

String Hashing for Linear Probing

Mikkel Thorup*

Abstract

Linear probing is one of the most popular implementations of dynamic hash tables storing all keys in a single array. When we get a key, we first hash it to a location. Next we probe consecutive locations until the key or an empty location is found. At STOC'07, Pagh et al. presented data sets where the standard implementation of 2-universal hashing leads to an expected number of $\Omega(\log n)$ probes. They also showed that with 5-universal hashing, the expected number of probes is constant. Unfortunately, we do not have 5-universal hashing for, say, variable length strings. When we want to do such complex hashing from a complex domain, the generic standard solution is that we first do collision free hashing (w.h.p.) into a simpler intermediate domain, and second do the complicated hash function on this intermediate domain. Our contribution is that for an expected constant number of linear probes, it suffices that each key has $O(1)$ expected collisions with the first hash function, as long as the second hash function is 5-universal. This means that the intermediate domain can be n times smaller, and such a smaller intermediate domain typically means that the overall hash function can be made simpler and at least twice as fast. The same doubling of hashing speed for $O(1)$ expected probes follows for most domains bigger than 32-bit integers, e.g., 64-bit integers and fixed length strings. In addition, we study how the overhead from linear probing diminishes as the array gets larger, and what happens if strings are stored directly as intervals of the array. These cases were not considered by Pagh et al.

1 Introduction

A *hash table* or *dictionary* is the most basic non-trivial data structure. We want to store a set S of keys from some universe U so that we can check membership, that is, if some $x \in U$ is in S , and if so, look up satellite information associated with x . Often we want the set S to be dynamic so that we can insert and delete keys. We are not interested in the ordering of the elements of U , and that is why we can use hashing for efficient implementation of these tables. Hash tables for strings and other complex objects are central to the

analysis of data, and they are directly built into high level programming languages such as `python`, and `perl`.

Linear probing is one of the most popular implementations of hash tables in practice. We store all keys in a single array T , and have a hash function h mapping keys from U into array locations. When we get a key x , we first check location $h(x)$ in T . If a different key is in $T[h(x)]$, we scan next locations sequentially until either x is found, or we get to an empty spot concluding that key x is new. If x is to be inserted, we place it in this empty spot. To delete a key x from location i , we have to check if there is a later location $j \geq i$ with a key y such that $h(y) \leq i$, and in that case delete y from j and move it up to i . Recursively, we look for a later key z to move up to j . This deletion process terminates when we get to an empty spot. Thus, for each operation, we only consider locations from $h(x)$ and to the first empty location. Above, the successor to the last location in the array is the first location, but we will generally ignore this boundary case. It can also be avoided in the code if we leave some extra space at the end of the array that we do not hash to. If the keys are complex objects like variable length strings, we will typically not store them directly in the array T . Instead of storing a key x , we store its hash value $h(x)$ and a pointer to x . Storing the hash value serves two purposes: (1) during deletions we can quickly identify keys to be moved up as described above, and (2) when looking for a key x , we only need to check keys with the same hash value. We will, however, also consider the option of storing strings directly as intervals of the array that we probe, and thus avoid the pointers.

Practice The practical use of linear probing dates back at least to 1954 to an assembly program by Samuel, Amdahl, Boehme (c.f. [10]). It is one of the simplest schemes to implement in dynamic settings where keys can be inserted and deleted. Several recent experimental studies [2, 7, 14] have found linear probing to be the fastest hash table organization for moderate load factors (30-70%). While linear probing is known to require more instructions than other open addressing methods, the fact that we access an interval of array entries means that linear probing works very well with modern architectures for which sequential access is much faster than random access (assuming that the

*AT&T Labs—Research, Shannon Laboratory,
180 Park Avenue, Florham Park, NJ 07932, USA.
mthorup@research.att.com.

elements we are accessing are each significantly smaller than a cache line, or a disk block, etc.). However, the hash functions used to implement linear probing in practice are heuristics, and there is no known theoretical guarantee on their performance. Since linear probing is particularly sensitive to a bad choice of hash function, Heileman and Luo [7] advice *against* linear probing for general-purpose use.

Analysis Linear probing was first analyzed by Knuth in a 1963 memorandum [9] now considered to be the birth of the area of analysis of algorithms [15]. Knuth's analysis, as well as most of the work that has since gone into understanding the properties of linear probing, is based on the assumption that h is a truly random function, mapping all keys independently. In 1977, Carter and Wegman's notion of universal hashing [3] initiated a new era in the design of hashing algorithms, where explicit and efficient ways of choosing provably good hash functions replaced the unrealistic assumption of complete randomness. They asked to extend the analysis to linear probing.

Carter and Wegman [3] defined *universal hashing* as having low collision probability, that is $1/t$ if we hash to a domain of size t . Later, in [20], they define k -universal hashing as a function mapping any k keys independently and uniformly at random. Note that 2-universal hashing is stronger than universal hashing in that the identity is universal but not 2-universal. Often the uniformity does not have to be exact, but the independence is critical for the analysis.

The first analysis of linear probing based on k -universal hashing was given by Siegel and Schmidt in [16, 17]. Specifically, they show that $O(\log n)$ -universal hashing is sufficient to achieve essentially the same performance as in the fully random case. Here n denotes the number of keys inserted in the hash table. However, we do not have any practical implementation of $O(\log n)$ -universal hashing.

In 2007, Pagh et al. [12] studied the expected number of probes with worst-case data sets. They showed that with the standard implementation of 2-universal hashing, the expected number of linear probes could be $\Omega(\log n)$. The worst-case is one or two intervals — something that could very well appear in practice, possibly explaining the experienced unreliability from [7]. It is interesting to contrast this with the result from [11] that simple hashing works if the input has high entropy. The situation is similar to the classic one for non-randomized quick sort, where we get into quadratic running time if the input is already sorted: something very unlikely for random data, but something that happens frequently in practice. Likewise, if we were hashing characters, then in practice it could happen that

they were mostly letters and digits, and then they would be concentrated in two intervals.

On the positive side, Pagh et al. [12] showed that with 5-universal hashing, the expected number of probes is $O(1)$. From [19] we know that 5-universal hashing is fast for small domains like 32-bit integers.

Main contribution Our main contribution is that for an expected constant number of linear probes, we can salvage 2-universal hashing and even universal hashing if we follow it by 5-universal hashing. The universal hashing collects keys with the same hash value in buckets. The subsequent 5-universal hashing shuffles the buckets. Clearly this is weaker hashing than if we applied the 5-universal hashing directly to all keys, but it may seem that the initial universal hashing is wasted work.

Our motivation is to get efficient hashing for linear probing of complex objects like variable length strings. The point is that we do not have any reasonable 5-universal hashing for these domains. When we want complex hashing from a complex domain, the generic standard solution is that we first do collision free hashing (w.h.p.) into a simpler intermediate domain, and second do the complicated hash function on this intermediate domain.

