Article

Two-Way Linear Probing Revisited

Ketan Dalal, Luc Devroye and Ebrahim Malalla *,†

School of Computer Science, McGill University, Montreal, QC H3A 2K6, Canada; lucdevroye@gmail.com (L.D.)
* Correspondence: emalalla@ahlia.edu.bh
† Current address: Department of Mathematical Sciences, Ahlia University, Manama P.O. Box 10878, Bahrain.

Abstract: Linear probing continues to be one of the best practical hashing algorithms due to its good average performance, efficiency, and simplicity of implementation. However, the worst-case performance of linear probing seems to degrade with high load factors due to a primary-clustering tendency of one collision to cause more nearby collisions. It is known that the maximum cluster size produced by linear probing, and hence the length of the longest probe sequence needed to insert or search for a key in a hash table of size n, is $\Omega(\log n)$, in probability. In this article, we introduce linear probing hashing schemes that employ two linear probe sequences to find empty cells for the keys. Our results show that two-way linear probing is a promising alternative to linear probing for hash tables. We show that two-way linear probing has an asymptotically almost surely $O(\log \log n)$ maximum cluster size when the load factor is constant. Matching lower bounds on the maximum cluster size produced by any two-way linear probing algorithm are obtained as well. Our analysis is based on a novel approach that uses the multiple-choice paradigm and witness trees.

Keywords: open addressing hashing; linear probing; parking problem; worst-case search time; two-way chaining; multiple-choice paradigm; randomized algorithms; witness tree; probabilistic analysis

1. Introduction

In classical open addressing hashing [1], m keys are hashed sequentially and on-line into a table of size $n > m$, (that is, a one-dimensional array with n cells which we denote by the set $\mathcal{T} = \{0, \dots, n-1\}$), where each cell can harbor at most one key. Each key x has only one infinite probe sequence $f_i(x) \in \mathcal{T}$, for $i \in \mathbb{N}$. During the insertion process, if a key is mapped to a cell that is already occupied by another key, a collision occurs, and another probe is required. The probing continues until an empty cell is reached where a key is placed. This method of hashing is pointer-free, unlike hashing with separate chaining where keys colliding in the same cell are hashed to a separate linked list or chain. For a discussion of different hashing schemes, see [2–4].

In classical linear probing, the probe sequence for each key x is defined by $f_{i+1}(x) = f_i(x) + 1 \mod n$, for $i \in [\![n]\!] := \{1, \dots, n\}$. Linear probing is known for its good practical performance, efficiency, and simplicity. It continues to be one of the best hash tables in practice due to its simplicity of implementation, absence of overhead for internally used pointers, cache efficiency, and locality of reference [5–8]. On the other hand, the performance of linear probing seems to degrade with high load factors m/n, due to a primary-clustering tendency of one collision to cause more nearby collisions. In particular, the length of the longest probe sequence needed to insert (or search for) a key in a hash table, with constant load factor, constructed by linear probing is proven [9] to be $\Omega(\log n)$, in probability.

The purpose of this paper is to design efficient open addressing hashing schemes that improve the worst-case performance of classical linear probing. Our study concentrates on schemes that use two linear probe sequences to find possible hashing cells for the keys. Each key chooses two initial cells independently and uniformly at random, with

replacement. From each initial cell, we probe linearly, and then cyclically whenever the last cell in the table is reached, to find two empty cells which we call terminal cells. The key then is inserted into one of these terminal cells according to a fixed strategy. We consider strategies that utilize the *greedy multiple-choice paradigm* [10,11]. We show that some of the trivial insertion strategies with two-way linear probing have unexpected poor performance. For example, one of the trivial strategies we study inserts each key into the terminal cell found by the shorter probe sequence. Another simple strategy inserts each key into the terminal cell that is adjacent to the smaller cluster, where a cluster is an isolated set of consecutively occupied cells. Unfortunately, the performances of these two strategies are not ideal. We prove that, when any of these two strategies is used to construct a hash table with constant load factor, the maximum unsuccessful search time is $\Omega(\log n)$, with high probability (w.h.p.). Indeed, we prove that, w.h.p., a giant cluster of size $\Omega(\log n)$ emerges in a hash table of constant load factor, if it is constructed by a two-way linear probing insertion strategy that always inserts any key upon arrival into the empty cell of its two initial cells whenever one of them is empty.

Consequently, we introduce two other strategies that overcome this problem. First, we partition the hash table into equal-sized blocks of size β, assuming n/β is an integer. We consider the following strategies for inserting the keys:

A.   Each key is inserted into the terminal cell that belongs to the least crowded block, i.e., the block with the least number of keys.
B.   For each block i, we define its weight to be the number of keys inserted into terminal cells found by linear probe sequences whose starting locations belong to block i. Each key, then, is inserted into the terminal cell found by the linear probe sequence that has started from the block of smaller weight.

For strategy B, we show that β can be chosen such that, for any constant load factor $\alpha := m/n$, the maximum unsuccessful search time is not more than $c \log_2 \log n$, w.h.p., where c is a function of α. If $\alpha < 1/2$, the same property also holds for strategy A. Furthermore, these schemes are optimal up to a constant factor in the sense that an $\Omega(\log \log n)$ universal lower bound holds for any strategy that uses two linear probe sequences, even if the initial cells are chosen according to arbitrary probability distributions.

Paper Scope

In the next section, we provide a summary of the related background and history. In Section 3, we present our proposed two-way linear probing hashing algorithms. We prove in Section 4 a universal lower bound of order of $\log \log n$ on the maximum unsuccessful search time of any two-way linear probing algorithm. We prove, in addition, that not every two-way linear probing scheme behaves efficiently. We devote Section 5 to the positive results, where we demonstrate that some of our two-way linear probing heuristics accomplish $O(\log \log n)$ worst-case unsuccessful search time. The simulation and the comparison results of the studied algorithms are summarized in Section 6.

2. Background and History

For hashing with separate chaining, one can achieve $O(\log \log n)$ maximum search time by applying the *two-way chaining* scheme [10,12,13], where each key is inserted into the shorter chain between two chains chosen independently and uniformly at random, with replacement, breaking ties randomly. It is proven [10,14,15] that, when $r = \Omega(n)$ keys are inserted into a hash table with n chains, the length of the longest chain upon termination is $\log_2 \log n + r/n \pm O(1)$, w.h.p. Of course, this idea can be generalized to open addressing. Assuming the hash table is partitioned into blocks of size β, we allow each key to choose two initial cells, and hence two blocks, independently and uniformly at random, with replacement. From each initial cell and *within its block*, we probe linearly and cyclically, if necessary, to find two empty cells; that is, whenever we reach the last cell in the block and it is occupied, we continue probing from the first cell in the same block. The key, then, is inserted into the empty cell that belongs to the least full block. Using the two-way chaining

result, one can show that, for suitably chosen β, the maximum unsuccessful search time is $O(\log \log n)$, w.h.p. However, this scheme uses probe sequences that are not totally linear; rather, they are locally linear within the blocks.

2.1. Probing and Replacement

Open addressing schemes are determined by the type of the probe sequence, and the replacement strategy for resolving the collisions. Some of the commonly used probe sequences are as follows:

1. **Random Probing [16]:** For every key x, the infinite sequence $f_i(x)$ is assumed to be independent and uniformly distributed over $\mathcal{T}$. That is, we require to have an infinite sequence f_i of truly uniform and independent hash functions. If for each key x, the first n probes of the sequence $f_i(x)$ are distinct, i.e., it is a random permutation, then it is called **uniform probing** [1].
2. **Linear Probing [1]:** For every key x, the first probe $f_1(x)$ is assumed to be uniform on $\mathcal{T}$, and the next probes are defined by $f_{i+1}(x) = f_i(x) + 1 \mod n$, for $i \in [\![n]\!]$. So we only require f_1 to be a truly uniform hash function.
3. **Double Probing [17]:** For every key x, the first probe is $f_1(x)$, and the next probes are defined by $f_{i+1}(x) = f_i(x) + g(x) \mod n$, for $i \in \mathbb{N}$, where f_1 and g are truly uniform and independent hash functions.

Random and uniform probings are, in some sense, the idealized models [18,19], and their plausible performances are among the easiest to analyze, but obviously they are unrealistic. Linear probing is perhaps the simplest to implement, but it behaves badly when the table is almost full. Double probing can be seen as a compromise.

During the insertion process of a key x, suppose that we arrive at the cell $f_i(x)$, which is already occupied by another previously inserted key y, that is, $f_i(x) = f_j(y)$, for some $j \in \mathbb{N}$. Then a replacement strategy for resolving the collision is needed. Three strategies have been suggested in the literature (see [20] for other methods):

1. FIRST COME FIRST SERVED (FCFS) [1]: The key y is kept in its cell, and the key x is referred to the next cell $f_{i+1}(x)$.
2. LAST COME FIRST SERVED (LCFS) [21]: The key x is inserted into the cell $f_i(x)$, and the key y is pushed along to the next cell in its probe sequence, $f_{j+1}(y)$.
3. ROBIN HOOD [22,23]: The key which traveled the farthest is inserted into the cell. That is, if $i > j$, then the key x is inserted into the cell $f_i(x)$, and the key y is pushed along to the next cell $f_{j+1}(y)$; otherwise, y is kept in its cell, and the key x tries its next cell $f_{i+1}(x)$.

2.2. Average Performance

Evidently, the performance of any open addressing scheme deteriorates when the ratio m/n approaches 1, as the cluster sizes increase, where a cluster is an isolated set of consecutively occupied cells (cyclically defined) that are bounded by empty cells. Therefore, we shall assume that the hash table is α-full, that is, the number of hashed keys $m = \lfloor \alpha n \rfloor$, where $\alpha \in (0,1)$ is a constant called the load factor. The asymptotic average-case performance has been extensively analyzed for random and uniform probing [1,16,18,19,24,25], linear probing [3,26–28], and double probing [17,29–32]. The expected search times were proven to be constants, more or less, depending on α only. Recent results about the average-case performance of linear probing and the limit distribution of the construction time have appeared in [33–36]. See also [37–39] for the average-case analysis of linear probing for non-uniform hash functions.

It is worth noting that the average search time of linear probing is independent of the replacement strategy; see [1,3]. This is because the insertion of any order of the keys results in the same set of occupied cells, i.e., the cluster sizes are the same; hence, the total displacement of the keys—from their initial hashing locations—remains unchanged. It is not difficult to see that this independence is also true for random and double probings. That

is, the replacement strategy does not have any effect on the average successful search time in any of the above probings. In addition, since, in linear probing, the unsuccessful search time is related to the cluster sizes (unlike random and double probings), the expected and the maximum unsuccessful search times in linear probing are invariant to the replacement strategy.

It is known that the LCFS [21,40] and ROBIN HOOD [20,22,23,35] strategies minimize the variance of displacement. Recently, Janson [41] and Viola [42] have reaffirmed the effect of these replacement strategies on the individual search times in linear probing hashing.

2.3. Worst-Case Performance

The focal point of this article, however, is the worst-case search time which is proportional to the length of the longest probe sequence over all keys (LLPS, for short). Many results have been established regarding the worst-case performance of open addressing.

The worst-case performance of linear probing with the FCFS policy was analyzed by Pittel [9]. He proved that the maximum cluster size, and hence the LLPS needed to insert (or search for) a key, is asymptotic to $(\alpha - 1 - \log \alpha)^{-1} \log n$, in probability. As we mentioned above, this bound holds for linear probing with *any* replacement strategy. Chassaing and Louchard [43] studied the threshold of emergence of a giant cluster in linear probing. They showed that, when the number of keys $m = n - \omega(\sqrt{n})$, the size of the largest cluster is $o(n)$, w.h.p.; however, when $m = n - o(\sqrt{n})$, a giant cluster of size $\Theta(n)$ emerges, w.h.p.

