On Deletions in Open Addressing Hashing*

Rosa M. Jiménez[1] and Conrado Martínez[1]

Dept. Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, E-08034 Barcelona, Spain.
{jimenez,conrado}@lsi.upc.edu

Abstract. Deletions in open addressing tables have often been seen as problematic. The usual solution is to use a special mark 'deleted' so that probe sequences continue past deleted slots, as if there was an element still sitting there. Such a solution, notwithstanding is wide applicability, may involve serious performance degradation. In the first part of this paper we review a practical implementation of the often overlooked deletion algorithm for linear probing hash tables, analyze its properties and performance, and provide several strong arguments in favor of the Robin Hood variant. In particular, we show how a small variation can yield substantial improvements for unsuccesful search. In the second part we propose an algorithm for true deletion in open addressing hashing with secondary clustering, like quadratic hashing. As far as we know, this is the first time that such an algorithm appears in the literature. Although it involves some extra memory for bookkeeping, the algorithm is comparatively easy and efficient, and might be of practical value, besides its theoretical interest.

1 Introduction

In open addressing hashing, a collection of n items is stored in an array t of M entries (also called *slots*), say $t[0]$ to $t[M-1]$, and *collisions*—that is, when two or more items should go into the same slot—are resolved by finding alternative locations within the table for the colliding items. In particular, given some item x, a sequence of slots at locations $i_0 = h(x)$, i_1, i_2, ... is examined (*probed*) until the item x or an empty[1] slot $t[i_j]$ is found. In the former case the search is successful; in the later, the search is unsuccessful and the empty slot may be used to store x into the table. Here, $h : \mathcal{U} \to [0..M-1]$ denotes the hash function, mapping the universe $\mathcal{U}$ from which the items are drawn to the locations of the hash table t. Different open addressing schemes differ in the way the probe sequences are defined. Furthermore, for any open addressing scheme, different variants can be obtained by changing the rule that specifies how to resolve a collision for a given slot, in particular, which item remains and which one continues with the probe sequence during an insertion. When collisions are always resolved in favor of the item that occupies the table and it is the new item the one that continues with the probe sequence, we have the standard variant, often nicknamed FCFS (First-Come-First-Served). When the collisions are resolved in favor of the newcomer, we have LCFS (Last-Come-First-Served), a variant introduced by Poblete and Munro [14]. In the Robin Hood variant, collisions are resolved in favor of the item which is further away from its hash address [4, 2], whereas in ordered hashing [1], the smaller item—according to some prespecified total order—remains and the greater continues with the probe sequence. Along the paper we will usually consider all these possible strategies, but special emphasis will be given to FCFS and to Robin Hood.

Deletions in open addressing hash tables have been seen as problematic by many authors. Indeed, we cannot remove an item x by simply marking the slot that contains it as empty. If we do that, later operations could be wrong. For instance, a search for an item y might report failure because the emptied slot appears in the probe sequence, even though the item y is present in the table, occurring somewhere later in the probe sequence.

Some authors thus argue that open addressing schemes are not suitable if deletions must be supported. A possible solution, often proposed in data structures textbooks, is to have three possible states for every slot: occupied, free or deleted. When a probe sequence passes through a deleted slot its contents are not checked, but the probes continue. If we want to insert a new element, we can use any deleted slot that we might have seen while searching for the item or a free slot (the search only stops if the item or a free slot is found). This scheme is usually called *lazy deletion*. It is certainly lacking elegance and, worst of all, the performance of the hash table might substantially degrade if many deletions take place. Additionally, a precise mathematical analysis of the performance in the presence of insertions and lazy deletions is quite complex [3, 15].

* This research was supported by project TIN2010-17254 (FRADA) from the Spanish Ministry of Science and Innovation.

[1] For simplicity, we will assume that there is always at least one empty slot, $n < M$.

Some variants of lazy deletion such as those proposed in [7] and [13] improve the overall performance of the hash table—as compared to using standard lazy deletion. But they still exhibit the main drawbacks of lazy deletion, even though in a lesser degree.

Yet, in the case of the simplest open addressing scheme, linear probing, deletions can be supported rather easily and efficiently. In linear probing hashing, the sequence of probes is defined by

$$i_j = (i_0 + j) \bmod M,$$

that is, after probing the hash location $i_0 = h(x)$, we examine the succesive positions i_0+1, i_0+2, ... in a cyclic fashion, until x or an empty slot is found. For simplicity, we will use for the rest of the paper the notation $x \oplus y := (x+y) \bmod M$ and $x \ominus y := (x-y) \bmod M$.