Our contribution is that for expected constant linear probes, it suffices that each key has $O(1)$ expected collisions with the first hash function, as long as the second hash function is 5-universal. This means that the intermediate domain can be n times smaller. Such a smaller intermediate domain typically means that both the first and the second hash function are simpler and run at least twice as fast. The same doubling of hashing speed for $O(1)$ expected probes follows for most domains bigger than 32-bit integers, e.g., fixed length strings and even the important case of 64-bit integers which may be stored directly in the array.

A different way of viewing our contribution is to consider the extra price paid for constant expected linear probing on worst-case input. For any efficient implementation of hash tables, we need a hash function h_1 into say $[2n] = \{0, \dots, 2n-1\}$ where each key has $O(1)$ expected collisions. Our result is that for linear probing with constant expected cost, it suffices to compose h_1 with a 5-universal hash function $h_2 : [2n] \rightarrow [2n]$. Using the fast implementation of h_2 from [19], the extra price of h_2 is much less than that of hashing a longer string or that of a single cache miss.

Our general contribution here is to prove good performance for a specific problem for a low-complexity hash function followed by a high complexity one, where the first hash function is not expected collision-free. We hope that this approach can be applied elsewhere to

speed up hashing for specific problems.

Additional results We extend our analysis to two scenarios not considered by Pagh et al. [12]. One is if the size of the array is large compared with the number of elements. The other is when we store strings directly as intervals of the array. Our results for these cases are thus also new in the case of a single direct 5-universal hash function from the original domain to locations in the array.

Contents In Section 2 we state our formal result and in Sections 3-4 we prove it. In Section 5 we show how to apply our result to different domains and how the traditional solution with a first collision free hashing into a larger intermediate domain would be at least twice as slow. In Section 6 we consider the option of storing strings as intervals of the linear probing array.

2 The formal result

As described above, we use linear probing to place a dynamic set of keys $S \subseteq U$ of size $|S| \leq n$ in an array T with $t > |S|$ locations $[t] = \{0, \dots, t-1\}$. We will often refer to $\alpha = n/t$ as the *fill*. Each key $x \in S$ is given a unique location $\wp(x) \in [t] = \{0, \dots, t-1\}$. Location arithmetic is done modulo t , so if $i < j$ then $[j, i] = \{j, \dots, t-1, 0, \dots, i\}$. If Q is a set of locations, then $i + Q = \{i + q | q \in Q\}$.

The locations depend on a hash function $h : U \rightarrow [t]$. For linear probing, the assignment of locations has to satisfy the following invariant:

INVARIANT 2.1. For every $x \in S$, the interval $[h(x), \wp(x)]$ is full.

Invariant 2.1 implies that to check if x is in S , we can start in location $h(x)$ and search consecutive locations until either x or an empty location i is found. In the later case, we can add x , setting $\wp(x) = i$. Invariant 2.1 also implies that a location i is full if and only if there is an ℓ such that $|S \cap h^{-1}(i - [\ell])| \geq \ell$. Here $S \cap h^{-1}(i - [\ell]) = \{y \in S | i - \ell < h(y) \leq i\}$.

For each key $x \in U$, we let $\text{full}_{h,S}(x)$ denote the number of locations from $h(x)$ to the nearest empty location, or more precisely, the largest ℓ such that $h(x) + [\ell]$ is full. We can now bound the number of locations considered by the different operations:

insert $_{h,S}(x)$ finds empty location for $x \notin S$ in $\text{full}_{h,S}(x) + 1$ probes.

delete $_{h,S}(x)$ considers $\text{full}_{h,S}(x) + 1$ locations, moving keys up to fill gaps created by the deletion.

find $_{h,S}(x)$ uses at most $\text{full}_{h,S \setminus \{x\}}(x) + 1$ probes. If $x \notin S$, it finds an empty location. If $x \in S$, to get to x , it only to passes locations that would be full without x in S .

Our hash function $h : U \rightarrow [t]$ will always be composed of two functions via some intermediate domain A ; namely $h_1 : U \rightarrow A$ and $h_2 : A \rightarrow [t]$. Then $h(x) = h_2 \circ h_1$. For each $x \in U$, let the bucket size $b_1(x)$ be the number of elements $y \in S$ with $h_1(y) = h_1(x)$. Note that $b_1(x)$ counts x if $x \in S$ and that $\sum_{x \in S} b_1(x)$ is the sum of the squares of the bucket sizes. Our main technical result can now be stated as follows:

THEOREM 2.1. Let h_1 be fixed arbitrarily, and let h_2 is be a 5-universal hash function from A to $[t]$ where for every $a \in A$ and $i \in [t]$, the probability that $h_2(a) = i$ is at most α/n . Then, for any $x_0 \in U$,

$$\mathbb{E}[\text{full}_{h,S}(x_0)] = O\left(\frac{b_1(x_0)}{(1-\alpha)} + \frac{\sum_{x \in S} b_1(x)/n}{(1-\alpha)^5}\right)$$

Concerning the factor $(1-\alpha)^{-5}$, we know from Knuth [9] that a factor $(1-\alpha)^{-2}$ is needed even when h_1 is the identity and $h = h_2$ is a truly random function.

Applying Theorem 2.1 to an universal hash function h_1 , we get the corollary below, which is what was claimed in the introduction.

COROLLARY 2.1. Let h_1 be a hash function from U to an intermediate domain A such that the expected bucket size $b_1(x)$ of any key x is constant. Let h_2 be a 5-universal hash function from A to $[t]$ where for every $a \in A$ and $i \in [t]$, the probability that $h_2(a) = i$ is at most α/n for some constant $\alpha < 1$. Then for every $x_0 \in U$, we have $\mathbb{E}[\text{full}_{h,S}(x_0)] = O(1)$.

2.1 Small fill We will also consider cases with fill $\alpha = o(1)$. It is then useful to consider

$$\text{coll}_{h,S}(x) = \text{full}_{h,S}(x) - [x \in S]$$

Here the Boolean expression $x \in S$ has value 1 if $x \in S$; 0 otherwise. We can think of $\text{coll}_{h,S}(x)$ as the number of elements from $S \setminus \{x\}$ that x ‘‘collides’’ with in the longest full interval starting in $h(x)$. Similarly, we define $c_1(x) = b_1(x) - [x \in S]$, which is the standard number of collisions between x and $S \setminus \{x\}$ under h_1 . Note that we may have $\text{full}_{h,S \setminus \{x\}}(x) \ll \text{coll}_{h,S}(x)$ because x may fill a an empty gap between two full intervals.