Gonnet [44] proved that, with uniform probing and the FCFS replacement strategy, the expected LLPS is asymptotic to $\log_{1/\alpha} n - \log_{1/\alpha} \log_{1/\alpha} n + O(1)$ for α-full tables. However, Poblete and Munro [21,40] showed that, if random probing is combined with the LCFS policy, then the expected LLPS is at most $(1 + o(1))\Gamma^{-1}(\alpha n) = O(\log n / \log \log n)$, where Γ is the gamma function.

On the other hand, the ROBIN HOOD strategy with random probing leads to a more striking performance. Celis [22] first proved that the expected LLPS is $O(\log n)$. However, Devroye, Morin, and Viola [45] tightened the bounds and revealed that the LLPS is indeed $\log_2 \log n \pm \Theta(1)$, w.h.p., thus achieving a double logarithmic worst-case insertion and search times for the first time in open addressing hashing. Unfortunately, one cannot ignore the assumption in random probing about the availability of an infinite collection of hash functions that are sufficiently independent and behave like truly uniform hash functions in practice. On the other side of the spectrum, we already know that the ROBIN HOOD policy does not affect the maximum unsuccessful search time in linear probing. However, ROBIN HOOD may be promising with double probing.

2.4. Other Initiatives

Open addressing methods that rely on the rearrangement of keys were under investigation for many years, see, e.g., [20,46–50]. Pagh and Rodler [51] studied a scheme called cuckoo hashing that exploits the LCFS replacement policy. It uses two hash tables of size $n > (1 + \epsilon)m$, for some constant $\epsilon > 0$, and two independent hash functions chosen from an $O(\log n)$-universal class—one function only for each table. Each key is hashed initially by the first function to a cell in the first table. If the cell is full, then the new key is inserted there anyway, and the old key is kicked out to the second table to be hashed by the second function. The same rule is applied in the second table. Keys are moved back and forth until a key moves to an empty location or a limit has been reached. If the limit is reached, new independent hash functions are chosen, and the tables are rehashed. The worst-case search time is at most two, and the amortized expected insertion time, nonetheless, is constant. However, this scheme utilizes less than 50% of the allocated memory, has a worst-case insertion time of $O(\log n)$, w.h.p., and depends on a wealthy source of provably good independent hash functions for the rehashing process. For further details see [52–55].

The space efficiency of cuckoo hashing is significantly improved when the hash table is divided into blocks of fixed size $b \geq 1$ and more hash functions are used to choose $k \geq 2$ blocks for each key where each is inserted into a cell in one of its chosen blocks using

the cuckoo random walk insertion method [56–61]. For example, it is known [57,58] that 89.7% space utilization can be achieved when $k = 2$ and the hash table is partitioned into non-overlapping blocks of size $b = 2$. On the other hand, when the blocks are allowed to overlap, the space utilization improves to 96.5% [57,61]. The worst-case insertion time of this generalized cuckoo hashing scheme, however, is proven [59,62] to be polylogarithmic, w.h.p.

Many real-time static and dynamic perfect hashing schemes achieving constant worst-case search time, and linear (in the table size) construction time and space were designed in [63–70]. All of these schemes, which are based, more or less, on the idea of multilevel hashing, employ more than a constant number of perfect hash functions chosen from an efficient universal class. Some of them even use $O(n)$ functions.

2.5. The Multiple-Choice Paradigm

Allocating balls into bins is one of the historical assignment problems [71,72]. We are given r balls that have to be placed into s bins. The balls have to be inserted sequentially and on-line, that is, each ball is assigned upon arrival without knowing anything about the future coming balls. The *load of a bin* is defined to be the number of balls it contains. We would like to design an allocation process that minimizes the maximum load among all bins upon termination. For example, in a classical allocation process, each ball is placed into a bin chosen independently and uniformly at random, with replacement. It is known [44,73–75] that, if $r = \Theta(s)$, the maximum load upon termination is asymptotic to $\log s / \log \log s$, in probability.

On the other hand, the *greedy multiple-choice allocation process*, which appeared in [76,77] and studied by Azar et al. [10], inserts each ball into the least loaded bin among $d \geq 2$ bins chosen independently and uniformly at random, with replacement, breaking ties randomly. Throughout, we will refer to this process as GREEDYMC(s, r, d) for inserting r balls into s bins. Surprisingly, the maximum bin load of GREEDYMC(s, s, d) decreases exponentially to $\log_d \log s \pm O(1)$, w.h.p., [10]. However, one can easily generalize this to the case $r = \Theta(s)$. It is also known that the greedy strategy is stochastically optimal in the following sense.

Theorem 1 (Azar et al. [10]). *Let $s, r, d \in \mathbb{N}$, where $d \geq 2$, and $r = \Theta(s)$. Upon termination of* GREEDYMC(s, r, d)*, the maximum bin load is $\log_d \log s \pm O(1)$, w.h.p. Furthermore, the maximum bin load of any on-line allocation process that inserts r balls sequentially into s bins where each ball is inserted into a bin among d bins chosen independently and uniformly at random, with replacement, is at least $\log_d \log s - O(1)$, w.h.p.*

Berenbrink et al. [14] extended Theorem 1 to the heavily loaded case where $r \gg s$, and recorded the following tight result.

Theorem 2 (Berenbrink et al. [14]). *There is a constant $C > 0$ such that, for any integers $r \geq s > 0$, and $d \geq 2$, the maximum bin load upon termination of the process* GREEDYMC(s, r, d) *is $\log_d \log s + r/s \pm C$, w.h.p.*

Theorem 2 is a crucial result that we have used to derive our results; see Theorems 8 and 9. It states that the deviation from the average bin load, which is $\log_d \log s$, stays unchanged as the number of balls increases.

Vöcking [11,78] demonstrated that it is possible to improve the performance of the greedy process, if non-uniform distributions on the bins and a tie-breaking rule are carefully chosen. He suggested the following variant, which is called *Always-Go-Left*. The bins are numbered from 1 to n. We partition the s bins into d groups of almost equal size, that is, each group has size $\Theta(s/d)$. We allow each ball to select upon arrival d bins independently at random, but the i-th bin must be chosen uniformly from the i-th group. Each ball is placed on-line, as before, in the least full bin, but upon a tie, the ball is always placed in the leftmost bin among the d bins. We shall write LEFTMC(s, r, d) to refer to this process. Vöcking [11] showed that, if $r = \Theta(s)$, the maximum load of LEFTMC(s, r, d) is $\log \log s / (d \log \phi_d) + O(1)$, w.h.p., where ϕ_d is a constant related to a generalized Fibonacci

sequence. For example, the constant $\phi_2 = 1.61...$ corresponds to the well-known golden ratio, and $\phi_3 = 1.83$. In general, $\phi_2 < \phi_3 < \phi_4 < \cdots < 2$, and $\lim_{d\to\infty} \phi_d = 2$. Observe the improvement on the performance of GREEDYMC(s, r, d), even for $d = 2$. The maximum load of LEFTMC$(s, r, 2)$ is $0.72... \times \log_2 \log s + O(1)$, whereas in GREEDYMC$(s, r, 2)$, it is $\log_2 \log s + O(1)$. The process LEFTMC(s, r, d) is also optimal in the following sense.

Theorem 3 (Vöcking [11]). *Let $r, s, d \in \mathbb{N}$, where $d \geq 2$, and $r = \Theta(s)$. The maximum bin load of of LEFTMC(s, r, d) upon termination is $\log \log s / (d \log \phi_d) \pm O(1)$, w.h.p. Moreover, the maximum bin load of any on-line allocation process that inserts r balls sequentially into s bins where each ball is placed into a bin among d bins chosen according to arbitrary, not necessarily independent, probability distributions defined on the bins is at least $\log \log s / (d \log \phi_d) - O(1)$, w.h.p.*

Berenbrink et al. [14] studied the heavily loaded case and recorded the following theorem.

Theorem 4 (Berenbrink et al. [14]). *There is a constant $A > 0$ such that, for any integers $r \geq s > 0$, and $d \geq 2$, the maximum bin load upon termination of the process LEFTMC(s, r, d) is $\log \log s / (d \log \phi_d) + r/s \pm A$, w.h.p.*

For other variants and generalizations of the multiple-choice paradigm see [79–84]. The paradigm has been used to derive many applications, e.g., in load balancing, circuit routing, IP address lookups, and computer graphics [75,85–88].

3. The Proposal

We design linear probing algorithms that accomplish double logarithmic worst-case search time. Inspired by the two-way chaining algorithm [10] and its powerful performance, we promote the concept of open addressing hashing with two-way linear probing. The essence of the proposed concept is based on the idea of allowing each key to generate two independent linear probe sequences and making the algorithm decide, according to some strategy, at the end of which sequence the key should be inserted. Formally, each input key x chooses two cells independently and uniformly at random, with replacement. We call these cells the *initial hashing cells* available for x. From each initial hashing cell, we start a linear probe sequence (with FCFS policy) to find an empty cell where we stop. Thus, we end up with two unoccupied cells. We call these cells the *terminal hashing cells*. The question now is: into which terminal cell should we insert the key x?

The insertion process of a two-way linear probing algorithm could follow one of the strategies we mentioned earlier: it may insert the key at the end of the shorter probe sequence, or into the terminal cell that is adjacent to the smaller cluster. Others may make an insertion decision even before linear probing starts. In any of these algorithms, the searching process for any key is basically the same: just start probing in both sequences alternately, until the key is found or the two empty cells at the end of the sequences are reached in the case of an unsuccessful search. Thus, the maximum unsuccessful search time is at most twice the size of the largest cluster plus two.

We study the two-way linear probing algorithms stated above, and show that the hash table, asymptotically almost surely, contains a giant cluster of size $\Omega(\log n)$. Indeed, we prove that a cluster of size $\Omega(\log n)$ emerges, asymptotically almost surely, in any hash table of constant load factor that is constructed by a two-way linear probing algorithm that inserts any key upon arrival into the empty cell of its two initial cells whenever one of them is empty.

We introduce two other two-way linear probing heuristics that lead to $\Theta(\log \log n)$ maximum unsuccessful search times. The common idea of these heuristics is the marriage between the two-way linear probing concept and a technique we call *blocking* where the hash table is partitioned into equal-sized blocks. These blocks are used by the algorithm to obtain some information about the keys allocation. The information is used to make better decisions about where the keys should be inserted, and hence, lead to a more even distribution of the keys.

Two-way linear probing hashing has several advantages over the other proposed hashing methods mentioned above: it reduces the worst-case behavior of hashing, it requires only two hash functions, it is easy to parallelize, it is pointer-free and easy to implement, and unlike the hashing schemes proposed in [51,58], it does not require any rearrangement of keys or rehashing. Its maximum cluster size is $O(\log \log n)$, and its average-case performance can be at most twice the classical linear probing as shown in the simulation results. Furthermore, it is not necessary to employ perfectly random hash functions as it is known [6–8] that hash functions with a smaller degree of universality will be sufficient to implement linear probing schemes. See also [31,32,51,53,70,76,89] for other suggestions on practical hash functions.