Our contribution in this paper is two-fold. First, we review the often overlooked deletion algorithm for linear probing hashing, its properties, and carry out a detailed analysis of its expected performance for several variants of linear probing. Such analysis, as far as we know, was only partial (for standard FCFS linear probing) or missing (for variants such as LCFS or Robin Hood). We discuss then a few implementation issues of this deletion algorithm. In particular, we argue strongly in favour of the Robin Hood variant of linear probing; among other reasons we introduce a new simple algorithm that improves significantly unsuccesful search cost and analyze its expected cost as a function of the *load factor* $\alpha = n/M$, as is usual for hashing schemes.

Secondly, we propose a new algorithm for deletions in more general open addressing schemes. The new algorithm performs actual deletion in any hashing table using open addressing with *secondary clustering* (here we follow the terminology introduced in [6]). This family of open addressing schemes is characterized by probing one slot using a hash function, and the remaining probes determined by the first one. A particular important instance is *quadratic hashing* [11]. Linear probing is also a special case of open addressing with secondary clustering; it is often said to exhibit *primary clustering*. Note, however, that other well know open addressing schemes, such as *double hashing*, do not fall into this category. In particular, *double hashing* uses an additional hash function to compute the increments in the probe sequence and thus it belongs to the class of open addressing schemes with *tertiary clustering*.

2 Deletions in linear probing hash tables

In this section we consider deletions in linear probing hashing. To delete an item x at $t[i]$, it suffices to scan the non-empty locations following $t[i]$, left to right, and whenever the element y at $t[j]$ hashes to a position $i' = h(y)$ that precedes i and j in the cyclic order, y and the empty slot can be swapped; then the process continues from position $j \oplus 1$ to see if an element can be swapped with the empty slot at $t[j]$, and so on. This algorithm was described in Knuth [8] (Algorithm R), and we do not know of any previous appearance on the literature. It has also been described by several other authors, for instance, Mehlhorn and Sanders [12], but as we have already discussed in the introduction, it has often been overlooked; surprisingly enough, in many well known textbooks on data structures.

A more "extreme" approach, yet simpler and essentially equivalent, is adopted by Sedgewick [16]: each element after the removed one is reinserted into the table. Sedgewick's algorithm is equivalent to Knuth's R algorithm on FCFS, Robin Hood and ordered hashing tables. However, Sedgewick's method is not equivalent for LCFS. Indeed, items with the same hash address appear in reverse order of insertion in the table, and if they are visited in that order and reinserted they'll end up in the order they were inserted.

More recently, Kolosovskiy [9] has proposed the algorithm for deletion sketched above—he seems unaware that the algorithm has been known for a long time—with a new twist. Rather than computing the hash location for each of the items following the removed item to see if they can be reallocated, he proposes to store for each slot the number of probes between that location and the hash location of the item that occupies the slot, if any. Thus, if $t[i]$ has been emptied and $t[j]$ is occupied by an item whose number of probes is larger or equal to the number of probes between i and j, then $t[j]$ can be reallocated to $t[i]$ (see Algorithm 1).

We start reviewing the deletion algorithm for linear probing hash (Algorithm 1), in the form (almost) given by Kolosovskiy [9]. As we have mentioned before, the deletion algorithm for linear probing hashing might be implemented by recomputing hash addresses to know if a given item can be moved to some previous emptied location or not. But we can store additional information at each slot to avoid recomputing hash addresses. For the latter, during a lookup for insertion we need to keep track of the number of probes made so far; when the new item is inserted at, say, $t[i]$, the number of probes needed to reach the slot is stored in $t[i].probes$. This is the variant proposed by Kolosovskiy.

Whether we use Knuth's or Kolosovskiy's variant, Algorithm 1 preserves randomness in a strong sense for all variants mentioned in this paper: FCFS, LCFS, Robin Hood and ordered hashing. For FCFS linear probing, Knuth [8] proves that removing an item with Algorithm 1 yields the same table that we would have obtained if we never had inserted that item. This is also true for the other variants, in particular for last-come-first-served (LCFS) [14] and Robin Hood [2, 4]. Take for instance LCFS and let x be the item to be removed. Let i the location of x in the table. Suppose that the test $t[j].probes > p$ does not succeed as j takes the values $i \oplus 1$, $i \oplus 2$, etc. (and we increment p). That means that i is not in the probe sequence of any of these elements, and the algorithm just leaves $t[i]$ empty, as it should. Suppose now that j is the closest location to i on the cyclic order such that $t[j].probes > p$. That element can occupy $t[i]$ and indeed, the algorihm puts $t[j]$ at $t[i]$ and empties $t[j]$. The question is: is that the element that should be moved to $t[i]$? If so, the rest of the argument simply follows by induction. In a LCFS table, the elements of the table such that i belongs to their probe sequence (in the sense that a search for any of these elements will probe location i) satisfy the property that they appear in reverse order of insertion in the table, when scanning the table from left to right in the cyclic order. Therefore the element at $t[j]$ is, among the elements that probe location i and were inserted before x, the newest one. Therefore to maintain the LCFS invariant, $t[j]$ is the element that must be moved to $t[i]$. Similarly, for a Robin Hood (RH) table, the element $t[j]$ that is moved to $t[i]$ (if such thing happens) is, among all elements probing location i different from x, the one with the smallest displacement (in case of ties, it is the element that is minimum according to that criterium), so the invariant of the RH table is preserved: namely, for any location i, the elements that probe to i appear in consecutive locations and in increasing order of displacement.