THEOREM 2.2. Let h_1 be fixed arbitrarily, and let h_2 is be a 5-universal hash function from A to $[t]$ where for every $a \in A$ and $i \in [t]$, the probability that $h_2(a) = i$ is at most α/n with $\alpha \leq 0.9$. Then, for any $x_0 \in U$,

$$\mathbb{E}[\text{full}_{h,S}(x_0)] = b_1(x_0) + O(\sqrt[3]{\alpha}) \left(b_1(x_0) + \sum_{x \in S} b_1(x)/n \right)$$

$$\mathbb{E}[\text{coll}_{h,S}(x_0)] = O\left(c_1(x_0) + \alpha \sum_{x \in S} b_1(x)/n \right)$$

Using the last bound of Theorem 2.2, we get

COROLLARY 2.2. *Let $n \leq 0.9t$ and $t \leq t'$. Let h_1 be a universal hash function from U to $[t']$ and let h_2 be a 5-universal hash function from $[t']$ to $[t]$. Then for every $x_0 \in U$,*

$$\mathbf{E}[\text{coll}_{h,S}(x_0)] = O(n/t).$$

Proof. Since h_1 is universal, we get $\mathbf{E}[c_1(x)] < n/t' \leq n/t = \alpha$ for all $x \in U$. \square

If h_1 is the identity and $h = h_2$, then this is the scenario studied by Pagh et al. [12], except that we have subconstant fill $\alpha = n/t = o(1)$. One can think of $\text{coll}_{h,S}(x)$ as the hashing overhead from not having each key mapped directly to a unique location in the array. Having $\mathbf{E}[\text{coll}_{h,S}(x)] = O(\alpha)$ is clearly best possible within a constant factor.

3 Proof of Theorem 2.1

We will now start proving Theorem 2.1. Let $[h(x) - H, h(x) + L]$ be the longest full interval containing $h(x)$. Then $\text{full}_{h,S}(x) = L$ and we want to limit the expectation of L . As in Theorem 2.1 we have $h = h_2 \circ h_1$ where h_1 is fixed and h_2 is 5-universal.

Define $S_0 = \{x \in S \mid h_1(x) \neq h_1(x_0)\}$. Then $|S_0| + b_1(x_0) = |S|$. Since $[h(x) - H, h(x) + L]$ is exactly full, we have

$$\begin{aligned} H + L &= |S \cap h^{-1}([h(x_0) - H, h(x_0) + L])| \\ &= b_1(x_0) + |S_0 \cap h^{-1}([h(x_0) - H, h(x_0) + L])| \\ &\quad + |S_0 \cap h^{-1}([h(x_0), h(x_0) + L])| \end{aligned}$$

It follows that at least one of the following cases are satisfied:

1. $b_1(x_0) \geq (1 - \alpha)/3 \cdot L$.
2. $|S_0 \cap h^{-1}([h(x_0), h(x_0) + L])| \geq (\alpha + (1 - \alpha)/3) \cdot L$.
3. $|S_0 \cap h^{-1}([h(x_0) - H, h(x_0)])| - H \geq (1 - \alpha)/3 \cdot L$.

We are going to pay L in each of the above cases. In fact, in each case i separately, we are going to seek the largest L_i satisfying the condition, and then we will use $\sum_{i=1}^3 L_i$ as a bound on L . In case 1, we have $L_1 \leq 3b_1(x_0)/(1 - \alpha)$, corresponding to the first term in the bound of Theorem 2.1. To complete the proof, we need to bound the expectations of L_2 and L_3 . We call L_2 the *tail* because it only involves keys hashing to $h(x_0)$ and later, and we call L_3 the *head* because it only involves keys hashing before $h(x_0)$.

3.1 A basic lemma The following lemma will be used to bound the expectations of both L_2 and L_3 .

LEMMA 3.1. *For any $Q \subseteq [t]$ of size q ,*

$$\begin{aligned} (3.1) \quad \Pr[\alpha q + d \leq |S_0 \cap h^{-1}(h(x_0) + Q)| < D] \\ \leq \frac{\alpha q (\sum_{x \in S_0} [b_1(x) < D] b_1^3(x))}{d^4 n} \\ + \frac{3\alpha^2 q^2 (\sum_{x \in S_0} [b_1(x) < D] b_1(x))^2}{d^4 n^2} \end{aligned}$$

Above, the Boolean expression $[b_1(x) < D]$ has value 1 if $b_1(x) < D$; 0 otherwise. Before continuing, note that if h_1 is the identity, that is, if we just used 5-universal hashing, then $b_1(x) = 1$ for all $x \in S$, and then the right hand side reduces to $\frac{\alpha q}{d^4} + \frac{3q^2 \alpha^2}{d^4}$, which is what is proved in [12]. Here we need to handle different bucket sizes $b_1(x)$. For our later analysis, it will be crucial that we have introduced the cut-off parameter D .

Proof of Lemma 3.1. Apart from the cut-off parameter D , our proof is very similar to one used in [12]. Since h_2 is 5-universal, it is 4-universal when we condition on any particular value of $h(x_0) = h_2(h_1(x_0))$, so we can think of $Q_0 = h(x_0) + Q$ as a fixed set and h_2 as 4-universal on $h_1(S_0) = h_1(S) \setminus \{h_1(x_0)\}$.

For each $a \in A$, set $b_a = |\{x \in S_0 \mid h_1(x) = a\}|$ and $\underline{b}_a = [b_a < D] b_a$, observing that

$$|S_0 \cap h^{-1}(Q_0)| < D \Rightarrow |S_0 \cap h^{-1}(Q_0)| = \sum_{a \in A} [h_2(a) \in Q_0] \underline{b}_a$$

Therefore, the event in (3.1) implies

$$(3.2) \quad \alpha q + d \leq \sum_{a \in A} [h_2(a) \in Q_0] \underline{b}_a$$

For each $a \in A$, let $p_a = \Pr[h_2(a) \in Q_0] \leq \alpha q/n$, and consider the random variables

$$X_a = \underline{b}_a ([h_2(a) \in Q_0] - p_a)$$

Then $\mathbf{E}[X_a] = 0$. Set $X = \sum_a X_a$. Then

$$\sum_{a \in A} [h_2(a) \in Q_0] \underline{b}_a = X + \sum_{a \in A} p_a \underline{b}_a \leq X + \alpha q.$$

Hence (3.2) implies

$$(3.3) \quad d \leq X$$

To bound the probability of (3.3), we use the 4th moment inequality $\Pr[X \geq d] \leq \mathbf{E}[X^4]/d^4$. Since $\mathbf{E}[X_i] = 0$ and the X_i are 4-wise independent, we get

$$\begin{aligned} \mathbf{E}[X^4] &= \sum_{a_1, a_2, a_3, a_4 \in A} \mathbf{E}[X_{a_1} X_{a_2} X_{a_3} X_{a_4}] \\ &\leq \sum_{a \in A} \mathbf{E}[X_a^4] + 3 \left(\sum_{a \in A} \mathbf{E}[X_a^2] \right)^2 \end{aligned}$$

Here

$$\sum_{a \in A} \mathbb{E}[X_i^4] < \sum_{a \in A} p_a b_a^4 \leq \frac{\alpha q}{n} \sum_{a \in S_0} ([b_1(a) < D] b_1^3(a))$$

and

$$\begin{aligned} \sum_{i \in I} \mathbb{E}[X_i^2] &\leq \left(\sum_{a \in A} p_a b_a^2 \right)^2 \\ &\leq \left(\frac{\alpha q}{n} \right)^2 \left(\sum_{x \in S_0} ([b_1(x) < D] b_1(x)) \right)^2 \end{aligned}$$

For the probability bound of the lemma, we divide the total expectation of $\mathbb{E}[X^4]$ by d^4 . \square

3.2 The tail L_2

We want to show that

$$(3.4) \quad \mathbb{E}[L_2] = O\left(\frac{\sum_{x \in S} b_1(x)/n}{(1-\alpha)^5}\right).$$

Let

$$W(\ell) = |\{S_0 \cap h^{-1}([h(x_0), h(x_0) + \ell])\}|$$

Then L_2 is the largest value in $[t]$ such that $W(L_2) \geq (\alpha + (1-\alpha)/3)L_2$.