Throughout, we assume the following. We are given m keys—from a universe set of keys $\mathcal{U}$—to be hashed into a hash table of size n such that each cell contains at most one key. The process of hashing is sequential and on-line, that is, we never know anything about the future keys. The constant $\alpha \in (0,1)$ is preserved in this article for the load factor of the hash table, that is, we assume that $m = \lfloor \alpha n \rfloor$. The n cells of the hash table are numbered $0, \ldots, n-1$. The linear probe sequences always move cyclically from left to right of the hash table. The replacement strategy of all of the introduced algorithms is FCFS. The *insertion time* is defined to be the number of probes the algorithm performs to insert a key. Similarly, the *search time* is defined to be the number of probes needed to find a key, or two empty cells in the case of unsuccessful search. Observe that, unlike classical linear probing, the insertion time of two-way linear probing may not be equal to the successful search time. However, they are both bounded by the unsuccessful search time. Notice also that we ignore the time to compute the hash functions.

3.1. Two-Way Linear Probing

To avoid any ambiguity, we consider this definition.

Definition 1. *A two-way linear probing algorithm is an open addressing hashing algorithm that inserts keys into cells using a certain strategy and does the following upon the arrival of each key:*

1.　*It chooses two initial hashing cells independently and uniformly at random, with replacement.*
2.　*Two terminal (empty) cells are then found by linear probe sequences starting from the initial cells.*
3.　*The key is inserted into one of these terminal cells.*

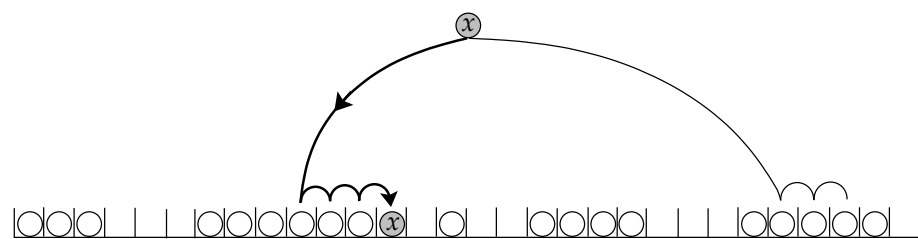To be clear, we give two examples of inefficient two-way linear probing algorithms.

Figure 1. An illustration of algorithm SHORTSEQ(n, m) in terms of balls (keys) and bins (cells). Each ball is inserted into the empty bin found by the shorter sequence.

The Shorter Probe Sequence: SHORTSEQ Algorithm

Our first algorithm places each key into the terminal cell discovered by the shorter probe sequence. More precisely, once the key chooses its initial hashing cells, we start two linear probe sequences. We proceed, sequentially and alternately, one probe from each sequence until we find an empty (terminal) cell where we insert the key. Formally, let $f, g : \mathcal{U} \to \{0, \ldots, n-1\}$ be independent and truly uniform hash functions. For $x \in \mathcal{U}$, define the linear sequence $f_1(x) = f(x)$, and $f_{i+1}(x) = f_i(x) + 1 \mod n$, for $i \in [\![n]\!]$; and similarly define the sequence $g_i(x)$. The algorithm, then, inserts each key x into the first unoccupied

cell in the following probe sequence: $f_1(x)$, $g_1(x)$, $f_2(x)$, $g_2(x)$, $f_3(x)$, $g_3(x)$,..., as shown in Figure 1. We denote this algorithm that hashes m keys into n cells by SHORTSEQ(n, m), for the shorter sequence.

The Smaller Cluster: SMALLCLUSTER Algorithm

The second algorithm inserts each key into the empty (terminal) cell that is the right neighbor of the smaller cluster among the two clusters containing the initial hashing cells, breaking ties randomly. If one of the initial cells is empty, then the key is inserted into it, and if both of the initial cells are empty, we break ties evenly. Recall that a cluster is a group of consecutively occupied cells whose left and right neighbors are empty cells. This means that one can compute the size of the cluster that contains an initial hashing cell by running two linear probe sequences in opposite directions starting from the initial cell and going to the empty cells at the boundaries. So practically, the algorithm uses four linear probe sequences. We refer to this algorithm by SMALLCLUSTER(n, m) for inserting m keys into n cells (Figure 2).
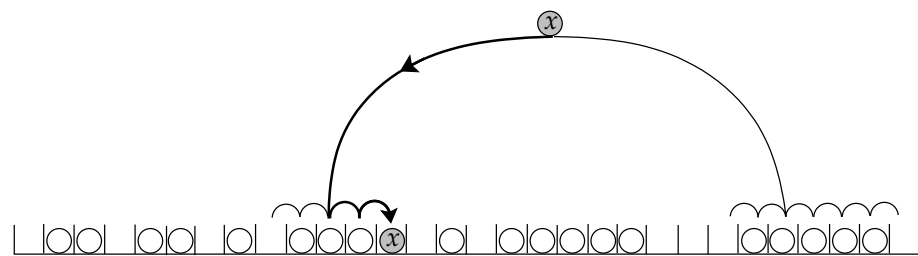

Figure 2. Algorithm SMALLCLUSTER(n, m) inserts each key into the empty cell adjacent to the smaller cluster, breaking ties randomly. The size of the clusters is determined by probing linearly in both directions.

In Section 4.2, we show that algorithms SHORTSEQ and SMALLCLUSTER have unexpected poor performance. Indeed we prove that such algorithms, which always insert any key upon arrival into the empty cell of its two initial cells whenever one of them is empty, produce a cluster of size $\Omega(\log n)$, asymptotically almost surely. To overcome the problems of these algorithms, we introduce *blocking*.

3.2. Hashing with Blocking

The hash table is partitioned into equal-sized disjoint blocks of cells. Whenever a key has two terminal cells, the algorithm considers the information provided by the blocks, e.g., the number of keys it harbors, to make a decision. Thus, the blocking technique enables the algorithm to avoid some of the bad decisions the previous algorithms make. This leads to a more controlled allocation process, and hence, to a more even distribution of the keys. We use the blocking technique to design two two-way linear probing algorithms, and an algorithm that uses linear probing locally within each block. The algorithms are characterized by the way the keys pick their blocks to land in. The worst-case performance of these algorithms is analyzed in Section 5 and proven to be $O(\log \log n)$, w.h.p.

Note also that (for insertion operations only) the algorithms require a counter with each block, but the extra space consumed by these counters is asymptotically negligible. In fact, we will see that the extra space is $O(n / \log \log n)$ in a model in which integers take $O(1)$ space, and at worst $O(n \log \log \log n / \log \log n) = o(n)$ units of memory, w.h.p., in a bit model.

Since the block size for each of the following algorithms is different, we assume throughout and without loss of generality that, whenever we use a block of size β, then n / β is an integer. Recall that the cells are numbered $0, \ldots, n - 1$, and hence, for $i \in [\![n / \beta]\!]$, the i-th block consists of the cells $(i - 1)\beta, \ldots, i\beta - 1$. In other words, the cell $k \in \{0, \ldots, n - 1\}$ belongs to block number $\lambda(k) := \lfloor k / \beta \rfloor + 1$.

Two-Way Locally Linear Probing: LOCALLYLINEAR Algorithm

As a simple example of the blocking technique, we present the following algorithm, which is a trivial application of the two-way chaining scheme [10]. The algorithm does not satisfy the definition of two-way linear probing as we explained earlier, because the linear probes are performed within each block and not along the hash table. That is, whenever the linear probe sequence reaches the right boundary of a block, it continues probing starting from the left boundary of the same block.

The algorithm partitions the hash table into disjoint blocks each of size $\beta_1(n)$, where $\beta_1(n)$ is an integer to be defined later. We save with each block its load, that is, the number of keys it contains, and keep it updated whenever a key is inserted in the block. For each key, we choose two initial hashing cells, and hence two blocks, independently and uniformly at random, with replacement. From the initial cell that belongs to the least loaded block, breaking ties randomly, we probe linearly and cyclically *within the block* until we find an empty cell where we insert the key. If the load of the block is β_1, i.e., it is full, then we check its right neighbor block and so on, until we find a block that is not completely full. We insert the key into the first empty cell there. Notice that only one probe sequence is used to insert any key. The search operation, however, uses two probe sequences as follows. First, we compute the two initial hashing cells. We start probing linearly and cyclically *within* the two (possibly identical) blocks that contain these initial cells. If both probe sequences reach empty cells, or if one of them reaches an empty cell and the other one finishes the block without finding the key, we declare the search to be unsuccessful. If both blocks are full and the probe sequences completely search them without finding the key, then the right neighbors of these blocks (cyclically speaking) are searched sequentially in the same way mentioned above, and so on. We will refer to this algorithm as LOCALLYLINEAR(n, m) for inserting m keys into n cells.

Two-Way Pre-Linear Probing: DECIDEFIRST Algorithm

In the previous two-way linear probing algorithms, each input key initiates linear probe sequences that reach two terminal cells, and then the algorithms decide in which terminal cell the key should be inserted. The following algorithm, however, allows each key to choose two initial hashing cells, and then decides, according to some strategy, which initial cell should start a linear probe sequence to find a terminal cell to harbor the key. Therefore, technically, the insertion process of any key uses only one linear probe sequence, but we still use two sequences for any search.
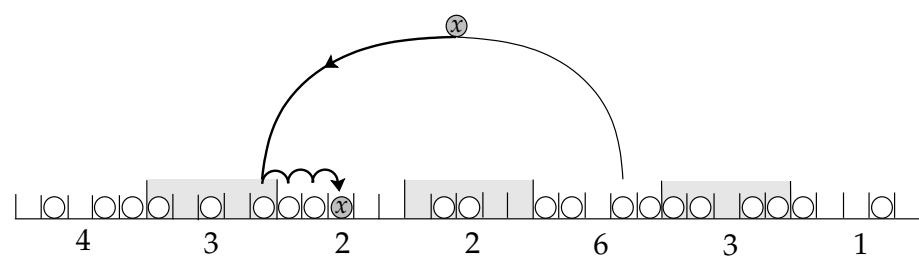

Figure 3. An illustration of algorithm DECIDEFIRST(n, m). The hash table is divided into blocks of size β_2. The number under each block is its weight. Each key decides first to land into the block of smaller weight, breaking ties randomly, then probes linearly to find its terminal cell.

Formally, we describe the algorithm as follows. Let $\alpha \in (0, 1)$ be the load factor. Partition the hash table into blocks of size $\beta_2(n)$, where $\beta_2(n)$ is an integer to be defined later. Each key x still chooses, independently and uniformly at random, two initial hashing cells, say I_x and J_x, and hence, two blocks which we denote by $\lambda(I_x)$ and $\lambda(J_x)$. For convenience, we say that the key x has *landed* in block i, if the linear probe sequence used to insert the key x has started (from the initial hashing cell available for x) in block i. Define the weight of a block to be the number of keys that have landed in it. We save with each block its weight and keep it updated whenever a key lands in it. Now, upon the arrival of key x, the algorithm allows x to land into the block among $\lambda(I_x)$ and $\lambda(J_x)$ of smaller weight, breaking ties randomly. Whence, it starts probing linearly from the initial cell

contained in the block until it finds a terminal cell into which the key x is placed. If, for example, both I_x and J_x belong to the same block, then x lands in $\lambda(I_x)$, and the linear sequence starts from an arbitrarily chosen cell among I_x and J_x.

We will write DECIDEFIRST(n, m) to refer to this algorithm for inserting m keys into n cells. In short, the strategy of DECIDEFIRST(n, m) as illustrated in Figure 3 is: land in the block of smaller weight, walk linearly, and insert into the first empty cell reached.