As a consequence, all analytical results on linear probing for insertion-only-tables (see, for example, [8, 5] and the references therein) do apply if Algorithm 1 is used for deletions.

Algorithm 1 Removal in a linear probing hash table.

```
procedure REMOVE(k)
        ▷ k: key of the item to remove; t: the hash table
        ▷ M: size of the hash table; n: number of items, n < M
    i ← LOOKUP(k)
        ▷ t[i].key = k ∨ t[i].probes = 0
    if t[i].probes ≠ 0 then
        t[i].probes ← 0
        COMPRESS(i)
        n ← n − 1
procedure COMPRESS(i)
    free ← i; i ← i ⊕ 1; p ← 1
    while t[i].probes ≠ 0 do
        if t[i].probes > p then
                ▷ Item at t[i] can be moved to t[free]
            t[free] ← t[i]
            t[free].probes ← t[i].probes − p
            t[i].probes ← 0
            free ← i; p ← 0
        i ← i ⊕ 1; p ← p + 1
```

The required extra information per slot is not a big disadvantage since we need only to store a short integer per slot. In practice, this extra space will not occupy more memory than the Boolean flag per slot which would be necessary to know whether the slot is empty or not, anyway. On the contrary, keeping the displacement or number of probes of each item is useful to implement the RH variant [2, 4]. Robin Hood hashing has the virtue of significantly reducing the variance of successful searches. And, of course, we are trading time for space, as we avoid recomputing the hash locations of all the items right after the one to be removed; such recomputation of hash address should take significantly more time than the time for bookkeeping of the distances (during both insertions and deletions) and for checking if a key might be reallocated using the `probes` attribute of each slot. Furthermore, the performance of unsuccesful searches and deletions can be greatly improved for RH tables, as we discuss in Subsection 2.2.

2.1 The expected cost of deletions in linear probing

The performance of linear probing hash has been throughly examined and analyzed by several authors. The first analysis is due to Knuth [8]. We give here a brief summary of the most relevant results for our purpose.

Consider a linear probing hash table with M slots and n items, $n < M$. Denote $S_{n,i}$ the number of probes needed to successfully locate the ith item inserted into the table. Denote $U_{n,j}$ the number of probes—including the probe of the empty slot that signals the end of the search—made by an unsucessful search that starts at location j, $0 \le j < M$.

The *total successful search path length* (*tSSPL*) is the sum of all $S_{n,i}$, and the average successful search path length $\overline{S}_n$ is the tSSPL divided by n; we take $\overline{S}_0 = 0$ by convention. Another interesting random variable is $S_n := S_{n,Z_n}$, with $Z_n \sim \text{Uniform}(1..n)$, that is, the successful search path length for a random element. While $\mathbb{E}\{S_n\} = \mathbb{E}\{\overline{S}_n\}$, both random variable are distinct. Similarly, the *total unsuccessful search path length* (*tUSPL*) is the sum of all $U_{n,j}$, and the average unsuccessful search path length $\overline{U}_n$ is the tUSPL divided by M. On the other hand, $U_n := U_{n,Z_M}$, with $Z_M \sim \text{Uniform}(0..M-1)$ is the number of probes when we make an unsuccessful search starting from a random location. Because of symmetry, all $U_{n,j}$ are identically distributed and U_n too; however, they are not independent. On the other hand, $\mathbb{E}\{U_n\} = \mathbb{E}\{\overline{U}_n\}$.

