n(n-1)+1$. For at least $(N-1)/2$ values of k, $1 \le k \le N-1$, the function h_k is injective on S. One such k can be found in $O(n^2)$ expected time.

Proof. According to Corollary 2, we can find in $O(m+nN) = O(n^2+nN)$ time, a value of k such that

$$B_k \le 2\,\frac{n(n-1)}{m} = 2\,\frac{n(n-1)}{n(n-1)+1} < 2.$$

Note that $B_k = \sum_{i=0}^{m-1} b_i^k(b_i^k - 1)$, where the b_i^k are non-negative integers. We claim that $b_i^k \in \{0,1\}$ for all $0 \le i \le m-1$.

To prove this claim assume there is an index j such that $b_j^k \ge 2$. Then

$$B_k = \sum_{i=0}^{m-1} b_i^k(b_i^k - 1) \ge b_j^k(b_j^k - 1) \ge 2,$$

which is a contradiction.

The fact that $b_i^k \in \{0,1\}$ for all $0 \le i \le m-1$, means that the function h_k is injective on S. This proves the first claim. The second claim can be proved in a similar way. ∎

We were looking for a hash function that is injective on S. Corollary 3 gives us such a function. This function, however, leads to a load factor

$\beta = n/m$ which is proportional to $1/n$. This is not what we want, because by using this hash function, the hash table will have size $O(n+m) = O(n^2)$.

Corollary 4 *Let* $m = n$.

1. *In* $O(nN)$ *deterministic time, we can compute a* k, $1 \leq k \leq N-1$, *such that* $B_k \leq 2(n-1)$.

2. *In* $O(n)$ *expected time, we can compute a* k, $1 \leq k \leq N-1$, *such that* $B_k \leq 4(n-1)$.

Proof. Apply Corollary 2 with $m = n$. ∎

4.2 A two-level injective hash function

We are now ready to give the injective hash function we were looking for. Corollaries 3 and 4 suggest the following two-level scheme. (We only give the randomized version.)

Level 1: Compute a hash function $h_k(x) = ((kx) \bmod N) \bmod n$ such that $B_k \leq 4(n-1)$. (By Corollary 4, such a hash function exists, and can be computed efficiently.)

Level 2: For each i, $0 \leq i \leq n-1$, let $W_i := \{x \in S : h_k(x) = i\}$, $b_i := |W_i|$, and $m_i := 2b_i(b_i - 1) + 1$. Note that $\sum_{i=0}^{n-1} b_i(b_i-1) = B_k$.

For each i, $0 \leq i \leq n-1$, compute a hash function

$$x \to ((k_i x) \bmod N) \bmod m_i$$

that is injective on W_i. (By Corollary 3, such a hash function exists, and can be computed efficiently.)

The hash table: Let $m := \sum_{i=0}^{n-1} m_i$ and $s_i := \sum_{j=0}^{i-1} m_j$, for $0 \leq i \leq n$. The hash table $T[0 \ldots m-1]$ stores the elements x of S as follows. (See Figure 1.)

1. compute $i := ((kx) \bmod N) \bmod n$.

2. compute $j := ((k_i x) \bmod N) \bmod m_i$.