Two-Way Post-Linear Probing: WALKFIRST Algorithm

We introduce yet another hashing algorithm that achieves $\Theta(\log \log n)$ worst-case search time, in probability, and shows better performance in experiments than DECIDEFIRST algorithm as demonstrated in the simulation results presented in Section 6. Suppose that the load factor $\alpha \in (0, 1/2)$, and that the hash table is divided into blocks of size

$$\beta_3(n) := \left\lceil \frac{\log_2 \log n + 8}{1 - \delta} \right\rceil,$$

where $\delta \in (2\alpha, 1)$ is an arbitrary constant. Define the load of a block to be the number of keys (or occupied cells) it contains. Suppose that we save with each block its load and keep it updated whenever a key is inserted into one of its cells. Recall that each key x has two initial hashing cells. From these initial cells, the algorithm probes linearly and cyclically until it finds two empty cells U_x and V_x, which we call terminal cells. Let $\lambda(U_x)$ and $\lambda(V_x)$ be the blocks that contain these cells. The algorithm, then, inserts the key x into the terminal cell (among U_x and V_x) that belongs to the least loaded block among $\lambda(U_x)$ and $\lambda(V_x)$, breaking ties randomly. We refer to this algorithm of open addressing hashing for inserting m keys into n cells as WALKFIRST(n, m) (Figure 4).
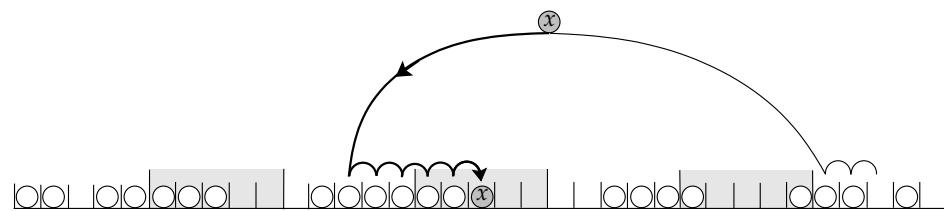

Figure 4. Algorithm WALKFIRST(n, m) inserts each key into the terminal cell that belongs to the least crowded block, breaking ties arbitrarily.

4. Lower Bounds

We prove here that the idea of two-way linear probing alone is not always sufficient to pull off a plausible hashing performance. We prove that a large group of two-way linear probing algorithms have an $\Omega(\log n)$ lower bound on their worst-case search time. We shall first record a lower bound that holds for any two-way linear probing algorithm.

4.1. Universal Lower Bound

The following lower bound holds for any two-way linear probing hashing scheme, in particular, the ones that are presented in this article.

Theorem 5. *Let $n \in \mathbb{N}$, and $m = \lfloor \alpha n \rfloor$, where $\alpha \in (0, 1)$ is a constant. Let **A** be any two-way linear probing algorithm that inserts m keys into a hash table of size n. Then upon termination of **A**, w.h.p., the table contains a cluster of size of at least $\log_2 \log n - O(1)$.*

Proof. Imagine that we have a bin associated with each cell in the hash table. Recall that, for each key x, algorithm **A** chooses two initial cells, and hence two bins, independently and uniformly at random, with replacement. Algorithm **A**, then, probes linearly to find two (possibly identical) terminal cells, and inserts the key x into one of them. Now imagine that after the insertion of each key x, we also insert a ball into the bin associated with the initial cell from which the algorithm started probing to reach the terminal cell into which the key x was placed. If both of the initial cells lead to the same terminal cell, then we

break the tie randomly. Clearly, if there is a bin with k balls, then there is a cluster of size of at least k, because the k balls represent k distinct keys that belong to the same cluster. However, Theorem 1 asserts that the maximum bin load upon termination of algorithm **A** is at least $\log_2 \log n - O(1)$, w.h.p.   □

The above lower bound is valid for all algorithms that satisfy Definition 1. A more general lower bound can be established on all open addressing schemes that use two linear probe sequences where the initial hashing cells are chosen according to some (not necessarily uniform or independent) probability distributions defined on the cells. We still assume that the probe sequences are used to find two (empty) terminal hashing cells, and the key is inserted into one of them according to some strategy. We call such schemes *non-uniform two-way linear probing*. The proof of the following theorem is basically similar to Theorem 5, but by using instead Vöcking's lower bound as stated in Theorem 3.

Theorem 6. *Let $n \in \mathbb{N}$, and $m = \lfloor \alpha n \rfloor$, where $\alpha \in (0,1)$ is a constant. Let **A** be any non-uniform two-way linear probing algorithm that inserts m keys into a hash table of size n where the initial hashing cells are chosen according to some probability distributions. Then the maximum cluster size produced by **A**, upon termination, is at least $\log \log n / (2 \log \phi_2) - O(1)$, w.h.p.*

4.2. Algorithms that Behave Poorly

We characterize some of the inefficient two-way linear probing algorithms. Notice that the main mistake in algorithms SHORTSEQ(n,m) and SMALLCLUSTER(n,m) is that the keys are allowed to be inserted into empty cells even if these cells are very close to some giant clusters. This leads us to the following theorem whose proof utilizes Lemmas 4–6, stated in Appendix A, regarding negative association of random variables. Throughout, we write binomial(n,p) to denote a binomial random variable with parameters $n \in \mathbb{N}$ and $p \in [0,1]$.

Theorem 7. *Let $\alpha \in (0,1)$ be constant. Let **A** be a two-way linear probing algorithm that inserts $m = \lfloor \alpha n \rfloor$ keys into n cells such that, whenever a key chooses an empty and an occupied initial cells, the algorithm inserts the key into the empty one. Then algorithm **A** produces a giant cluster of size $\Omega(\log n)$, w.h.p.*

Proof. Let $\beta = \lfloor b \log_a n \rfloor$ for some positive constants a and b to be defined later, and without loss of generality, assume that $N := n/\beta$ is an integer. Suppose that the hash table is divided into N disjoint blocks, each of size β. For $i \in [\![N]\!]$, let $\mathfrak{B}_i = \{\beta(i-1)+1, \ldots, \beta i\}$ be the set of cells of the i-th block, where we consider the cell numbers in a circular fashion. We say that a cell $j \in [\![n]\!]$ is "covered" if there is a key whose first initial hashing cell is the cell j and its second initial hashing cell is an occupied cell. A block is *covered* if all of its cells are covered. Observe that, if a block is covered, then it is fully occupied. Thus, it suffices to show that there would be a covered block, w.h.p.

For $i \in [\![N]\!]$, let Y_i be the indicator that the i-th block is covered. The random variables $Y_1, \ldots, Y_N$ are negatively associated which can been seen as follows. For $j \in [\![n]\!]$ and $t \in [\![m]\!]$, let $X_j(t)$ be the indicator that the j-th cell is covered by the t-th key, and set $X_0(t) := 1 - \sum_{j=1}^{n} X_j(t)$. Notice that the random variable $X_0(t)$ is binary. The zero-one Lemma asserts that the binary random variables $X_0(t), \ldots, X_n(t)$ are negatively associated. However, since the keys choose their initial hashing cells independently, the random variables $X_0(t), \ldots, X_n(t)$ are mutually independent from the random variables $X_0(t'), \ldots, X_n(t')$, for any distinct $t, t' \in [\![m]\!]$. Thus, by Lemma 5, the union $\cup_{t=1}^{m} \{X_0(t), \ldots, X_n(t)\}$ is a set of negatively associated random variables. The negative association of the Y_i is assured now by Lemma 6 as they can be written as non-decreasing functions of disjoint subsets of the indicators $X_j(t)$. Since the Y_i are negatively associated and identically distributed, then

$$\mathbb{P}\{Y_1 = 0, \ldots, Y_N = 0\} \leq \mathbb{P}\{Y_1 = 0\} \times \cdots \times \mathbb{P}\{Y_N = 0\} \leq \exp(-N\mathbb{P}\{Y_1 = 1\}).$$

Thus, we only need to show that $N\mathbb{P}\{Y_1 = 1\}$ tends to infinity as n goes to infinity. To bound the last probability, we need to focus on the way the first block $\mathfrak{B}_1 = \{1, 2, \ldots, \beta\}$

is covered. For $j \in [\![n]\!]$, let t_j be the smallest $t \in [\![m]\!]$ such that $X_j(t) = 1$ (if such exists), and $m + 1$ otherwise. We say that the first block is "covered in order" if and only if $1 \le t_1 < t_2 < \cdots < t_\beta \le m$. Since there are $\beta!$ orderings of the cells in which they can be covered (for the first time), we have

$$\mathbb{P}\{Y_1 = 1\} = \beta! \, \mathbb{P}\{\mathfrak{B}_1 \text{is covered in order}\} .$$

For $t \in [\![m]\!]$, let $M(t) = 1$ if block $\mathfrak{B}_1$ is full before the insertion of the t-th key, and otherwise be the minimum $i \in \mathfrak{B}_1$ such that the cell i has not been covered yet. Let A be the event that, for all $t \in [\![m]\!]$, the first initial hashing cell of the t-th key is either cell $M(t)$ or a cell outside $\mathfrak{B}_1$. Define the random variable $W := \sum_{t=1}^m W_t$, where W_t is the indicator that the t-th key covers a cell in $\mathfrak{B}_1$. Clearly, if A is true and $W \ge \beta$, then the first block is covered in order. Thus,

$$\mathbb{P}\{Y_1 = 1\} \ge \beta! \, \mathbb{P}\{[W \ge \beta] \cap A\} = \beta! \, \mathbb{P}\{A\}\mathbb{P}\{W \ge \beta \,|\, A\} .$$

However, since the initial hashing cells are chosen independently and uniformly at random, then for n chosen large enough, we have

$$\mathbb{P}\{A\} \ge \left(1 - \frac{\beta}{n}\right)^m \ge e^{-2\beta} ,$$

and for $t \ge \lceil m/2 \rceil$,

$$\mathbb{P}\{W_t = 1 \,|\, A\} = \frac{1}{n - \beta + 1} \cdot \frac{t - 1}{n} \ge \frac{\alpha}{4n} .$$

Therefore, for n that is sufficiently large, we obtain

$$
\begin{aligned}
N\mathbb{P}\{Y_1 = 1\} &\ge N\beta! \, e^{-2\beta} \, \mathbb{P}\{\text{binomial}(\lceil m/2 \rceil, \alpha/(4n)) \ge \beta\} \\
&\ge N\beta! \, e^{-2\beta} \frac{(m/2 - \beta)^\beta}{\beta!} \left(\frac{\alpha}{4n}\right)^\beta \left(1 - \frac{\alpha}{4n}\right)^n \\
&\ge N\left(\frac{\alpha^2}{8e^2}\right)^\beta \left(1 - \frac{2\beta}{m}\right)^\beta \left(1 - \frac{1}{4n}\right)^n \\
&\ge \frac{n}{4\beta}\left(\frac{\alpha^2}{8e^2}\right)^\beta ,
\end{aligned}
$$

which goes to infinity as n approaches infinity whenever $a = 8e^2/\alpha^2$ and b is any positive constant less than 1. □

Clearly, algorithms SHORTSEQ(n, m) and SMALLCLUSTER(n, m) satisfy the condition of Theorem 7. So this corollary follows.

Corollary 1. *Let $n \in \mathbb{N}$, and $m = \lfloor \alpha n \rfloor$, where $\alpha \in (0, 1)$ is constant. The size of the largest cluster generated by algorithm* SHORTSEQ(n, m) *is $\Omega(\log n)$, w.h.p. The same result holds for algorithm* SMALLCLUSTER(n, m).

5. Upper Bounds

In this section, we establish upper bounds on the worst-case performance of the two-way linear probing algorithms that use a blocking technique: LOCALLYLINEAR, DE-CIDEFIRST, and WALKFIRST. We show that the block size can be chosen for each of these algorithms to demonstrate that the maximum cluster size is $O(\log \log n)$, w.h.p.