In [12] they have $b_1(x) = [x \in S] \leq 1$ for all $x \in U$, and they use that $\mathbb{E}[L_2] = \sum_{i=1}^{\infty} \Pr[L_2 \geq i]$. However, with some large $b_1(x)$, we have no good bound on $\Pr[L_2 \geq i]$. This is why we introduced our cut-off parameter D in Lemma 3.1.

Let $\ell_0 = \lceil 27/(1-\alpha) \rceil$ and $\ell_i = \lceil \ell_{i-1}/(1-(1-\alpha)/9) \rceil$. Then $\ell_{i-1} > (1-(1-\alpha)4/27)\ell_i$. We say that $i > 0$ is *good* if

$$W(\ell_i) \in [(\alpha + (1-\alpha)/9)\ell_i, (\alpha + (1-\alpha)/3)\ell_i].$$

LEMMA 3.2. *If $\ell_{i-1} \leq L_2 < \ell_i$, then i is good.*

Proof. If $L_2 < \ell_i$ then $W(\ell_i) < (\alpha + (1-\alpha)/3)\ell_i$. Also, we have $W(\ell_i) \geq W(L_2) \geq (\alpha + (1-\alpha)/3)L_2 \geq (\alpha + (1-\alpha)/3)\ell_{i-1} > (\alpha + (1-\alpha)/3)(1-(1-\alpha)4/27)\ell_i > (\alpha + (1-\alpha)/9)\ell_i$. \square

Hence

$$\mathbb{E}[L_2] < \ell_0 + \sum_{i=1}^{\infty} \Pr[i \text{ good}] \ell_i.$$

Here ℓ_0 satisfies (3.4). To bound $\Pr[i \text{ good}]$ we use Lemma 3.1 with $q = \ell_i$, $Q = [q]$, $d = (1-\alpha)/9 \cdot \ell_i$, and $D = \ell_i$. Then

$$\begin{aligned} \Pr[i \text{ good}] &= \frac{\alpha q (\sum_{x \in S_0} [b_1(x) < D] b_1^3(x))}{d^4 n} \end{aligned}$$

$$\begin{aligned} &+ \frac{3\alpha^2 q^2 (\sum_{x \in S_0} [b_1(x) < D] b_1(x))^2}{d^4 n^2} \\ &= O\left(\frac{\sum_{x \in S_0} [b_1(x) < \ell_i] b_1^3(x)}{(1-\alpha)^4 \ell_i^3 n}\right) \\ &\quad + \frac{(\sum_{x \in S_0} [b_1(x) < \ell_i] b_1(x))^2}{(1-\alpha)^4 \ell_i^2 n^2} \end{aligned}$$

Hence

$$\begin{aligned} &\sum_i \Pr[i \text{ good}] \ell_i \\ &= \sum_i O\left(\frac{\sum_{x \in S_0} [b_1(x) < \ell_i] b_1^3(x)}{(1-\alpha)^4 \ell_i^3 n}\right) \\ &\quad + \frac{(\sum_{x \in S_0} [b_1(x) < \ell_i] b_1(x))^2}{(1-\alpha)^4 \ell_i n^2} \\ (*) &= \sum_{x \in S_0} \sum_{i: \ell_i > b_1(x)} O\left(\frac{b_1^3(x)}{(1-\alpha)^4 \ell_i^3 n}\right) \\ &\quad + \frac{b_1(x)}{(1-\alpha)^4 \ell_i n^2} \sum_{x \in S_0} b_1(x) \\ &= \sum_{x \in S_0} O\left(\frac{b_1(x)}{(1-\alpha)^5 n} + \frac{1}{(1-\alpha)^5 n^2} \sum_{x \in S_0} b_1(x)\right) \\ &= O\left(\frac{\sum_{x \in S_0} b_1(x)}{(1-\alpha)^5 n}\right). \end{aligned}$$

Above, the (*) marks the point at which we exploit our cut-off $[b_1(x) < \ell_i]$. This completes the proof of (3.4). Below we are going to need several similar calculations, but with different parameters, carefully chosen depending on the context. The presentation will be focused on the differences, only sketching the similar parts.

3.3 The head L_3

We want to show that

$$(3.5) \quad \mathbb{E}[L_3] = O\left(\frac{\sum_{x \in S} b_1(x)/n}{(1-\alpha)^5}\right).$$

Here L_3 is the largest value such that from some H , we have $|S_0 \cap h^{-1}([h(x_0) - H, h(x_0)])| - H \geq (1-\alpha)/3 \cdot L_3$. We want to bound the expectation of L_3 . The argument is basically a messy version of that in Section 3.2 for the tail L_2 .

Let $U(H) = |S_0 \cap h^{-1}([h(x_0) - H, h(x_0)])|$. Then H maximizes $D = U(H) - H$, and then $L_3 = 3D/(1-\alpha)$, so we are really trying to bound D . Losing at most a factor 2, it suffices to consider cases where $U(H') \leq 2H'$, for suppose $U(H) > 2H$ and consider the largest H' such that $U(H' - 1) > 2(H' - 1)$. Then $H' - 1 \geq$

$(H + D)/2$ so $U(H' - 1) - (H' - 1) > (H + D)/2$, implying $U(H' - 1) - (H' - 1) \geq D/2 + 1$. Consequently, $D' = U(H') - H' \geq U(H' - 1) - H' \geq D/2$. Thus there is an H' such that $U(H') \leq 2H'$ and such that $U(H') - H' \geq (1 - \alpha)/6 \cdot L_3$. Finally, define R' such that $U(H') = (\alpha + R'(1 - \alpha))H'$. Then $1 \leq R' \leq 2/(1 - \alpha)$ and $L_3 \leq 6(R' - 1)H = O(R'H)$.