The variable U_n gives also the number of probes I_n to insert a new item in a table that contains n items, when the hash address of the new item is uniformly distributed in $0..M-1$. Since $S_{n,i}$ is the number of probes that was necessary to insert the ith item when the table contained $i-1$ elements it easily follows that

$$\mathbb{E}\{S_n\} = \mathbb{E}\{\overline{S}_n\} = \frac{1}{n}\sum_{1\le i\le n}\mathbb{E}\{S_{n,i}\} = \frac{1}{n}\sum_{0\le i<n}\mathbb{E}\{U_i\}.$$

Since the deletion algorithm preserves randomness, given a table with M slots and n items, we can forget about the particular "history" that produced the table, and analyze it just as if the table had been produced using insertions alone. Now, let $D_{n,i}$ be the number of probes that we need to delete the ith element from such table. This number will consist of two contributions: the $S_{n,i}$ probes made from the item's home location until we reach it at location j, and then the $U_{n,j}$ probes that we must make until we reach the end of the cluster where the item sits. As before, we call the *total deletion path length* (*tDPL*) the sum of all $D_{n,i}$, the average deletion path length $\overline{D}_n$ is the tDPL divided by n, with $D_0 = 0$, and D_n is the cost of the deletion of a random element.

Theorem 1. *For any linear probing hash table T,*

$$tDPL = tUSPL + tSSPL - M.$$

As a consequence, $\mathbb{E}\{\overline{D}_n\} = \mathbb{E}\{D_n\} = \mathbb{E}\{S_n\} + \frac{M}{n}(\mathbb{E}\{U_n\} - 1)$, for $0 < n < M$.

The proof of the theorem is immediate from the observation that if the ith element hashes at position j (with equal probability for the M slots) then $D_{n,i} = S_{n,i} + U_{n,j+S_{n,i}-1} - 1$. Summing over all i, $1 \le i \le n$, yields the statement of the theorem, since the index $j' = j + S_{n,i} - 1$ will span all the indices of the occupied slots in the table. Using well-known results for $\mathbb{E}\{S_n\}$ and $\mathbb{E}\{U_n\}$ [8] we can conclude with the following theorem.

Theorem 2. *The expected cost of deletion of a random element in a linear probing hash table storing of capacity M and storing n items, $0 < n < M$ is*

$$\mathbb{E}\{D_n\} = \frac{1}{2}\sum_{k\ge 1}\frac{(n-1)^{\underline{k}}}{M^k}\left(k+2+\frac{n-1-k}{M}\right) + \frac{1}{2}\left(3-\frac{M}{n}\right) = \frac{1}{2}\frac{(2-\alpha)^2}{(1-\alpha)^2} - \frac{1}{2}\frac{4-\alpha}{(1-\alpha)^4 M} + O(M^{-2}),$$

where $\alpha = n/M \in [0,1)$ is the so-called load factor *and $x^{\underline{k}} = x(x-1)\cdots(x-k+1)$.*

For instance, a deletion in a table with $\alpha = 0.5$ needs 4.5 probes on average, while 12.5 probes are necessary if $\alpha = 0.75$. The performance of deletions substantially degrades as $\alpha \to 1$, just as for all other operations (insertions, successful and unsuccesful searches).

2.2 Improving unsuccessful search performance

Unsuccessful search in linear probing tables with Robin Hood can be improved by observing that the search can be stopped if we probe a location such that the number of probes there is strictly smaller than the number of probes that we have made to arrive from the starting hashing location. Thus we can replace the main search loop

while $t[i]$.probes $\neq 0 \wedge k \neq t[i]$.key **do** ...

by the more efficient

while $t[i]$.probes $\geq p \wedge k \neq t[i]$.key **do** $\ldots; p \leftarrow p \oplus 1$

Let $U_{n,j}^{\text{opt}}$ denote the cost of an unsuccessful search that starts at location j using this optimized search loop, and $tOPTUSPL$ the total optimized unsucessful search path length. We use U_n^{opt} for the cost of an optimized unsuccessful search starting at a random location, and $\overline{U}_n^{\text{opt}} = tOPTUSPL/M$.

The following lemma gives us the fundamental performance of this optimized unsuccessful search.

Lemma 1.

$$tSSPL + M = tOPTUSPL.$$

Proof. Assume we start a probe sequence at location j. Then we stop at location $j \oplus k$ if and only if $t[j \oplus k]$.probes $< k+1$ and $t[j \oplus k']$.probes $\geq k'+1$ for all $1 \leq k' < k$. Now, if we insert an element with hash address j but that is "greater" than any other according to the rule choosen to decide ties in RH, it would be finally be inserted at location $j \oplus k$ (eventually displacing the element that sitted there). Hence $U_{n,j}^{\text{opt}} = S_{n,i} + 1$ with i corresponding to the element at location $j \oplus (k-1)$. Indeed, the ith inserted element must have hash address $\leq j$ for otherwise, the optimized search algorithm have stopped before. Summing over all possible j we get

$$\begin{aligned} tOPTUSPL = \sum_{0 \leq j < M} U_{n,j}^{\text{opt}} &= \sum_{t[j].\text{probes}=0} 1 + \sum_{t[j].\text{probes}\neq 0} U_{n,j}^{\text{opt}} \\ &= M - n + \sum_{1 \leq i \leq n} (S_{n,i} + 1) = M - n + n + tSSPL = M + tSSPL. \end{aligned}$$

From there and well known results on the performance of successful search, it is easy to derive the following theorem.