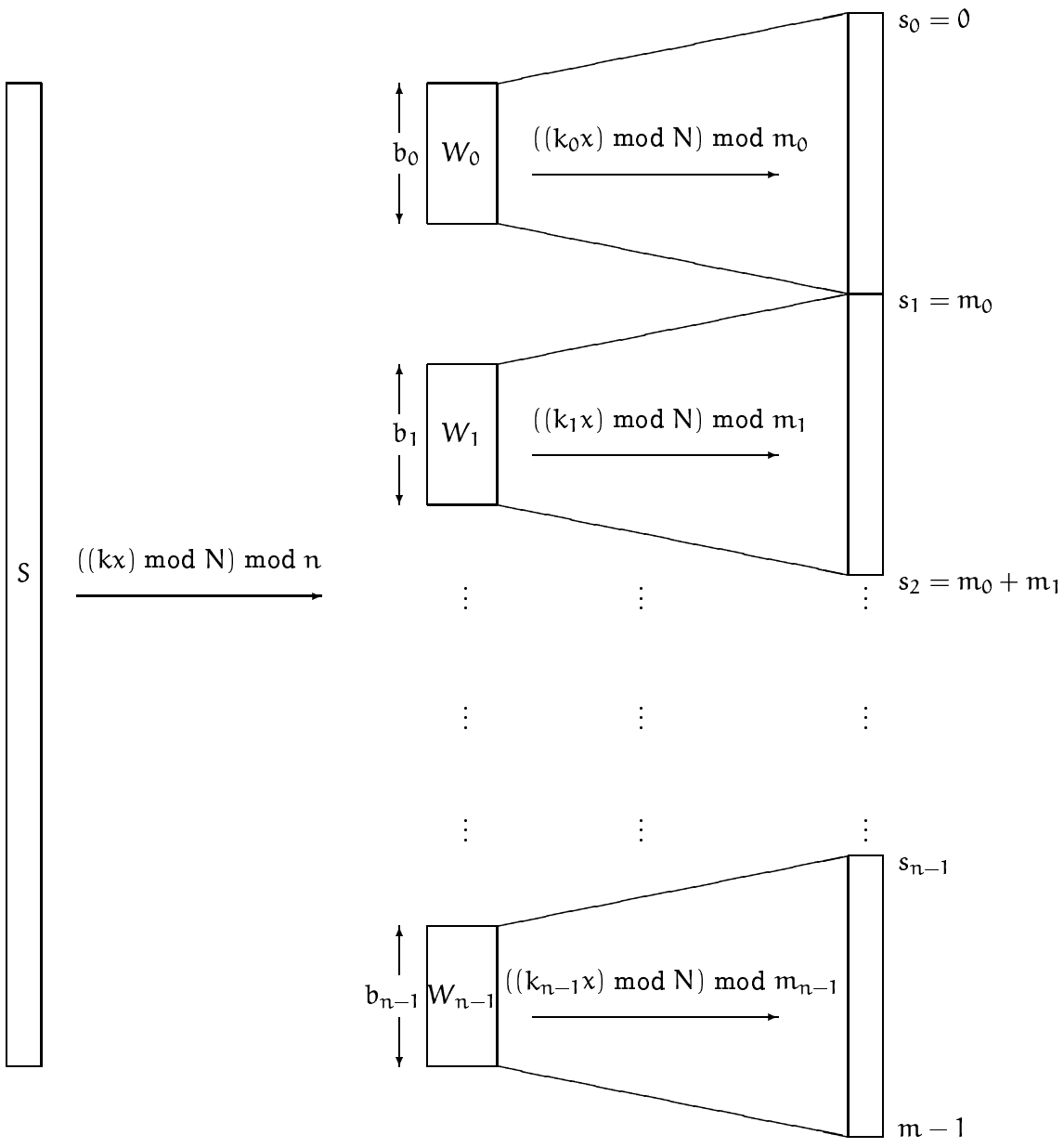


Figure 1: *A perfect hash table.*

3. store x in $T[s_i + j]$.

Exercise 3 Convince yourself of the fact that the complete hash function is indeed injective on S.

How much space do we need to store the hash function and the hash table? First note that

$$m = \sum_{i=0}^{n-1} m_i = \sum_{i=0}^{n-1} (2b_i(b_i - 1) + 1) = 2B_k + n,$$

which implies that

$$m \leq 2 \cdot 4(n-1) + n = 9n - 8.$$

Since $m \geq n$, the load factor lies in between $1/9$ and 1.

The entire data structure is specified by the following:

1. N, n, and k.
2. Three arrays storing the sequences $k_0, k_1, \ldots, k_{n-1}$, $m_0, m_1, \ldots, m_{n-1}$, and $s_0, s_1, \ldots, s_n$.
3. The table $T[0 \ldots m-1]$ containing the elements of S.

Hence, the total amount of space used is

$$3 + 2n + (n+1) + m = 4 + 3n + m \leq 4 + 3n + 9n - 8 = 12n - 4 = O(n).$$

Remark 3 Since $m_i = s_{i+1} - s_i$, we do not have to store the m_i's explicitly.

The search time is easy to analyze: In order to search for an element x, we make two multiplications, four integer divisions, and one addition. Hence the time for an access operation is $O(1)$.