For $I = 0, 1, \dots, \lfloor -\log_2(1 - \alpha) \rfloor$, let H_I be the largest value such that $U(H_I) \geq (\alpha + 2^I(1 - \alpha))H_I$. Let I' be such that $R'/2 < 2^{I'} \leq R'$. Then $H_{I'} \geq H'$, so $2^{I'}H_{I'} > R'H'/2$. Thus we conclude that

$$L_3 = O(\max_I \{2^I H_I\}) = O\left(\sum_{I=0}^{\lfloor -\log_2(1-\alpha) \rfloor} 2^I H_I\right).$$

As in Section 3.2, let $\ell_0 = \lceil 27/(1 - \alpha) \rceil$, and $\ell_i = \lceil \ell_{i-1}/(1 - (1 - \alpha)/9) \rceil$. We say that i is I -good if

$$U(\ell_i) \in [(\alpha + 2^I(1 - \alpha)/3)\ell_i, (\alpha + 2^I(1 - \alpha))\ell_i].$$

Similar to Lemma 3.2, we have that i is I -good if $\ell_{i-1} \leq H_I < \ell_i$. Hence

$$\begin{aligned} \mathbb{E}[L_3] &= O\left(\sum_{I=0}^{\lfloor -\log_2(1-\alpha) \rfloor} (2^I(\ell_0 + \sum_i \Pr[i \text{ } I\text{-good}]\ell_i))\right) \\ &= O\left(1/(1 - \alpha)^2\right. \\ &\quad \left. + \sum_{I=0}^{\lfloor -\log_2(1-\alpha) \rfloor} \sum_i (\Pr[i \text{ } I\text{-good}]2^I\ell_i)\right) \end{aligned}$$

The first term satisfies (3.5) so we only have to consider the double sum. To bound $\Pr[i \text{ } I\text{-good}]$ we use Lemma 3.1 with $q = \ell_i$, $Q = [q]$, $d = 2^I(1 - \alpha)/3 \cdot \ell_i$, and $D = 2^I\ell_i$. With calculations similar to those in Section 3.2, we get

$$\begin{aligned} &\sum_i (\Pr[i \text{ } I\text{-good}]2^I\ell_i) \\ &= \sum_i O\left(\frac{\sum_{x \in S_0} [b_1(x) < 2^I\ell_i] b_1^3(x)}{(1 - \alpha)^4 2^{3I} \ell_i^2 n}\right. \\ &\quad \left. + \frac{(\sum_{x \in S_0} [b_1(x) < 2^I\ell_i] b_1(x))^2}{(1 - \alpha)^4 2^{3I} \ell_i n^2}\right) \\ &= O\left(\frac{\sum_{x \in S_0} b_1(x)}{2^I(1 - \alpha)^5 n}\right). \end{aligned}$$

It follows that

$$\sum_{I=0}^{\lfloor -\log_2(1-\alpha) \rfloor} \sum_i (\Pr[i \text{ } I\text{-good}]2^I\ell_i)$$

$$\begin{aligned} &= \sum_{I=0}^{\lfloor -\log_2(1-\alpha) \rfloor} O\left(\frac{\sum_{x \in S_0} b_1(x)}{2^I(1 - \alpha)^5 n}\right) \\ &= O\left(\frac{\sum_{x \in S_0} b_1(x)}{(1 - \alpha)^5 n}\right). \end{aligned}$$

This completes the proof of (3.5), hence of Theorem 2.1.

4 Proof of Theorem 2.2

We will now prove Theorem 2.2. The bound coincides with that of Theorem 2.1 when $\alpha < 1$ is a constant, so we may assume $\alpha = o(1)$.

We will use the same basic definitions as in the proof of Theorem 2.1. Recall that $h = h_2 \circ h_1$ where h_1 is fixed and h_2 is 5-universal. Define $S_0 = \{x \in S | h_1(x) \neq h_1(x_0)\}$. We let $[h(x) - H, h(x) + L]$ be the longest full interval containing $h(x)$. Then $\text{full}_{h,S}(x) = L$ and we want to limit the expectation of L .

We introduce a parameter β , and note that at least one of the following cases are satisfied:

4. $b_1(x_0) \geq (1 - \alpha)(1 - \beta) \cdot L$.
5. $|S_0 \cap h^{-1}([h(x_0), h(x_0) + L])| \geq (\alpha + (1 - \alpha)\beta/2) \cdot L$.
6. $|S_0 \cap h^{-1}([h(x_0) - H, h(x_0)])| - H \geq (1 - \alpha)\beta/2 \cdot L$.

With $\beta = 2/3$, the above coincides with cases 1-3 from Section 3. However, in our analysis below, we will have $\alpha < \beta/8$ and $\beta \leq 2/5$.

In each case i separately, we are going to seek the largest L_i satisfying the condition, and then we will use $\sum_{i=4}^6 L_i$ as a bound on L . In case 4, we trivially have

$$(4.6) \quad L_4 \leq b_1(x_0)/((1 - \alpha)(1 - \beta)).$$

We are going to bound L_5 and L_6 by

$$(4.7) \quad L_5 + L_6 = O\left(\frac{\alpha}{\beta^2} \sum_{x \in S} b_1(x)/n\right).$$

We will now show how (4.6) and (4.7) imply Theorem 2.2. For the bound on $\text{full}_{h,S}(x_0)$ we set $\beta = \alpha^{1/3}$. Then $L_4 \leq b_1(x_0)/((1 - \alpha)(1 - \sqrt[3]{\alpha})) = b_1(x_0)(1 + O(\sqrt[3]{\alpha}))$ while $L_5 + L_6 = O(\sqrt[3]{\alpha} \sum_{x \in S} b_1(x)/n)$.

Getting the bound on $\text{coll}_{h,S}(x_0)$ is a bit more subtle. We use $\beta = 2/5$ and $\alpha < 1/6$. Then $L_4 \leq b_1(x_0)/((1 - \alpha)(1 - \beta)) < b_1(x_0)/(5/6 \cdot 3/5) = 2b_1(x_0)$. Since L_4 and $b_1(x_0)$ are integer, we get that $L_4 = b_1(x_0)$ for $b_1(x_0) \leq 1$, and $L_4 - 1 \leq 3(b_1(x_0) - 1)$ for $b_1(x_0) > 1$. Since $x_0 \in S$ implies $b_1(x_0) \geq 1$, we get

$$L_4 - [x_0 \in S] \leq 3(b_1(x_0) - [x_0 \in S]) = 3c_1(x_0).$$

Hence

$$\text{coll}_{h,S}(x_0) = \text{full}_{h,S}(x_0) - [x_0 \in S]$$

$$\begin{aligned} &\leq L_4 - [x_0 \in S] + L_5 + L_6 \\ &\leq 3c_1(x_0) + O(\alpha \sum_{x \in S} b_1(x)/n). \end{aligned}$$

Thus Theorem 2.2 follows from (4.6) and (4.7). It remains to bound the expectations of L_5 and L_6 as in (4.7). We call them the *small* head and tail because they are much smaller now that $\alpha = o(1)$.

4.1 The small tail L_5

We want to show that

$$(4.8) \quad \mathbb{E}[L_5] = O\left(\frac{\alpha}{\beta^2} \sum_{x \in S} b_1(x)/n\right).$$

using $\alpha \ll \beta < 1$. The proof is very similar to that in Section 3.2. Let

$$W(\ell) = |\{S_0 \cap h^{-1}([h(x_0), h(x_0) + \ell])\}|$$

Then L_2 is the largest value in $[t]$ such that $W(L_2) \geq (\alpha + (1 - \alpha)\beta/2)L_5 > \beta/2 \cdot L_5$.