5.1. Two-Way Locally Linear Probing: LOCALLYLINEAR Algorithm

Recall that algorithm LOCALLYLINEAR(n, m) inserts keys into a hash table with disjoint blocks of size $\beta_1(n)$. We show next that β_1 can be defined such that none of the blocks

are completely full, w.h.p. This means that, whenever we search for any key, most of the time, we only need to search linearly and cyclically the two blocks that the key chooses initially.

Theorem 8. *Let $n \in \mathbb{N}$, and $m = \lfloor \alpha n \rfloor$, where $\alpha \in (0,1)$ is a constant. Let C be the constant defined in Theorem 2, and define*

$$\beta_1(n) := \left\lfloor \frac{\log_2 \log n + C}{1 - \alpha} + 1 \right\rfloor.$$

Then, w.h.p., the maximum unsuccessful search time of LOCALLYLINEAR(n, m) *with blocks of size β_1 is at most $2\beta_1$, and the maximum insertion time is at most $\beta_1 - 1$.*

Proof. Notice the equivalence between algorithm LOCALLYLINEAR(n, m) and the allocation process GREEDYMC$(n/\beta_1, m, 2)$ where m balls (keys) are inserted into n/β_1 bins (blocks) by placing each ball into the least loaded bin among two bins chosen independently and uniformly at random, with replacement. It suffices, therefore, to study the maximum bin load of GREEDYMC$(n/\beta_1, m, 2)$, which we denote by L_n. However, Theorem 2 says that, w.h.p.,

$$L_n \le \log_2 \log n + C + \alpha\beta_1 < (1 - \alpha)\beta_1 + \alpha\beta_1 = \beta_1.$$

and similarly,

$$L_n \ge \log_2 \log n + \alpha\beta_1 - C > \frac{\log_2 \log n + C}{1 - \alpha} - 2C \ge \beta_1 - 2C - 1.$$

□

5.2. Two-Way Pre-Linear Probing: DECIDEFIRST *Algorithm*

The next theorem describes the worst-case performance of algorithm DECIDEFIRST(n, m) with blocks of size β_2 showing that the size of the largest cluster produced by the algorithm is $\Theta(\log \log n)$, w.h.p.

Theorem 9. *Let $n \in \mathbb{N}$, and $m = \lfloor \alpha n \rfloor$, where $\alpha \in (0,1)$ is a constant. There is a constant $\eta > 0$ such that, if*

$$\beta_2(n) := \left\lceil \frac{(1 + \sqrt{2 - \alpha})}{\sqrt{2 - \alpha}(1 - \alpha)} (\log_2 \log n + \eta) \right\rceil,$$

then, w.h.p., the worst-case unsuccessful search time of algorithm DECIDEFIRST(n, m) *with blocks of size β_2 is at most $\xi_n := 12(1 - \alpha)^{-2}(\log_2 \log n + \eta)$, and the maximum insertion time is at most $\xi_n/2$.*

Proof. Assume first that DECIDEFIRST(n, m) is applied to a hash table with blocks of size $\beta = \lceil b(\log_2 \log n + \eta) \rceil$, and that n/β is an integer, where $b = (1 + \epsilon)/(1 - \alpha)$, for some arbitrary constant $\epsilon > 0$. Consider the resulting hash table after termination of the algorithm. Let $M \ge 0$ be the maximum number of consecutive blocks that are fully occupied. Without loss of generality, suppose that these blocks start at block $i + 1$, and let $S = \{i, i+1, \ldots, i+M\}$ represent these full blocks in addition to the left adjacent block that is not fully occupied (Figure 5).

Figure 5. A portion of the hash table showing the largest cluster, and the set S, which consists of the full consecutive blocks and their left neighbor.

Notice that each key chooses two cells (and hence, two possibly identical blocks) independently and uniformly at random. Moreover, any key always lands in the block

of smaller weight. Since there are n/β blocks, and $\lfloor \alpha n \rfloor$ keys, then by Theorem 2, there is a constant $C > 0$ such that the maximum block weight is not more than $\lambda_n := (\alpha b + 1) \log_2 \log n + \alpha b \eta + \alpha + C$, w.h.p. Let A_n denote the event that the maximum block weight is at most λ_n. Let W be the number of keys that have landed in S, i.e., the total weight of blocks contained in S. Plainly, since block i is not full, then all the keys that belong to the M full blocks have landed in S. Thus, $W \geq Mb(\log_2 \log n + \eta)$, deterministically. Now, clearly, if we choose $\eta = C + \alpha$, then the event A_n implies that $(M+1)(\alpha b + 1) \geq Mb$, because otherwise, we have

$$W \leq (M+1)(\alpha b + 1)\left(\log_2 \log n + \frac{\alpha b \eta + \alpha + C}{\alpha b + 1}\right) < Mb(\log_2 \log n + \eta),$$

which is a contradiction. Therefore, A_n yields that

$$M \leq \frac{\alpha b + 1}{(1-\alpha)b - 1} \leq \frac{1 + \epsilon \alpha}{\epsilon(1-\alpha)}.$$

Recall that $(\alpha b + 1) < b = (1 + \epsilon)/(1 - \alpha)$. Again, since block i is not full, the size of the largest cluster is not more than the total weight of the $M + 2$ blocks that cover it. Consequently, the maximum cluster size is, w.h.p., not more than

$$(M+2)(\alpha b + 1)(\log_2 \log n + \eta) \leq \frac{\psi(\epsilon)}{(1-\alpha)^2}(\log_2 \log n + \eta),$$

where $\psi(\epsilon) := 3 - \alpha + (2 - \alpha)\epsilon + 1/\epsilon$. Since ϵ is arbitrary, we choose it such that $\psi(\epsilon)$ is minimum, i.e., $\epsilon = 1/\sqrt{2 - \alpha}$; in other words, $\psi(\epsilon) = 3 - \alpha + 2\sqrt{2 - \alpha} < 6$. This concludes the proof as the maximum unsuccessful search time is at most twice the maximum cluster size plus two. $\quad\square$

Remark. We have showed that, w.h.p., the maximum cluster size produced by DECIDE-FIRST(n, m) is in fact not more than

$$\frac{3 - \alpha + 2\sqrt{2 - \alpha}}{(1-\alpha)^2} \log_2 \log n + O(1) < \frac{6}{(1-\alpha)^2} \log_2 \log n + O(1).$$

5.3. Two-Way Post-Linear Probing: WALKFIRST *Algorithm*

Next, we analyze the worst-case performance of algorithm WALKFIRST(n, m) with blocks of size β_3. Recall that the maximum unsuccessful search time is bounded from above by twice the maximum cluster size plus two. The following theorem asserts that upon termination of the algorithm, it is most likely that every block has at least one empty cell. This implies that the length of the largest cluster is at most $2\beta_3 - 2$.

Theorem 10. *Let $n \in \mathbb{N}$, and $m = \lfloor \alpha n \rfloor$, for some constant $\alpha \in (0, 1/2)$. Let $\delta \in (2\alpha, 1)$ be an arbitrary constant, and define*

$$\beta_3(n) := \left\lceil \frac{\log_2 \log n + 8}{1 - \delta} \right\rceil.$$

Upon termination of algorithm WALKFIRST(n, m) *with blocks of size β_3, the probability that there is a fully loaded block goes to zero as n tends to infinity. That is, w.h.p., the maximum unsuccessful search time of* WALKFIRST(n, m) *is at most $4\beta_3 - 2$, and the maximum insertion time is at most $4\beta_3 - 4$.*

For $k \in [\![m]\!]$, let us denote by A_k the event that after the insertion of k keys (i.e., at time k), none of the blocks is fully loaded. To prove Theorem 10, we shall show that $\mathbb{P}\{A_m^c\} = o(1)$. We do that by using a witness tree argument; see e.g., [11,85,90–93]. We show that, if a fully loaded block exists, then there is a witness binary tree of height β_3 that describes the history of that block. The formal definition of a witness tree is given below.

Let us number the keys $1, \ldots, m$ according to their insertion time. Recall that each key $t \in [\![m]\!]$ has two initial cells which lead to two terminal empty cells belonging to two blocks. Let us denote these two blocks available for the t-th key by X_t and Y_t. Notice that all the initial cells are independent and uniformly distributed. However, all terminal cells—and so their blocks—are not. Nonetheless, for each fixed t, the two random values X_t and Y_t are independent.

The History Tree

We define for each key t a *full history tree* T_t that describes essentially the history of the block that contains the t-th key up to its insertion time. It is a colored binary tree that is labeled by key numbers except possibly the leaves, where each key refers to the block that contains it. Thus, it is indeed a binary tree that represents all the pairs of blocks available for all other keys upon which the final position of the key t relies. Formally, we construct the binary tree node by node in Breadth-First-Search (BFS) order as follows. First, the root of T_t is labeled t, and is colored white. Any white node labeled τ has two children: a left child corresponding to the block X_τ, and a right child corresponding to the block Y_τ. The left child is labeled and colored according to the following rules:

(a)   If the block X_τ contains some keys at the time of insertion of key τ, and the last key inserted in that block, say σ, has not been encountered thus far in the BFS order of the binary tree T_t, then the node is labeled σ and colored white.

(b)   As in case (a), except that σ has already been encountered in the BFS order. We distinguish such nodes by coloring them black, but they are given the same label σ.

(c)   If the block X_τ is empty at the time of insertion of key τ, then it is a "dead end" node without any label and it is colored gray.

Next, the right child of τ is labeled and colored by following the same rules but with the block Y_τ. We continue processing nodes in BFS fashion. A black or gray node in the tree is a leaf and is not processed any further. A white node with label σ is processed in the same way we processed the key τ, but with its two blocks X_σ and Y_σ. We continue recursively constructing the tree until all the leaves are black or gray. See Figure 6 for an example of a full history tree.

Notice that the full history tree is totally deterministic as it does not contain any random value. It is also clear that the full history tree contains at least one gray leaf and every internal (white) node in the tree has two children. Furthermore, since the insertion process is sequential, node values (key numbers) along any path down from the root must be decreasing (so the binary tree has the heap property), because any non-gray child of any node represents the last key inserted in the block containing it at the insertion time of the parent. We will not use the heap property, however.

Clearly, the full history tree permits one to deduce the load of the block that contains the root key at the time of its insertion: it is the length of the shortest path from the root to any gray node. Thus, if the block's load is more than h, then all gray nodes must be at a distance more than h from the root. This leads to the notion of a *truncated history tree* of height h, that is, with $h + 1$ levels of nodes. The top part of the full history tree that includes all nodes at the first $h + 1$ levels is copied, and the remainder is truncated.
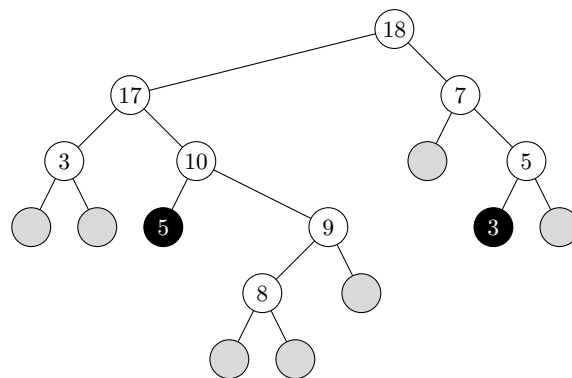
Figure 6. The full history tree of key 18. White nodes represent type (a) nodes. Black nodes are type (b) nodes—they refer to keys already encountered in BFS order. Gray nodes are type (c) nodes—they occur when a key selects an empty block.