Theorem 3.

$$\begin{aligned} \mathbb{E}\{U_n^{opt}\} &= 1 + \frac{n}{M}\mathbb{E}\{S_n\} = 1 + \frac{n}{2M}\left(Q_0(n-1,M)+1\right) \\ &= 1 + \frac{n}{2M}\left(1 + \sum_{k\geq 0} \frac{(n-1)^{\underline{k}}}{M^k}\right) \sim 1 + \frac{\alpha}{2}\left(1 + \frac{1}{1-\alpha}\right) + o(1). \end{aligned}$$

If the order of keys is used to resolve ties in RH then the search loop can be further optimized:

while $t[i]$.probes $\geq p \wedge k > t[i]$.key **do** ...

For this new optimized unsucessful search, the performance varies depending on the key we were looking for. Call $tOPTUSPL'$ the cumulated cost for this new search loop, then

$$tSSPL + M - n = tOPTUSPL - n \leq tOPTUSPL' \leq tOPTUSPL = tSSPL + M,$$

the range corresponding to unsucessful search of a key smaller than any other, and to unsuccessful search of a key large than any other. When the order of keys is not taken into account we are effectively doing the same steps as if the sought key were larger than any other and thus attaining the upper bound. For an average case analysis, we should assume that if there are n_j keys that hash to position j then the sought key fails with identical probability in any of the $n_j + 1$ intervals that they define. Thus,

$$\mathbb{E}\{U_{n,j}^{\text{opt'}}\} = \sum_{k=0}^{n_j} \frac{1}{n_j+1}(\mathbb{E}\{U_{n,j}^{\text{opt}}\} - k) = \mathbb{E}\{U_{n,j}^{\text{opt}}\} - \frac{1}{n_j+1}\sum_{k=0}^{n_j} k = \mathbb{E}\{U_{n,j}^{\text{opt}}\} - \frac{n_j}{2},$$

and asymptotically

$$\mathbb{E}\{U_n^{\text{opt'}}\} = \mathbb{E}\{U_n^{\text{opt}}\} - \frac{n}{2M} = 1 - \frac{\alpha}{2} + \alpha\mathbb{E}\{S_n\} = \frac{1}{2}\frac{2-\alpha}{1-\alpha},$$

with $\alpha = n/M$, as usual.

3 Deletions in other open addressing hash tables

In this section we develop a deletion algorithm that applies to many other open addressing schemes, in particular, to any scheme where the jth probe location is given by

$$i_j = i_0 \oplus f(j), \quad j > 0,$$

for some function f depending only on j and such that the equation $(i + f(j)) \bmod M = (i' + f(j)) \bmod M$, with $i, i' \in [0..M-1]$ and $0 < j < M$, does only admit the solution $i = i'$. This is the case, for instance, of quadratic hashing [11] with carefully choosen values for the mulipliers and/or the table size. Linear probing falls in this category too, but the deletion algorithm studied in Section 2 must be used in that case. Our algorithm does not apply to open addressing schemes like double hashing [8], where a second hash function is applied to the item to compute the increments, namely $i_j = i_0 \oplus f(j,x)$. In the sequel, we will use $\Delta(j) := f(j) - f(j-1)$, so that $i_j = i_{j-1} \oplus \Delta(j)$.

In our deletion algorithm we assume that each entry in the table has two additional attributes: $t[i]$.probes, indicates the number of probes made from the initial hash location of the item at $t[i]$ until the current location i; $t[i]$.next is a bit vector, such that its jth bit is 1 if and only if i is the $(j+1)$th probe location for some item in the table (other than the item at $t[i]$).

If a FCFS scheme is used for insertions, then we can say that $t[i]$.next$[j] = 1$ if and only if i is the $(j+1)$th probe location for some item in the table which was inserted after the item at $t[i]$. Although the deletion algorithm in this section can be easily adapted to other reordering schemes, we will describe the algorithm for the case of FCFS insertions, since it makes that description easier.