This time, for $i = 0, 1, \dots$, we define $\ell_i = 2^i$, which is a much faster growth than it had in Section 3.2. We say that $i > 0$ is *good* if

$$W(\ell_i) \in [\beta/4 \cdot \ell_i, \beta\ell_i].$$

Similar to Lemma 3.2, we have that i is good if $\ell_{i-1} \leq L_5 < \ell_i$. Hence

$$\mathbb{E}[L_5] \leq \sum_{i=0}^{\infty} (\Pr[i \text{ good}] \ell_i).$$

To bound $\Pr[i \text{ good}]$ we use Lemma 3.1 with $q = \ell_i$, $Q = [q]$, $d = (\beta/4 - \alpha)\ell_i \geq \beta/8 \cdot \ell_i$ for $\alpha \leq \beta/8$, and $D = \beta\ell_i$. With calculations similar to those in Section 3.2, we get

$$\begin{aligned} &\sum_i (\Pr[i \text{ good}] \ell_i) \\ &= \sum_i O\left(\frac{\alpha \sum_{x \in S_0} [b_1(x) < \beta\ell_i] b_1^3(x)}{\beta^4 \ell_i^2 n} + \frac{\alpha^2 (\sum_{x \in S_0} [b_1(x) < \beta\ell_i] b_1(x))^2}{\beta^4 \ell_i n^2}\right) \\ &= O\left(\frac{\alpha \sum_{x \in S_0} b_1(x)}{\beta^2 n}\right). \end{aligned}$$

This completes the proof of (4.8).

4.2 The small head L_6

We want to show that

$$(4.9) \quad \mathbb{E}[L_6] = O\left(\frac{\alpha}{\beta} \sum_{x \in S} b_1(x)/n\right).$$

Here L_6 is the largest value such that for some H_6 , we have $|S_0 \cap h^{-1}([h(x_0) - H_6, h(x_0)])| - H_6 \geq (1 - \alpha)\beta/2 \cdot L_6$.

Define $U(H) = |S_0 \cap h^{-1}([h(x_0) - H, h(x_0)])|$. We will look for the largest H' such that $U(H') \geq H'$. Note that $U(H_6 + (1 - \alpha)\beta/2 \cdot L_6) \geq U(H_6) \geq H_6 + (1 - \alpha)\beta/2 \cdot L_6$. Hence $H' \geq H_6 + (1 - \alpha)\beta/2 \cdot L_6$, and therefore $L_6 = O(H'/\beta)$.

As in the previous section, we define $\ell_i = 2^i$. This time we say that i is *good* if

$$U(\ell_i) \in [\ell_i/2, \ell_i].$$

Similar to Lemma 3.2, we have that i is good if $\ell_{i-1} \leq H' < \ell_i$. Hence

$$\mathbb{E}[L_3] = O(\mathbb{E}[H']/\beta) = \sum_{i=0}^{\infty} (\Pr[i \text{ good}] \ell_i/\beta).$$

To bound $\Pr[i \text{ good}]$ we use Lemma 3.1 with $q = \ell_i$, $Q = [q]$, $d = (1/2 - \alpha)\ell_i \geq 1/3 \cdot \ell_i$ for $\alpha \leq 1/6$, and $D = \ell_i$. With calculations similar to those in Section 3.2, we get

$$\begin{aligned} &\sum_i (\Pr[i \text{ good}] \ell_i/\beta) \\ &= \sum_i O\left(\frac{\alpha \sum_{x \in S_0} [b_1(x) < 2\ell_i] b_1^3(x)}{\ell_i^2 n} + \frac{\alpha^2 (\sum_{x \in S_0} [b_1(x) < 2\ell_i] b_1(x))^2}{\ell_i n^2}\right) \\ &= O\left(\frac{\alpha \sum_{x \in S_0} b_1(x)}{\beta n}\right). \end{aligned}$$

This completes the proof of (4.9), hence of Theorem 2.2.

5 Application

We will now show how to apply our result to different domains and how the traditional solution with a first collision free hashing into a larger intermediate domain would be at least twice as slow. First we consider 64-bit integers, then fixed length strings, and finally variable length strings. We assume that the implementation is done in a programming language like C [8] or C++ [18] leading to efficient and portable code that can be inlined from many other programming languages.

We note that this section in itself has no theoretical contribution. It is demonstrating how the theoretical result of the previous section plays together with the fastest existing techniques, yielding simpler code running at least twice as fast. For the reader less familiar with efficient hashing, this section may serve as a mini-survey of hashing that is both fast and theoretically good.

Recall the general situation: We are going to hash a subset S of size n of some domain U . The hash range should be indices of the linear probing array. For simplicity, the range will be $[2n]$, and we assume that $2n$ is a power of 4.

Our hash function $h : U \rightarrow [2n]$ will always be composed of two functions via some intermediate domain $[m]$; namely $h_1 : U \rightarrow [m]$ and $h_2 : [m] \rightarrow [2n]$. Then $h(x) = h_2(h_1(x))$.

With the traditional method, we want h_2 to be collision free w.h.p., which means $m_{old} = \omega(n^2)$. From our Corollary 2.1, it follows that for expected constant number of probes, it suffices with expected constant bucket size, and hence it suffices with $m_{new} = O(n)$. Thus $m_{new} = o(\sqrt{m_{old}})$. As a concrete example, we will think of $m_{new} = 2^{30}$ and $m_{old} = 2^{60}$. We denote the corresponding bit-lengths $\ell_{new} = 30$ and $\ell_{old} = 60$.

5.1 5-universal hashing from the intermediate domain Below we consider two different methods for 5-universal hashing.

Degree 4 polynomial The classic method [20] for 5-universal hashing is to use a random degree 4 polynomial $h_{a_0, \dots, a_4}(x) = \sum_{i=0}^4 a_i x^i$ over \mathbb{Z}_p . To get a fast mod p operation, Carter and Wegman [3] suggest using a Mersenne prime p such as $2^{31} - 1$, $2^{61} - 1$, $2^{89} - 1$, or $2^{107} - 1$. The point is that with $p = 2^a - 1$, we can exploit that $y = (y \& p) + (y \gg a) \pmod{p}$ where \gg is right shift and $\&$ is bit-wise and. At the end, we extract the ℓ least significant bits.

A typical situation would be that $m_{new} < p_{new} = 2^{31} - 1$ while $2^{31} - 1 < m_{old} < p_{old} = 2^{61} - 1$. For multiplications in $\mathbb{Z}_{2^{31}-1}$, we can use that 64-bit multiplication provides exact 32-bit multiplication. However, for multiplication in $\mathbb{Z}_{2^{61}-1}$, we have the problem that 64-bit multiplication discards overflow, so we need 4 64-bit multiplications to get the full answer. For larger n , we might have $p_{new} = 2^{61} - 1$ and $p_{old} \in \{2^{81} - 1, 2^{107} - 1\}$, but, we still need more than twice as many 64-bit multiplications with the large old domain.