We are in particular interested in truncated history trees without gray nodes. Thus, by the property mentioned above, the length of the shortest path from the root to any gray node (and as noted above, there is at least one such node) would have to be at least $h + 1$, and therefore, the load of the block harboring the root's key would have to be at least $h + 1$. More generally, if the load is at least $h + \xi$ for a positive integer ξ, then all nodes at the bottom level of the truncated history tree that are not black nodes (and there is at least one such node) must be white nodes whose children represent keys that belong to blocks with load of at least ξ at their insertion time. We redraw these node as boxes to denote the fact that they represent blocks of load at least ξ, and we call them "block" nodes.

The Witness Tree

Let $\xi \in \mathbb{N}$ be a fixed integer to be picked later. For positive integers h and k, where $h + \xi \leq k \leq m$, a *witness tree* $W_k(h)$ is a truncated history tree of a key in the set $[\![k]\!]$, with $h + 1$ levels of nodes (thus, of height h) and with two types of leaf nodes, black nodes and "block" nodes. This means that each internal node has two children, and the node labels belong to the set $[\![k]\!]$. Each black leaf has a label of an internal node that precedes it in BFS order. Block nodes are unlabeled nodes that represent blocks with load of at least ξ. Block nodes must all be at the furthest level from the root, and there is at least one such node in a witness tree. Notice that every witness tree is deterministic. An example of a witness tree is shown in Figure 7.
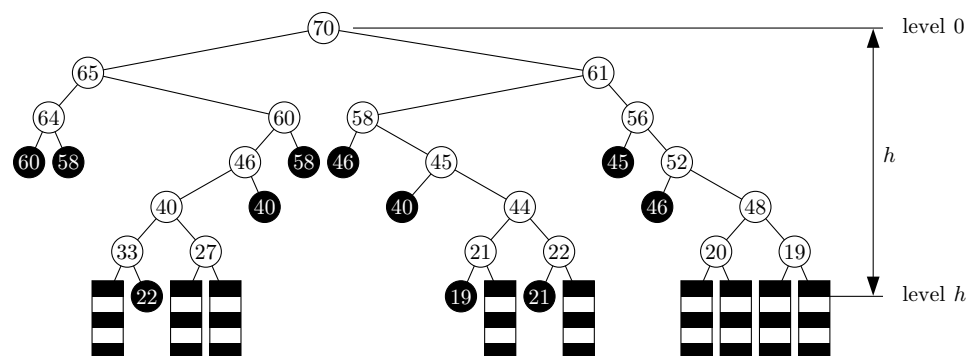
Figure 7. A witness tree of height h which is a truncated history tree without gray nodes. The boxes at the lowest level are block nodes. They represent selected blocks with load of at least ξ. The load of the block that contains key 70 is at least $h + \xi$.

Let $\mathcal{W}_k(h, w, b)$ denote the class of all witness trees $W_k(h)$ of height h that have $w \geq 1$ white (internal) nodes, and $b \leq w$ black nodes (and thus $w - b + 1$ block nodes). Notice that, by definition, the class $\mathcal{W}_k(h, w, b)$ could be empty, e.g., if $w < h$, or $w \geq 2^h$. However, $|\mathcal{W}_k(h, w, b)| \leq 4^w 2^{w+1} w^b k^w$, which is due to the following. Without the labeling, there are

at most 4^w different shape binary trees, because the shape is determined by the w internal nodes, and hence, the number of trees is the Catalan number $\binom{2w}{w}/(w+1) \leq 4^w$. Having fixed the shape, each of the leaves is of one of two types. Each black leaf can receive one of the w white node labels. Each of the white nodes obtains one of k possible labels.

Note that, unlike the full history tree, not every key has a witness tree $W_k(h)$: the key must be placed into a block of load of at least $h + \xi - 1$ just before the insertion time. We say that a witness tree $W_k(h)$ *occurs*, if upon execution of algorithm WALKFIRST, the random choices available for the keys represented by the witness tree are actually as indicated in the witness tree itself. Thus, a witness tree of height h exists if and only if there is a key that is inserted into a block of load of at least $h + \xi - 1$ before the insertion.

Before we embark on the proof of Theorem 10, we highlight three important facts whose proofs are provided in Appendix A. First, we bound the probability that a valid witness tree occurs.

Lemma 1. *Let D denote the event that the number of blocks in* WALKFIRST(n, m) *with load of at least ξ, after termination, is at most $n/(a\beta_3\xi)$, for some constant $a > 0$. For $k \in [\![m]\!]$, let A_k be the event that, after the insertion of k keys, none of the blocks is fully loaded. Then for any positive integers h, w and $k \geq h + \xi$, and a non-negative integer $b \leq w$, we have*

$$\sup_{W_k(h) \in \mathcal{W}_k(h,w,b)} \mathbb{P}\{W_k(h) \text{ occurs} \mid A_{k-1} \cap D\} \leq \frac{4^w \beta_3^{w+b-1}}{(a\xi)^{w-b+1} n^{w+b-1}} .$$

The next lemma asserts that the event D in Lemma 1 is most likely to be true, for sufficiently large $\xi < \beta_3$.

Lemma 2. *Let $\alpha, \delta,$ and β_3 be as defined in Theorem 10. Let N be the number of blocks with load of at least ξ upon termination of algorithm* WALKFIRST(n, m). *If $\xi \geq \delta\beta_3$, then $\mathbb{P}\{N \geq n/(a\beta_3\xi)\} = o(1)$, for any constant $a > 0$.*

Lemma 3 addresses a simple but crucial fact. If the height of a witness tree $W_k(h) \in \mathcal{W}_k(h, w, b)$ is $h \geq 2$, then the number of white nodes w is at least two, (namely, the root and its left child); but what can we say about b, the number of black nodes?

Lemma 3. *In any witness tree $W_k(h) \in \mathcal{W}_k(h, w, b)$, if $h \geq 2$ and $w \leq 2^{h-\eta}$, where $\eta \geq 1$, then the number b of black nodes is $\geq \eta$, i.e., $\mathbb{I}_{\left[[b \geq \eta] \cup [w > 2^{h-\eta}]\right]} = 1$.*

Proof of Theorem 10.

Recall that A_k, for $k \in [\![m]\!]$, is the event that after the insertion of k keys (i.e., at time k), none of the blocks is fully loaded. Notice that $A_m \subseteq A_{m-1} \subseteq \cdots \subseteq A_1$, and the event $A_{\beta_3 - 1}$ is deterministically true. We shall show that $\mathbb{P}\{A_m^c\} = o(1)$. Let D denote the event that the number of blocks with load of at least ξ, after termination, is at most $n/(a\beta_3\xi)$, for some constant $a > 1$ to be decided later. Observe that

$$
\begin{aligned}
\mathbb{P}\{A_m^c\} &\leq \mathbb{P}\{D^c\} + \mathbb{P}\{A_m^c \mid D\} \\
&\leq \mathbb{P}\{D^c\} + \mathbb{P}\{A_m^c \mid A_{m-1} \cap D\} + \mathbb{P}\{A_{m-1}^c \mid D\} \\
&\ \vdots \\
&\leq \mathbb{P}\{D^c\} + \sum_{k=\beta_3}^{m} \mathbb{P}\{A_k^c \mid A_{k-1} \cap D\}.
\end{aligned}
$$

Lemma 2 reveals that $\mathbb{P}\{D^c\} = o(1)$, and hence, we only need to demonstrate that $p_k := \mathbb{P}\{A_k^c \mid A_{k-1} \cap D\} = o(1/n)$, for $k = \beta_3, \ldots, m$. We do that by using the witness tree argument. Let $h, \xi, \eta \in [2, \infty)$ be some integers to be picked later such that $h + \xi \leq \beta_3$. If after the insertion of k keys, there is a block with load of at least $h + \xi$, then a witness tree $W_k(h)$ (with block nodes representing blocks with load of at least ξ) must have occurred.

Recall that the number of white nodes w in any witness tree $W_k(h)$ is at least two. Using Lemmas 1 and 3, we see that

$$
\begin{aligned}
p_k \;&\le\; \sum_{W_k(h)} \mathbb{P}\{W_k(h) \text{ occurs} \mid A_{k-1} \cap D\} \\[4pt]
&\le\; \sum_{w=2}^{2^h-1} \sum_{b=0}^{w} \sum_{W_k(h)\in\mathcal{W}_k(h,w,b)} \mathbb{P}\{W_k(h) \text{ occurs} \mid A_{k-1} \cap D\} \\[4pt]
&\le\; \sum_{w=2}^{2^h-1} \sum_{b=0}^{w} |\mathcal{W}_k(h,w,b)| \sup_{W_k(h)\in\mathcal{W}_k(h,w,b)} \mathbb{P}\{W_k(h) \text{ occurs} \mid A_{k-1} \cap D\} \\[4pt]
&\le\; \sum_{w=2}^{2^h} \sum_{b=0}^{w} \frac{2^{w+1}4^{2w}w^b k^w \beta_3^{w+b-1}}{(a\xi)^{w-b+1} n^{w+b-1}} \mathbb{I}_{\left[\,[b\ge\eta]\cup[w>2^{h-\eta}]\,\right]} \\[4pt]
&\le\; \frac{2n}{a\xi\beta_3} \sum_{w=2}^{2^h} \left(\frac{32\alpha\beta_3}{a\xi}\right)^w \sum_{b=0}^{w} \left(\frac{aw\xi\beta_3}{n}\right)^b \mathbb{I}_{\left[\,[b\ge\eta]\cup[w>2^{h-\eta}]\,\right]}\,.
\end{aligned}
$$

Note that we disallow $b = w+1$, because any witness tree has at least one block node. We split the sum over $w \le 2^{h-\eta}$, and $w > 2^{h-\eta}$. For $w \le 2^{h-\eta}$, we have $b \ge \eta$, and thus

$$
\begin{aligned}
\sum_{b=0}^{w} \left(\frac{aw\xi\beta_3}{n}\right)^b \mathbb{I}_{\left[\,[b\ge\eta]\cup[w>2^{h-\eta}]\,\right]} \;&=\; \sum_{b=\eta}^{w} \left(\frac{aw\xi\beta_3}{n}\right)^b \\[4pt]
&\le\; \left(\frac{aw\xi\beta_3}{n}\right)^\eta \sum_{b=0}^{\infty} \left(\frac{aw\xi\beta_3}{n}\right)^b \\[4pt]
&<\; 2\left(\frac{aw\xi\beta_3}{n}\right)^\eta,
\end{aligned}
$$

provided that n is so large that $a2^{h+1}\xi\beta_3 \le n$, (this insures that $aw\xi\beta_3/n < 1/2$). For $w \in (2^{h-\eta}, 2^h]$, we bound trivially, assuming the same large n condition:

$$
\sum_{b=0}^{w} \left(\frac{aw\xi\beta_3}{n}\right)^b \le 2\,.
$$

In summary, we see that

$$
p_k \le 4n \sum_{w>2^{h-\eta}} \left(\frac{32\alpha\beta_3}{a\xi}\right)^w + 4\left(\frac{a\xi\beta_3}{n}\right)^{\eta-1} \sum_{w=2}^{2^{h-\eta}} \left(\frac{32\alpha\beta_3}{a\xi}\right)^w w^\eta\,.
$$

We set $a = 32$, and $\xi = \lceil \delta\beta_3 \rceil$, so that $32\alpha\beta_3/(a\xi) \le 1/2$, because $\delta \in (2\alpha, 1)$. With this choice, we have

$$
p_k \le \frac{4n}{2^{2^{h-\eta}}} + 4c\left(\frac{32\beta_3^2}{n}\right)^{\eta-1},
$$

where $c = \sum_{w\ge2} w^\eta/2^w$. Clearly, if we put $h = \eta + \lceil \log_2 \log_2 n^\eta \rceil$, and $\eta = 3$, then we see that $h + \xi \le \beta_3$, and $p_k = o(1/n)$. Notice that h and ξ satisfy the technical condition $a2^{h+1}\xi\beta_3 \le n$, asymptotically. $\qquad\square$

Remark. The restriction on α is needed only to prove Lemma 2 where the binomial tail inequality is valid only if $\alpha < 1/2$. Simulation results, as we show next, suggest that a variant of Theorem 10 might hold for any $\alpha \in (0,1)$ with block size $\lfloor (1-\alpha)^{-1} \log_2 \log n \rfloor$.