To begin with, the number of bits in the next bit vectors imposes a restriction in the hash table. If that length is w, say $w = 32$, then special measures must be adopted if some probe sequence exceeded that length. We assume that in that case all the elements would be rehashed into a larger table, just as we would do if the load factor $\alpha = n/M$ had reached some threshold. Since the longest probe sequence has expected length $O(\log n)$ in many open addressing schemes, we don't need a large number of bits w unless we need to store an extremely large number of items.

Now, suppose we want to delete item x. First, we use the standard search algorithm to locate x, probing locations $i_0 = h(x)$, i_1, ..., i_{p-1}. Assume that x is present at location $i = i_{p-1}$, so we have found it after p probes. We must have thus $t[i]$.probes $= p$. If $t[i]$.next$[j] = 0$ for all $0 \le j < w$ then no element follows x in the hash table, and we can safely remove it—setting $t[i]$.probes $= 0$ indicates that the slot is empty. Furthermore, the bit vectors next at locations ni_{p-1}, i_{p-2}, ..., i_0 must be updated to reflect the deletion of $t[i]$. This task is accomplished by the procedure FIXBITSBACKWARDS, which we describe later.

If $t[i]$.next$[j] = 1$ for at least some j then we cannot simply mark the slot as empty and update the bit vectors of the preceding probe locations. But the bit vector $t[i]$.next tells us exactly which probe sequences have passed through $t[i]$. Namely, if $t[i]$.next$[p-1] = 1$ then there is some element that has probed the same locations as x and we must "compress" the sequence at locations $i_{p-1}, i_p, i_{p+1}, \ldots$ much in the spirit of the compression that we did in linear hashing. The compression is carried out by the procedure COMPRESS(i). The compression of the sequence ends at the first slot $t[i_q]$ in the sequence such that $t[i_q]$.next$[q] = 0$. Each time that some $t[i_k]$ satisfies $t[i_k]$.probes $= k+1$ we conclude that such element can be swapped with the emptied slot, the argument being similar to that underlying the linear probing deletions. Because of the invariant of our representation, the slot at i_k must contain a synonym of x, because $t[i_k]$.probes $= k+1$. As we have already done when emptying the slot with x, once the compression of the sequence ending at i_q is finished, we must traverse the sequence backwards to update the bit vectors 'next'. More specifically, every element preceding i_q in the probe sequence but that has not been moved from its location and that comes after the last reallocated item by COMPRESS(i) loses the item at i_q as a successor, and that must be reflected by its 'next' bit vector. Again, it is FIXBITSBACKWARDS the procedure that takes care.

By definition, $t[i_q]$.next$[q] = 0$; however, some other bit $t[i_q]$.next might be 1, and thus the corresponding sequence must be compressed, now starting from i_q. By the way, the procedure COMPRESS returns the location where the compression finished, i_q in our discussion. In the case, that there were more than one probe sequence passing through i_q we must choose some. We propose to choose the sequence with the largest number of probes when reaching i_q, in other words, the largest j such that $t[i_q]$.next$[j] = 1$. The idea is here trying to compress the probe sequence for which the number of probes to reach i_q is largest. The choice of a sequence to compress, when there is more than one choice, shows that Algorithm 2 does not preserve randomness, in the strong sense, unlike Algorithm 1. After deletion of x, the table might not be the same as if x were never inserted. To achieve that property we should "record" information not about probe sequences passing through every location, but the "time" as well. For instance in an FCFS table we should

compress the oldest probe sequence passing through the location. But for a RH the choice is exactly the one of our algorithm: the probe sequence with the largest displacement must be compressed. Despite we have not completed a proof that our deletion algorithm has this strong randomness preservation property, we conjecture that this is the case. On the other hand, it can be true that there is some weak randomness preservation in all variants, despite the table obtained after a deletion is not the same as if the item wasn't inserted at all. By weak randomness preservation we mean that the probability of obtaining a certain hash table with random insertions alone is the same as the probability of obtaining it with random insertions and random deletions, in any arbitrary order. Establishing such strong or weak randomness preservation is crucial for the subsequent analysis.

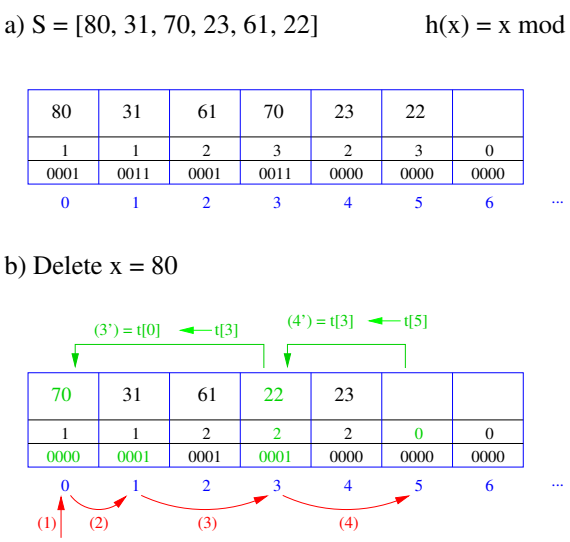

Fig. 1. Example of an execution of the deletion algorithm for quadratic hashing.