Tabulation More recently [19] suggested a tabulation based approach to 5-universal hashing which is roughly 5 times faster than the above approach¹.

The simplest case is that we divide a key $x \in [m]$ into 2 characters $x_0 x_1$. The hash function is computed as $T_0[x_0] \oplus T[x_1] \oplus T[x_0 + x_1]$. Here \oplus is bit-wise xor and T_0, T_1, T_2 are independently tabulated 5-universal hash functions. T_0 and T_1 have \sqrt{m} entries, and T_2 has $2\sqrt{m}$ entries. This is fast if $O(\sqrt{m})$ is small enough to fit in

fast memory. The experiments in [19] found this to be the case for $m = 2^{32}$, gaining a factor 5 in speed over the polynomial-based method above. This is perfect for our new small intermediate domain ($m = m_{new} = 2^{30}$), but not with the old large intermediate domain ($m = m_{old} = 2^{60}$).

The method in [19] is generalized so that we can divide the key into q characters, and then perform $2q - 1$ look-ups in tables of size $\leq 2m^{1/q}$, but the method is more complicated for $q > 2$. Assuming that we want the same small table size with the old and the new method, this means that $q_{old} \geq 2q_{new}$, hence that the large old domain will require more than twice as many look-ups and twice as much space.

5.2 64-bit integers We only need universal hashing from 64-bit integers to ℓ -bit integers, so we can use the method from [6]: We pick a random odd 64-bit integer a , and compute $h_a(x) = a * x \gg (64 - \ell)$. This method actually exploits that the overflow from $a * x$ is discarded. The probability of collision between any two keys is at most $1/2^\ell$. We use this method for h_1 using $\ell_{new} = 30$ in the new approach and $\ell_{old} = 60$ in the old approach. The value of ℓ does not affect the speed of h_1 , but h_1 is very fast compared with the 5-universal h_2 . The overall hashing time is dominated by h_2 which we above found more than twice as fast with our smaller intermediate domain.

5.3 Fixed length strings We now consider the case that the input domain is a string of $2r$ 32-bit characters. For h_1 we will use a straightforward combination of a trick for fast signatures in [1] with the 2-universal hashing from [4]. The combination is folklore [13]. Thus, our input is $x = x_0 \cdots x_{2r-1}$, where x_i is a 32-bit integer. The hash function is defined in terms of $2r$ random 64-bit integers a_0, \dots, a_{2r-1} . The hash function is computed as

$$h_{a_0, \dots, a_{2r-1}}(x_0 \cdots x_{2r-1}) = \left(\bigoplus_{i=0}^{r-1} (x_{2i} + a_{2i}) * (x_{2i+1} + a_{2i+1}) \right) \gg (64 - \ell)$$

This method only works if $\ell \leq 33$, but then the collision probability for any two keys is $\leq 2^{-\ell}$. The cost of this function is dominated by the r 64-bit multiplications.

We can use the above method directly for our $\ell_{new} = 30$. However, for $\ell_{old} = 60$, the best we can do is to apply the above type of hash function twice, using a new set of random indices a'_0, \dots, a'_{2r-1} for the second application, and then concatenate the resulting hash values, thus getting an output of $2\ell_{new} = \ell_{old}$ bits. Thus for the initial hashing h_1 of fixed length strings, we gain

¹The method is designed for 4-universal hashing, but as pointed out in [12] it works unchanged for 5-universal hashing.

a factor 2 in speed with our new smaller intermediate domain.

5.4 Variable length strings For the initial hashing h_1 of variable length strings $x = x_0x_1 \cdots x_v$, we can use the method from [5]. We view x_0, \dots, x_v as coefficients of a polynomial over \mathbb{Z}_p , assuming $x_0, \dots, x_v \in [p]$. We pick a single random $a \in [p]$, and compute the hash function

$$h_a(x_0 \cdots x_v) = \sum_{i=0}^v x_i a^i.$$

If y is another string that is no longer than x , then $\Pr[h_a(x) = h_a(y)] \leq v/p$. As usual, the computations are faster if we use Mersenne primes. In this case, we would probably use $p = 2^{61} - 1$ and 32-bit characters x_i , regardless of the size of the intermediate domain. We will hash down to the right size using the 64-bit hashing from Section 5.2.

As stated above, our smaller intermediate domain does not make any difference in speed, but if we want fast hashing of longer variable length strings, then the above method is too slow. For higher speed, we divide the input string into chunks of, say, ten 32-bit characters, and apply the fixed length hashing from Section 5.3 to each chunk, thus getting a reduced string of chunk hash values. If we have more than one chunk, we apply the above variable length string hashing to the string of chunk hash values. Note that two strings remain different under this reduction if there is a chunk in which they differ, and the hashing of that chunk preserves the difference. Hence we can apply exactly the same hash function to each chunk.

The overall time bound is dominated by the length reducing chunk hashing, and here, with the old method, we should hash chunks to 64 bits while we with the new method can hash chunks to 32 bits. As discussed in Section 5.3, the new method gains a factor 2 in speed.

6 Storing variable length strings directly

Above we assumed that each string x had a single array entry containing its hash value and a pointer to x . Pagh [13] suggested analyzing the alternative where we store variable length strings directly as intervals of the array we probe. Each string is terminated by an end-of-string character $\backslash 0$. If a string x is in the array, it should be preceded by either $\backslash 0$ or an empty location, and it should be located between $h(x)$ and the first empty location after $h(x)$, as stated by Invariant 2.1. We insert a new key x starting at the first empty location after $h(x)$. We may have to push some later keys further back to make room, but all in all, we only consider locations between $h(x)$ and the first empty location after $h(x)$ in

the state after x has been inserted. The advantage to the above direct representation is that we do not need to follow a pointer to get to x . Deletions are implemented using the same idea.

As in the previous sections, we let $\text{full}_{h,S}(x)$ denote the largest ℓ such that $h(x) + [\ell]$ is full. Then $\text{full}_{h,S}(x) + 1$ bounds the number of locations considered by any operation associated with x . In this measure we should include x in S if x is inserted or deleted. By a simple reduction to Theorem 2.2, we will prove

THEOREM 6.1. *Let $S \subseteq U$ be the set of variable length strings x . Let $\text{length}(x)$ be the length of x including an end-of-string character $\backslash 0$. Suppose the total length of the strings in S is at most αt for some $\alpha < 0.9$. Let h_1 be a universal hash function from U to $[t]$, and let h_2 be a 5-universal hash function from $[t]$ to $[t]$. Then for every $x_0 \in U$, we have*

$$(6.10) \quad \begin{aligned} \mathbb{E} [\text{full}_{h,S}(x_0)] &= \text{length}(x_0) \\ &+ O(\sqrt[3]{\alpha}) \left(\text{length}(x_0) + \frac{\sum_{x \in S} \text{length}^2(x)}{\sum_{x \in S} \text{length}(x)} \right) \end{aligned}$$