Finally, we bound the expected time to construct the data structure. By Corollary 4, it takes $O(n)$ expected time to find an integer k for which $B_k \leq 4(n-1)$. Given this k, the sets W_i, the values b_i and m_i, $0 \leq i \leq n-1$, and the values s_i, $0 \leq i \leq n$, can be determined in $O(n)$ time. Then, by Corollary 3, for each i, the integer k_i can be found in $O(b_i^2)$ expected time. Finally, the table T can be filled with the elements of S in $O(n)$ time. This

proves that the expected time to build the entire data structure is bounded by

$$O\left(n + \sum_{i=0}^{n-1} b_i^2\right).$$

Note that $z^2 \leq 2z(z-1)+1$ for all real numbers z. Therefore,

$$\sum_{i=0}^{n-1} b_i^2 \leq \sum_{i=0}^{n-1} (2b_i(b_i-1)+1) = 2B_k + n.$$

It follows that the expected building time is bounded by $O(n+B_k) = O(n)$.

We summarize our result. (We have only analyzed the randomized construction. The proof for the deterministic construction is similar.)

Theorem 3 *Let N be a prime number, and S a subset of $[0 \ldots N-1]$ of size n. A perfect hash table for S can be built in $O(nN)$ deterministic time (resp. $O(n)$ expected time). The data structure has size $O(n)$ and each access operation can be processed in $O(1)$ worst-case time.*

4.3 Dynamic perfect hashing

We have proved Theorem 3 for the case when the set S is fixed. In this section, we consider the case when also insertions and deletions have to be supported. The results of this section are due to Dietzfelbinger *et al.*[2].

We first consider a somewhat simpler problem. Let $S = \{x_1, x_2, \ldots, x_n\}$ be a subset of the universe K. We want to build a perfect hash table *on-line*. That is, we start with an empty hash table, and insert $x_1, x_2, \ldots, x_n$, in this order. When we insert element x_l, we only know the elements $x_1, \ldots, x_{l-1}$. The value of n is known at the start of the algorithm.

The basic approach is similar to the halving-doubling method of Section 3.3. If $B_k > 4(n-1)$, or if our hash function is not injective on the current set $\{x_1, \ldots, x_l\}$, then we rebuild the complete, or a part of the current hash table. We take care that such rebuildings do not occur too often. In this way, the amortized insertion time will be low. The algorithm is given in Figure 2.

Remark 4 If the function $((k_i x) \bmod N) \bmod m_i$ is not injective on W_i, then we have to find a new hash function for W_i. We choose a random k_i and check if it is a good choice. Of course, we want the probability of a

Algorithm *BuildTable*:
choose a random k in $[1 \ldots N-1]$;
for $i := 0$ **to** $n-1$
do $W_i := \emptyset$; $b_i := 0$; $m_i := 1$; $s_i := i$;
choose a random k_i in $[1 \ldots N-1]$
endfor;
$m := n$; $B := 0$;
$(* \; B = \sum b_i(b_i - 1) \; *)$
for $l := 1$ **to** n **do** *insert*(x_l) **endfor**

Figure 2: *Building a hash table on-line. The procedure insert is given in Figure 3.*

---

random k_i being "good" to be sufficiently large, say at least $1/2$. Note that with each insertion into W_i, the value of b_i increases. Therefore, in order to apply Corollary 3, we also increase m_i from time to time. It turns out that doubling m_i guarantees a linear space bound.

We analyze the complexity of the insertion algorithm. Consider the fragment starting at *TryAgain*. There, we choose a random k_i and test if it is a good choice. One such test takes $O(b_i)$ time. (Note that either $T[s_i \ldots s_i + m_i - 1]$ is empty already, or we can make it empty in $O(b_i)$ time.) Corollary 3 implies that a random k_i induces an injective function on W_i with probability at least $1/2$. It follows that, on the average, at most two tries are needed to find a good k_i. That is, during an insertion into W_i, we spend $O(b_i)$ expected time for finding a new hash function for W_i.