Fig. 1 shows an example of the execution of Algorithm 2. We use the probe sequence i_0, i_0+1, i_0+3, i_0+6, ...; in general, $i_j = i_0 \oplus j(j+1)/2$, that is, $\Delta(j) = j$. The first part (a) shows the result of inserting the items in S, in the indicated order, with the hash function $h(x) = x \bmod 10$ (which is obviously a bad choice in practice). The bottom part of each slot shows the number of probes needed to locate the corresponding item, as well as the bit vector '`next`', for $w = 4$. The second part (b) of the figure shows the result of removing item $x = 80$ from the previous table. The main loop of the call COMPRESS(0) makes 2 probes, and moves item 70 to slot $t[0]$. The procedure FIXBITSBACKWARDS updates $t[1]$.`next`, containing item 31, as there is no element probing $i = 1$ during its second probe. On exit from the call COMPRESS(0), the main loop of the deletion will make the call COMPRESS(3) to compress the probe sequence of the elements that passed through location $i = 3$ on their second probe. Such probe sequence contains one element only, 22, which moves to slot $t[3]$. Thus the second call to COMPRESS makes a single probe. There are 4 probes in total, three probes needed by the two calls to COMPRESS plus an additional probe to locate the element that we wish to remove.

As a by-product of storing the 'next' bits, we can optimize the searching procedure; if we reach a location i after p probes and $t[i].next[p-1] = 0$ then either $x = t[i]$ or x is not in the table. Note that $t[i].next[p-1] = 0$ does not imply that the next location in the probe sequence is empty: it might be occupied. But neither that position nor those further away in the probe sequence can contain x since no element with the same porbe sequence as x was inserted past location i. So, the ordinary loop that looks for x or an empty slot is necessary for insertions, but an optimized search loop can be used for both search and deletions. This type of optimization, but using only a single bit per slot to flag whether there was any item that probed that location and had to be inserted further away, has been proposed by Furukawa (as cited by Gunji and Goto [7]) and by Amble and Knuth [1] in the context of ordered hashing. Lyon [10] proposes to keep the length of the longest successful search path length; any search can be stopped as soon as that number of probes has been made. The ordinary loop is necessary for insertions, but the optimization described above

can be used both for search and for deletions (to locate the item to be removed). It is interesting to note that the cost of unsuccesful searches is then no longer related to that of insertions (and vice-versa), contrary to what happens if we use the ordinary loop for search.

Algorithm 2 Removal in a open addressing hash table.

```
procedure REMOVE(k)
        ▷ k: key of the item to remove; t: the hash table
        ▷ M: size of the hash table; n: number of items, n < M
    i ← LOOKUP(k)
        ▷ t[i].key = k ∨ t[i].probes = 0
    if t[i].probes ≠ 0 then
        while t[i].probes ≠ 0 do
            i ← COMPRESS(i)        ▷ compress the chain from i onwards
            t[i].probes ← 1 + the largest j such that t[i].next[j] = 1 or -1 if not such j exists
        n ← n − 1
procedure COMPRESS(i)
    p ← t[i].probes
    free ← i; freep ← p
    while t[i].next[p − 1] ≠ 0 do
        i ← i ⊕ Δ(p); p ← p + 1;
            ▷ Check if the current item at i can be moved to the free slot
        if t[i].probes = p then
            t[free].key ← t[i].key
            t[free].probes ← freep
            free ← i; freep ← p
    FIXBITSBACKWARDS(i, p)
    return i
procedure FIXBITSBACKWARDS(i, p)
    p ← p − 1; i ← i ⊖ Δ(p);
    while p > 0 ∧ t[i].probes ≠ p do
        t[i].next[p − 1] ← 0
        p ← p − 1; i ← i ⊖ Δ(p);
    if p > 0 then
        t[i].next[p − 1] ← 0
```

4 Conclusions

Let us put it blunty: there is a good deletion algorithm for linear probing hash tables. It has been around for a long time. It's simple. It's elegant. It's practical. It leaves the table just as if it the removed item was never there. We should stop teaching CS undergraduates that the only way to delete in an open addressing hash table is using lazy deletions. We may teach them lazy deletions as a general solution that works for any open addressing hash table. But we should avoid the message that lazy deletions are the only way to perform deletions that we have—that's only true for some open addressing schemes. In particular, lazy deletions shouldn't be the method of choice for linear probing hash.