Moreover, when $x_0 \notin S$,

$$(6.11) \quad \mathbb{E} [\text{full}_{h,S}(x_0)] = O \left(\alpha \frac{\sum_{x \in S} \text{length}^2(x)}{\sum_{x \in S} \text{length}(x)} \right)$$

To bound the number of cells considered when inserting or deleting x_0 , we use (6.10) and get a bound of

$$\text{length}(x_0) + O(\sqrt[3]{\alpha}) \left(\text{length}(x_0) + \frac{\sum_{x \in S} \text{length}^2(x)}{\sum_{x \in S} \text{length}(x)} \right).$$

For finding x_0 we can use the stronger bound from (6.11), yielding

$$\begin{aligned} &1 + O \left(\alpha \frac{\sum_{x \in S} \text{length}^2(x)}{\sum_{x \in S} \text{length}(x)} \right) \quad \text{if } x_0 \notin S \\ &\text{length}(x_0) + O \left(\alpha \frac{\sum_{x \in S} \text{length}^2(x)}{\sum_{x \in S} \text{length}(x)} \right) \quad \text{if } x_0 \in S \end{aligned}$$

Proof of Theorem 6.1. To apply Theorem 2.2, we simply think of all characters of a key x as hashing individually to $h(x)$. The filling of the array is independent of the order in which elements are inserted, so as long as we satisfy Invariant 2.1, it doesn't matter for full that we want characters of x to land at consecutive locations. More specifically, we let the hash function h_1 apply this way to the individual characters. Recall here that h_1 is

arbitrary in Theorem 2.2. We get that

$$\begin{aligned} & \mathbb{E}[\text{full}_{h,S}(x_0)] - b_1(x_0) \\ &= O(\sqrt[3]{\alpha}) \left(b_1(x_0) + \frac{\sum_{a \in x \in S} b_1(x)}{|\{a \in x \in S\}|} \right) \\ &= O(\sqrt[3]{\alpha}) \left(b_1(x_0) + \frac{\sum_{x \in S} \text{length}(x) b_1(x)}{\sum_{x \in S} \text{length}(x)} \right), \end{aligned}$$

Now $b_1(x)$ is the total length of strings $y \in S$ with $h_1(y) = h_1(x)$. Since h_1 is universal, for each string x ,

$$\begin{aligned} & \mathbb{E}[b_1(x)] \\ &= [x \in S] \text{length}(x) + \sum_{y \in S \setminus \{x\}} \text{length}(y)/t \\ &\leq [x \in S] \text{length}(x) + \alpha. \end{aligned}$$

By linearity of expectation, (6.10) follows of the theorem.

For the bound (6.11) when $x_0 \notin S$, we use the bound on $\text{coll}_{h,S}(x_0)$ from Theorem 2.2. When $x_0 \notin S$, we have $\text{coll}_{h,S}(x_0) = \text{full}_{h,S}(x_0)$ and $c_1(x_0) = b_1(x_0)$ so $\mathbb{E}[c_1(x_0)] = \alpha$. We therefore get

$$\begin{aligned} & \mathbb{E}[\text{full}_{h,S}(x_0)] = \mathbb{E}[\text{coll}_{h,S}(x_0)] \\ &= c_1(x_0) + O(\alpha) \frac{\sum_{a \in x \in S} b_1(x)}{|\{a \in x \in S\}|} \\ &= O\left(\alpha \frac{\sum_{x \in S} (\text{length}(x))^2}{\sum_{x \in S} \text{length}(x)}\right). \end{aligned}$$

□

It is easy to see that the bound of Theorem 6.1 is tight within a constant factor even if a truly random hash function h is used. The first term is trivially needed. For the second term, the basic point in the square is that the probability of hitting y is proportional to its length, and so is the expected cost of hitting y . This direct storage of strings was not considered by Pagh et al. [12], so this is the first proof that limited randomness suffices in this case.

The bound of Theorem 6.1 gives a large penalty for large strings, and one may consider a hybrid approach where one for strings of length bigger than some parameter ℓ store a pointer to the suffix, that is, if the string does not end after ℓ characters, then the next characters represent a pointer to a location with the remaining characters. Then it is only for longer strings that we need to follow a pointer, and the cost of that can be amortized over the work on the first ℓ characters.

References

- [1] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: fast and secure message authentication. In *Proc. 19th CRYPTO*, pages 216–233, 1999.
- [2] J. R. Black, C. U. Martel, and H. Qi. Graph and hashing algorithms for modern architectures: Design and performance. In *Proc. 2nd WAE*, pages 37–48, 1998.
- [3] J. Carter and M. Wegman. Universal classes of hash functions. *J. Comp. Syst. Sci.*, 18:143–154, 1979. Announced at STOC’77.
- [4] M. Dietzfelbinger. Universal hashing and k -wise independent random variables via integer arithmetic without primes. In *Proc. 13th STACS, LNCS 1046*, pages 569–580, 1996.
- [5] M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial hash functions are reliable (extended abstract). In *Proc. 19th ICALP, LNCS 623*, pages 235–246, 1992.
- [6] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *J. Algorithms*, 25:19–51, 1997.
- [7] G. L. Heileman and W. Luo. How caching affects hashing. In *Proc. 7th ALENEX*, pages 141–154, 2005.
- [8] B. Kernighan and D. Ritchie. *The C Programming Language (2nd ed.)*. Prentice Hall, 1988.
- [9] D. E. Knuth. Notes on “open” addressing, 1963. Unpublished memorandum. Available at <http://citeseer.ist.psu.edu/knuth63notes.html>.
- [10] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.
- [11] M. Mitzenmacher and S. P. Vadhan. Why simple hash functions work: exploiting the entropy in a data stream. In *Proc. 19th SODA*, pages 746–755, 2008.
- [12] A. Pagh, R. Pagh, and M. Ruzic. Linear probing with constant independence. In *Proc. 39th STOC*, pages 318–327, 2007.
- [13] R. Pagh. Personal communication, 2008.
- [14] R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithms*, 51:122–144, 2004.
- [15] H. Prodinger and W. Szpankowski. Preface for special issue on average case analysis of algorithms). *Algorithmica*, 22(4):363–365, 1998.
- [16] J. P. Schmidt and A. Siegel. The analysis of closed hashing under limited randomness. In *Proc. 22nd STOC*, pages 224–234, 1990.
- [17] A. Siegel and J. Schmidt. Closed hashing is computable and optimally randomizable with universal hash functions. Technical Report TR1995-687, Currant Institute, 1995.
- [18] B. Stroustrup. *The C++ Programming Language, Special Edition*. Addison-Wesley, Reading, MA, 2000.
- [19] M. Thorup and Y. Zhang. Tabulation based 4-universal hashing with applications to second moment estimation. In *Proc. 15th SODA*, pages 615–624, 2004.
- [20] M. Wegman and J. Carter. New hash functions and their use in authentication and set equality. *J. Comp. Syst. Sci.*, 22:265–279, 1981.