5.4. Trade-offs

We have seen that, by using two linear probe sequences instead of just one, the maximum unsuccessful search time decreases exponentially from $O(\log n)$ to $O(\log \log n)$. The average search time, however, could at worst double, as shown in the simulation results. Most of the results presented in this article can be improved, by a constant factor though, by increasing the number of hashing choices per key. For example, Theorems 5 and 6 can be easily generalized for open addressing hashing schemes that use $d \geq 2$ linear probe sequences. Similarly, all the two-way linear probing algorithms we design here can be generalized to d-way linear probing schemes. The maximum unsuccessful search time will, then, be at most $d\, C \log_d \log n + O(d)$, where C is a constant depending on α. This means that the best worst-case performance is when $d = 3$ where the minimum of $d / \log d$ is attained. The average search time, on the other hand, could triple.

The performance of these algorithms can be further improved by using Vöcking's scheme LEFTMC(n, m, d), explained in Section 2.5, with $d \geq 2$ hashing choices. The maximum unsuccessful search time, in this case, is at most $C \log \log n / \log \phi_d + O(d)$, for some constant C depending on α. This is minimized when $d = o(\log \log n)$, but we know that it cannot be better than $C \log_2 \log n + O(d)$, because $\lim_{d \to \infty} \phi_d = 2$.

6. Simulation Results

We simulate all linear probing algorithms we discussed in this article with the FCFS replacement strategy: the classical linear probing algorithm CLASSICLINEAR, the locally linear algorithm LOCALLYLINEAR, and the two-way linear probing algorithms SHORTSEQ, SMALLCLUSTER, WALKFIRST, and DECIDEFIRST. For each value of $n \in \{2^8, 2^{12}, 2^{16}, 2^{20}, 2^{22}\}$, and constant $\alpha \in \{0.4, 0.9\}$, we simulate each algorithm 1000 times divided into 10 iterations (experiments). Each iteration consists of 100 simulations of the same algorithm where we insert $\lfloor \alpha n \rfloor$ keys into a hash table with n cells. In each simulation, we compute the average and the maximum successful search and insert times. For each iteration (100 simulations), we compute the average of the average values and and the average of the maximum values computed during the 100 simulations for the successful search and insert times. The overall results are finally averaged over the 10 iterations and recorded in the next tables. Similarly, the average maximum cluster size is computed for each algorithm as it can be used to bound the maximum unsuccessful search time, as mentioned earlier. Notice that in the case of the algorithms CLASSICLINEAR and SHORTSEQ, the successful search time is the same as the insertion time.

Tables 1 and 2 contain the simulation results of the algorithms CLASSICLINEAR, SHORTSEQ, and SMALLCLUSTER. With the exception of the average insertion time of the SMALLCLUSTER algorithm, which is slightly bigger than the CLASSICLINEAR algorithm, it is evident that the average and the worst-case performances of SMALLCLUSTER and SHORTSEQ are better than CLASSICLINEAR. The SMALLCLUSTER algorithm seems to have the best worst-case performance among the three algorithms. This is not a total surprise to us, because the algorithm considers more information (relative to the other two) before it makes its decision of where to insert the keys. It is also clear that there is a nonlinear increase, as a function of n, in the difference between the performances of these algorithms. This may suggest that the worst-case performances of algorithms SHORTSEQ and SMALLCLUSTER are roughly of the order of $\log n$.

Table 1. The average and the maximum successful search and insert times averaged over 10 iterations each consisting of 100 simulations of the algorithms. The best successful search time is shown in boldface and the best insert time is shown in italic.

n	α	CLASSICLINEAR Insert/Search Time		SHORTSEQ Insert/Search Time		SMALLCLUSTER Search Time		SMALLCLUSTER Insert Time	
		Avg	Max	Avg	Max	Avg	Max	Avg	Max
2^8	0.4	1.33	5.75	*1.28*	*4.57*	1.28	**4.69**	1.50	9.96
	0.9	4.38	68.15	*2.86*	*39.72*	3.05	**35.69**	6.63	71.84
2^{12}	0.4	1.33	10.66	*1.28*	*7.35*	1.29	**7.49**	1.52	14.29
	0.9	5.39	275.91	*2.90*	*78.21*	3.07	**66.03**	6.91	118.34
2^{16}	0.4	1.33	16.90	*1.28*	*10.30*	1.29	**10.14**	1.52	18.05
	0.9	5.49	581.70	*2.89*	*120.32*	3.07	**94.58**	6.92	155.36
2^{20}	0.4	1.33	23.64	*1.28*	*13.24*	1.29	**13.03**	1.52	21.41
	0.9	5.50	956.02	*2.89*	*164.54*	3.07	**122.65**	6.92	189.22
2^{22}	0.4	1.33	26.94	*1.28*	*14.94*	1.29	**14.44**	1.52	23.33
	0.9	5.50	1157.34	*2.89*	*188.02*	3.07	**136.62**	6.93	205.91

Table 2. The average maximum cluster size and the average cluster size over 100 simulations of the algorithms. The best performances are drawn in boldface.

n	α	CLASSICLINEAR		SHORTSEQ		SMALLCLUSTER	
		Avg	Max	Avg	Max	Avg	Max
2^8	0.4	2.02	8.32	1.76	6.05	**1.76**	**5.90**
	0.9	15.10	87.63	12.27	50.19	**12.26**	**43.84**
2^{12}	0.4	2.03	14.95	1.75	9.48	**1.75**	**9.05**
	0.9	15.17	337.22	12.35	106.24	**12.34**	**78.75**
2^{16}	0.4	2.02	22.54	1.75	12.76	**1.75**	**12.08**
	0.9	15.16	678.12	12.36	155.26	**12.36**	**107.18**
2^{20}	0.4	2.02	29.92	1.75	16.05	**1.75**	**15.22**
	0.9	15.17	1091.03	12.35	203.16	**12.35**	**136.19**
2^{22}	0.4	2.02	33.81	1.75	17.74	**1.75**	**16.65**
	0.9	15.17	1309.04	12.35	226.44	**12.35**	**150.23**

The simulation data of algorithms LOCALLYLINEAR, WALKFIRST, and DECIDEFIRST are presented in Tables 3–5. These algorithms are simulated with blocks of size $\lfloor (1-\alpha)^{-1} \log_2 \log n \rfloor$. The purpose of this is to show that, practically, the additive and the multiplicative constants appearing in the definitions of the block sizes stated in Theorems 8–10 can be chosen to be small. The hash table is partitioned into equal-sized blocks, except possibly the last one. The average and the maximum values of the successful search time, inset time, and cluster size (averaged over 10 iterations each consisting of 100 simulations of the algorithms) are recorded in the tables below where the best performances are drawn in boldface.

Results show that the LOCALLYLINEAR algorithm has the best performance, whereas WALKFIRST appears to perform better than DECIDEFIRST. Indeed, the sizes of the cluster produced by WALKFIRST appears to be very close to that of LOCALLYLINEAR. This supports the conjecture that Theorem 10 is, in fact, true for any constant load factor $\alpha \in (0,1)$, and the maximum unsuccessful search time of WALKFIRST is at most $4(1-\alpha)^{-1} \log_2 \log n + O(1)$, w.h.p. The average maximum cluster size of DECIDEFIRST seems to be close to the other ones when α is small, but it almost doubles when α is large. This may suggest that the multiplicative constant in the maximum unsuccessful search time established in Theorem 9 could be improved.

Table 3. The average and the maximum successful search time averaged over 10 iterations each consisting of 100 simulations of the algorithms. The best performances are drawn in boldface.

n	α	LOCALLYLINEAR		WALKFIRST		DECIDEFIRST	
		Avg	**Max**	**Avg**	**Max**	**Avg**	**Max**
2^8	0.4	**1.73**	**4.73**	1.78	5.32	1.75	5.26
	0.9	**4.76**	**36.23**	4.76	43.98	5.06	59.69
2^{12}	0.4	**1.74**	**6.25**	1.80	7.86	1.78	7.88
	0.9	**4.76**	**47.66**	4.80	67.04	4.94	108.97
2^{16}	0.4	**1.76**	**7.93**	1.80	9.84	1.78	10.08
	0.9	**4.78**	**56.40**	4.89	89.77	5.18	137.51
2^{20}	0.4	**1.76**	**8.42**	1.81	12.08	1.79	12.39
	0.9	**4.77**	**65.07**	4.98	108.24	5.26	162.04
2^{22}	0.4	**1.76**	**9.18**	1.81	12.88	1.79	13.37
	0.9	**4.80**	**71.69**	5.04	118.06	5.32	181.46

Table 4. The average and the maximum insert time averaged over 10 iterations each consisting of 100 simulations of the algorithms. The best performances are drawn in boldface.

n	α	LOCALLYLINEAR		WALKFIRST		DECIDEFIRST	
		Avg	**Max**	**Avg**	**Max**	**Avg**	**Max**
2^8	0.4	**1.14**	**2.78**	2.52	6.05	1.15	3.30
	0.9	**2.89**	**22.60**	6.19	48.00	3.19	42.64
2^{12}	0.4	**1.14**	**3.38**	2.53	8.48	1.17	5.19
	0.9	**2.91**	**27.22**	6.28	69.30	3.16	84.52
2^{16}	0.4	**1.15**	**4.08**	2.53	10.40	1.17	6.56
	0.9	**2.84**	**31.21**	6.43	91.21	3.17	106.09
2^{20}	0.4	**1.15**	**4.64**	2.54	12.58	1.18	8.16
	0.9	**2.89**	**35.21**	6.54	109.71	3.22	117.42
2^{22}	0.4	**1.15**	**4.99**	2.54	13.41	1.18	8.83
	0.9	**2.91**	**38.75**	6.61	119.07	3.26	132.83

Table 5. The average and the maximum cluster sizes averaged over 10 iterations each consisting of 100 simulations of the algorithms. The best performances are drawn in boldface.

n	α	LOCALLYLINEAR		WALKFIRST		DECIDEFIRST	
		Avg	**Max**	**Avg**	**Max**	**Avg**	**Max**
2^8	0.4	**1.57**	**4.34**	1.65	4.70	1.63	4.81
	0.9	**12.18**	**33.35**	12.54	34.40	13.48	47.76
2^{12}	0.4	**1.62**	**6.06**	1.68	6.32	1.68	6.82
	0.9	**12.42**	**48.76**	12.78	51.80	13.45	94.98
2^{16}	0.4	**1.62**	**7.14**	1.68	7.31	1.68	8.92
	0.9	**12.66**	**59.61**	12.98	62.24	13.53	125.40
2^{20}	0.4	**1.65**	**8.25**	1.71	8.50	1.71	10.76
	0.9	**12.83**	**67.23**	13.11	69.45	13.62	145.30
2^{22}	0.4	**1.62**	**8.90**	1.71	8.95	1.71	11.46
	0.9	**12.72**	**65.58**	13.19	73.22	13.66	164.45

Comparing the simulation data from all tables, one can see that the best average performance is achieved by the algorithms LOCALLYLINEAR and SHORTSEQ. Notice that the SHORTSEQ algorithm achieves the best average successful search time when $\alpha = 0.9$. The best (average and maximum) insertion time is achieved by the LOCALLYLINEAR

algorithm. On the other hand, algorithms WALKFIRST and LOCALLYLINEAR are superior to the others in worst-case performance. It is worth noting that surprisingly, the worst-case successful search time of SMALLCLUSTER is very close to the one achieved by WALKFIRST and better than that of DECIDEFIRST, although, it appears that the difference becomes larger as *n* increases.