Linear probing hashing is virtually explained in every textbook as an example of open addressing hashing; often other schemes are only described in passing and very briefly, or not explained at all. Algorithm 1 (in a way or another) should be always explained, not only because of its simplicity and good properties; it also gives the student a deeper understanding of the inner workings of linear probing hashing, and it opens the door for introducing interesting variations such as Robin Hood hashing.

Understanding Algorithms 1 and 2 also helps appreciate the difficulty of deletions in general open addressing hash tables. The case for lazy deletion is very strong when considering open addressing hashing schemes other than linear probing, as multiple, nonoverlapping probe sequences pass along any given position of the table. However, we have shown here that true deletions are possible, neither too complex nor totally unpractical, for some more general open addressing hash tables, including the important particular case of quadratic hash tables. A throughly experimental study

of quadratic hash tables with our deletion algorithm, as compared to other simple and practical hashing schemes (e.g., separate chaining, linear probing, cuckoo hashing) would be very useful to have a proper perspective of the usefulness, in practical terms, of the deletion algorithm that we propose. Moreover, a detailed analysis of the probabilistic behavior of our deletion algorithm would be desirable and interesting, and it would undoubtedly imply a deep understanding of the structure of clusters in open addressing hash schemes with secondary clustering. As we have already mentioned, the analysis of insertions and searches in such tables is often approximated with the idealized model of random hashing, and the analysis of the deletion algorithm will likely need such simplifying assumptions, as well. On an intuitive basis, once the element to be removed is located, the typical scenario for moderate values of the load factor should be that we need only a small number of calls to the "compression" routine, each dealing with a short probe sequence. If that's the case then the overall performance of the deletion algorithm would be quite reasonable on practical grounds. In any case, whether our deletion algorithm is more or less competitive in practice, we feel it is of independent theoretical interest and it can also help us to understand better how open addressing hash tables work.

References

1. O. Amble and D.E. Knuth. Ordered hash tables. *The Computer Journal*, 17(2):135–142, 1974.
2. P. Celis. *Robin Hood Hashing*. PhD thesis, Computer Science Department, University of Waterloo, 1986. Technical Report CS-86-14.
3. P. Celis and J. Franco. The analysis of hashing with lazy deletions. *Information Sciences: an International Journal*, 62(1–2):13–26, 1992.
4. P. Celis, P.-A. Larson, and J. I. Munro. Robin Hood hashing. In *Proc. of the 26$^{\text{th}}$ Annual Symposium on the Foundations of Computer Science (FOCS)*, pages 281–288, 1985.
5. Ph. Flajolet, P. V. Poblete, and A. Viola. On the analysis of linear probing hashing. *Algorithmica*, 22(4):490–515, 1998.
6. L. J. Guibas. The analysis of hashing techniques that exhibit *k*-ary clustering. *J. Assoc. Comput. Mach.*, 25(4):544–555, 1978.
7. T. Gunji and E. Goto. Studies on hashing part-1: A comparison of hashing algorithms with key deletion. *J. Information Processing*, 3(1):1–12, 1980.
8. D.E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, Reading, Mass., 2$^{\text{nd}}$ edition, 1998.
9. M. Kolosovskiy. Simple implementation of deletion from open-address hash table. Preprint arXiv:0909.2547v1, September 2009. Available from http://arxiv.org/abs/0909.2547.
10. G. Lyon. Packed scatter tables. *Comm. ACM*, 21(10):857–865, 1978.
11. W.D. Maurer. An improved hash code for scatter storage. *Comm. ACM*, 11(1):35–38, 1968.
12. K. Mehlhorn and P. Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer-Verlag, 2008.
13. J.I. Munro and P. Celis. Techniques for collision resolution in hash tables with open addressing. In *Proc. of the 1986 ACM Fall Joint Computer Conference*, pages 601–610. IEEE Computer Society Press, 1986.
14. P. Poblete and J.I. Munro. Last-come-first-served hashing. *J. Algorithms*, 10:228–248, 1989.
15. P. V. Poblete and A. Viola. The effect of deletions on different insertion disciplines for hash tables (extended abstract). *Electronic Notes in Discrete Mathematics*, 7:146–149, 2001.
16. R. Sedgewick. *Algorithms in C*, volume 1. Addison-Wesley, 3$^{\text{rd}}$ edition, 1997.