Let b_i^f be the final value of b_i, i.e., the value of b_i after all n elements have been inserted. Then b_i runs from zero to b_i^f. The total expected time for finding new hash functions for W_i, during all insertions, is bounded by

$$O\left(\sum_{j=1}^{b_i^f} j\right) = O\left(\left(b_i^f\right)^2\right).$$

The complete hash table is rebuilt if $B > 4(n-1)$. By Corollary 4, a random choice for k gives

$$B^f := \sum_{i=0}^{n-1} b_i^f(b_i^f - 1) \leq 4(n-1)$$

```
Algorithm insert(x_l):
i := ((k x_l) mod N) mod n;
j := ((k_i x_l) mod N) mod m_i;
W_i := W_i ∪ {x_l};
B := B + 2b_i;
b_i := b_i + 1;
(* B = ∑ b_i(b_i − 1) *)
if B > 4(n − 1)
then (* try another k *)
       BuildTable
endif;
if T[s_i + j] is free
then store x_l in T[s_i + j]
else (* the function ((k_i x) mod N) mod m_i is not injective
          on W_i. See Remark 4. *)
     if m_i < 2b_i(b_i − 1) + 1
     then (* we have to increase m_i *)
          m_i := 4b_i(b_i − 1) + 1;
          s_i := m;
          m := m + m_i
          (* we reserve T[s_i ... m − 1] for the new table for W_i *)
     endif;
     TryAgain: choose a random k_i in [1 ... N − 1];
               make T[s_i ... s_i + m_i − 1] empty;
               for all x ∈ W_i
               do j := ((k_i x) mod N) mod m_i;
                  if T[s_i + j] is free
                  then store x in T[s_i + j]
                  else goto TryAgain
                  endif
               endfor
endif
```

Figure 3: *Inserting element* x_l *into a perfect hash table.*

with probability at least $1/2$. Since $B \leq B^f$ during the algorithm, we have $B \leq 4(n-1)$ with probability at least $1/2$. Hence, we try, on average, at most two values for k before we find a good one. (This means that the complete hash table is rebuilt, on average, at most once.)

It follows that the total expected time for n insertions is bounded by

$$O\left(n + \sum_{i=0}^{n-1} \left(b_i^f\right)^2\right) = O(n + B_f) = O(n).$$

Equivalently, the amortized expected insertion time is bounded by $O(1)$.

How large is the hash table? Let $m_{i,1}, m_{i,2}, \ldots$, be the different values of m_i during the n insertions, and let m_i^f be the final value.

Claim 1 $m_{i,p+1} \geq 2 \cdot m_{i,p}$.

Proof. There is an integer b such that $m_{i,p+1} = 4b(b-1)+1$. We set m_i to this value as soon as the current value of m_i is less than $2b(b-1)+1$. Hence, $m_{i,p} \leq 2b(b-1)$. ∎

For one set W_i, we need space

$$\sum_p m_{i,p} \leq \sum_{p=0}^{\infty} \frac{m_i^f}{2^p} = 2 \cdot m_i^f.$$

Note that $m_i^f \leq 4b_i^f(b_i^f - 1) + 1$. Hence for the complete set S, we need a table of size

$$\sum_{i=0}^{n-1} 2 \cdot m_i^f \leq \sum_{i=0}^{n-1} \left(8b_i^f(b_i^f - 1) + 2\right) \leq 8 \cdot 4(n-1) + 2n \leq 34n.$$

We have proved the following result.

Theorem 4 *Let N be a prime number, and S a subset of $[0 \ldots N-1]$ of size n. We can build a perfect hash table for S on-line in $O(n)$ expected time, i.e., in $O(1)$ amortized expected time per insertion. The hash table has size $O(n)$ and each access operation can be processed in $O(1)$ worst-case time.*

Remark 5 In the on-line algorithm, we assumed that the value of n is known at the start of the construction. If this is not the case, then we

can apply the halving-doubling method of Section 3.3. Hence, the result of Theorem 4 remains valid.

We only showed how to insert elements. The algorithm can be extended to support deletions as well: To delete an element x, we first search it, and then *mark* it as being deleted. If we rebuild the hash table, then we delete all marked elements. In order to guarantee that the number of marked elements does not become too large, we again apply the halving-doubling method. This gives a fully dynamic perfect hash table with $O(1)$ amortized expected insertion and deletion time, and $O(1)$ worst-case access time, that uses $O(n)$ space.

References

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, 1990.

[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.* SIAM Journal on Computing **23** (1994), pp. 738–761.

[3] M.L. Fredman, J. Komlós, E. Szemerédi. *Storing a sparse table with $O(1)$ worst case access time.* Journal of the ACM **31** (1984), pp. 538–544.

[4] G.H. Gonnet. *Expected length of the longest probe sequence in hash code searching.* Journal of the ACM **28** (1981), pp. 289–304.

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