7. Conclusions

In this research, we designed efficient open addressing hashing schemes that improve the worst-case performance of classical linear probing. We proposed two-way linear probing hashing schemes that use two independent linear probe sequences and accomplish $\Theta(\log \log n)$ worst-case insertion and search times, w.h.p. The common idea of these schemes is the successful marriage between the two-way linear probing concept and the blocking technique where the hash table is partitioned into equal-sized blocks. Simulation and comparison results supported our theoretical analyses of all algorithms discussed in this research and illustrated that the worst-case and average performances of such schemes are practically plausible. Thus, we conclude that two-way linear probing hashing with blocking has several advantages over other proposed hashing methods as it reduces the worst-case behavior of hashing, it requires only two hash functions, it is easy to parallelize, it is pointer-free and easy to implement, and it does not require any rearrangement of keys or rehashing. Its maximum cluster size is $O(\log \log n)$, and its average-case performance can be at most twice the classical linear probing as shown in the simulation results.

Furthermore, we also showed that not every two-way linear probing algorithm has a good worst-case performance. We proved that, if the two-way linear probing algorithm always inserts any key upon arrival into the empty cell of its two initial cells whenever one of them is empty, then, w.h.p., a cluster of size $\Omega(\log n)$ emerges in the hash table.

Our results suggest that two-way linear probing may be a more promising open addressing hashing scheme than classical linear probing for many applications, including, e.g., mobile networks [94,95].

It would be interesting to extend the result of Theorem 10 to any constant load factor $\alpha \in (0, 1)$—as is suggested by the simulation results—and to investigate its applications to other problems in computer science.

Author Contributions: Conceptualization and methodology, L.D. and E.M.; validation and formal analysis, K.D., L.D. and E.M.; writing—original draft preparation, E.M.; writing—review and editing, K.D. and L.D.; supervision and project administration, L.D. All authors have read and agreed to the published version of the manuscript.

Appendix A

Appendix A.1. Lemmas Needed for Theorem 10

For completeness, we prove the lemmas used in the proof of Theorem 10.

Lemma 1. *Let D denote the event that the number of blocks in* WALKFIRST(n, m) *with load of at least ξ, after termination, is at most $n/(a\beta_3\xi)$, for some constant $a > 0$. For $k \in [\![m]\!]$, let A_k be the event that after the insertion of k keys, none of the blocks is fully loaded. Then for any positive integers h, w and $k \geq h + \xi$, and a non-negative integer $b \leq w$, we have*

$$\sup_{W_k(h) \in \mathcal{W}_k(h,w,b)} \mathbb{P}\{W_k(h) \text{ occurs} \mid A_{k-1} \cap D\} \leq \frac{4^w \beta_3^{w+b-1}}{(a\xi)^{w-b+1} n^{w+b-1}}.$$

Proof. Notice first that given A_{k-1}, the probability that any fixed key in the set $[\![k]\!]$ chooses a certain block is at most $2\beta_3/n$. Let $W_k(h) \in \mathcal{W}_k(h, w, b)$ be a fixed witness tree. We compute the probability that $W_k(h)$ occurs given that A_{k-1} is true, by looking at each node

in BFS order. Suppose that we are at an internal node, say u, in $W_k(h)$. We would like to find the conditional probability that a certain child of node u is exactly as indicated in the witness tree, given that A_{k-1} is true, and everything is revealed except those nodes that precede u in the BFS order. This depends on the type of the child. If the child is white or black, the conditional probability is not more than $2\beta_3/n$. This is because each key refers to the unique block that contains it, and moreover, the initial hashing cells of all keys are independent. Multiplying just these conditional probabilities yields $(2\beta_3/n)^{w+b-1}$, as there are $w + b - 1$ edges in the witness tree that have a white or black nodes as their lower endpoint. On the other hand, if the child is a block node, the conditional probability is at most $2/(a\xi)$. This is because a block node corresponds to a block with load of at least ξ, and there are at most $n/(a\beta_3\xi)$ such blocks each of which is chosen with probability of at most $2\beta_3/n$. Since there are $w - b + 1$ block nodes, the result follows plainly by multiplying all the conditional probabilities. □

To prove Lemma 2, we need to recall the following binomial tail inequality [96]: for $p \in (0, 1)$, and any positive integers r, and $t \geq \eta rp$, for some $\eta > 1$, we have

$$\mathbb{P}\{\text{binomial}(r, p) \geq t\} \leq \left(\varphi\left(\frac{t}{rp}\right)\right)^t \leq (\varphi(\eta))^t,$$

where $\varphi(x) = x^{-1}e^{1-1/x}$, which is decreasing on $(1, \infty)$. Notice that $\varphi(x) < 1$, for any $x > 1$, because $1/x = (1 - z) < e^{-z} = e^{1/x-1}$, for some $z \in (0, 1)$.

Lemma 2. *Let α, δ, and β_3 be as defined in Theorem 10. Let N be the number of blocks with load of at least ξ upon termination of algorithm* WALKFIRST(n, m). *If $\xi \geq \delta\beta_3$, then $\mathbb{P}\{N \geq n/(a\beta_3\xi)\} = o(1)$, for any constant $a > 0$.*

Proof. Fix $\xi \geq \delta\beta_3$. Let B denote the last block in the hash table, i.e., B consists of the cells $n - \beta_3, \ldots, n - 1$. Let L be the load of B after termination. Since the loads of the blocks are identically distributed, we have

$$\mathbf{E}[N] = \frac{n}{\beta_3}\mathbb{P}\{L \geq \xi\}.$$

Let S be the set of the consecutively occupied cells, after termination, that occur between the first empty cell to the left of the block B and the cell $n - \beta_3$; see Figure A1.

Figure A1. The last part of the hash table showing clusters, the last block B, and the set S.

We say that a key is born in a set of cells A if at least one of its two initial hashing cells belong to A. For convenience, we write $\nu(A)$ to denote the number of keys that are born in A. Obviously, $\nu(A)$ is binomial$(m, 2|A|/n)$. Since the cell adjacent to the left boundary of S is empty, all the keys that are inserted in S are actually born in S. That is, if $|S| = j$, then $\nu(S) \geq j$. So, by the binomial tail inequality given earlier, we see that

$$\mathbb{P}\{|S| = j\} = \mathbb{P}\{[\nu(S) \geq j] \cap [|S| = j]\} \leq \mathbb{P}\{\text{binomial}(m, 2j/n) \geq j\} \leq c^j,$$

where the constant $c := \varphi(1/(2\alpha)) = 2\alpha e^{1-2\alpha} < 1$, because $\alpha < 1/2$. Let

$$\ell := \log_c \frac{1 - c}{\xi^2} = O(\log \beta_3).$$

and notice that, for n that is large enough,

$$\xi \geq \delta\beta_3 \geq \frac{\delta 2m(\ell + \beta_3)}{(1 + \ell/\beta_3)2\alpha n} \geq y\frac{2m(\ell + \beta_3)}{n},$$

where $y = 1/2 + \delta/(4\alpha) > 1$, because $\delta \in (2\alpha, 1)$. Clearly, by the same property of S stated above, $L \leq \nu(S \cup B)$; and hence, by the binomial tail inequality again, we conclude that, for n that is sufficiently large,

$$
\begin{aligned}
\mathbb{P}\{L \geq \xi\} &\leq& \mathbb{P}\{[\nu(S \cup B) \geq \xi] \cap [|S| \leq \ell]\} + \sum_{j=\ell}^{m} \mathbb{P}\{|S| = j\} \\
&\leq& \mathbb{P}\{\text{binomial}(m, 2(\ell + \beta_3)/n) \geq \xi\} + \frac{c^{\ell}}{1 - c} \\
&\leq& (\varphi(y))^{\xi} + \frac{c^{\ell}}{1 - c} \leq \frac{1}{\xi^2} + \frac{1}{\xi^2} = \frac{2}{\xi^2}.
\end{aligned}
$$

Thence, $\mathbf{E}[N] \leq 2n/(\beta_3\xi^2)$ which implies by Markov's inequality that

$$\mathbb{P}\left\{N \geq \frac{n}{a\beta_3\xi}\right\} \leq \frac{2a}{\xi} = o(1).$$

$\square$

Lemma 3. *In any witness tree $W_k(h) \in \mathcal{W}_k(h, w, b)$, if $h \geq 2$ and $w \leq 2^{h-\eta}$, where $\eta \geq 1$, then the number b of black nodes is $\geq \eta$, i.e., $\mathbb{I}_{[[b \geq \eta] \cup [w > 2^{h-\eta}]]} = 1$.*

Proof. Note that any witness tree has at least one block node at distance h from the root. If we have b black nodes, the number of block nodes is at least 2^{h-b}. Since $w \leq 2^{h-\eta}$, then $2^{h-\eta} - b + 1 \geq w - b + 1 \geq 2^{h-b}$. If $b = 0$, then we have a contradiction. So, assume $b \geq 1$. But then $2^{h-\eta} \geq 2^{h-b}$; that is, $b \geq \eta$. $\square$

Appendix A.2. Lemmas Needed for Theorem 7

The following definition and lemmas are used to prove Theorem 7.

Definition 1 (See, e.g., [97]). *Any non-negative random variables $X_1, \ldots, X_n$ are said to be negatively associated, if for every disjoint index subsets $I, J \subseteq [\![n]\!]$, and for any functions $f : \mathbb{R}^{|I|} \to \mathbb{R}$, and $g : \mathbb{R}^{|J|} \to \mathbb{R}$ that are both non-decreasing or both non-increasing (component-wise), we have*

$$\mathbf{E}[f(X_i, i \in I)\,g(X_j, j \in J)] \leq \mathbf{E}[f(X_i, i \in I)]\,\mathbf{E}[g(X_j, j \in J)].$$

Once we establish that $X_1, \ldots, X_n$ are negatively associated, it follows, by considering inductively the indicator functions, that

$$\mathbb{P}\{X_1 < x_1, \ldots, X_n, < x_n\} \leq \prod_{i=1}^{n} \mathbb{P}\{X_i < x_i\}.$$

The next lemmas, which are proven in [97–99], provide some tools for establishing the negative association.

Lemma 4 (Zero-One Lemma). *Any binary random variables $X_1, \ldots, X_n$ whose sum is one are negatively associated.*

Lemma 5. *If $\{X_1, \ldots, X_n\}$ and $\{Y_1, \ldots, Y_m\}$ are independent sets of negatively associated random variables, then the union $\{X_1, \ldots, X_n, Y_1, \ldots, Y_m\}$ is also a set of negatively associated random variables.*

Lemma 6. *Suppose that $X_1, \ldots, X_n$ are negatively associated. Let $I_1, \ldots, I_k \subseteq [\![n]\!]$ be disjoint index subsets, for some positive integer k. For $j \in [\![k]\!]$, let $h_j : \mathbb{R}^{|I_j|} \to \mathbb{R}$ be non-decreasing functions, and define $Z_j = h_j(X_i, i \in I_j)$. Then the random variables $Z_1, \ldots, Z_k$ are negatively associated. In other words, non-decreasing functions of disjoint subsets of negatively associated random variables are also negatively associated. The same holds if h_j are non-increasing functions.*

References

1. Peterson, W.W. Addressing for random-access storage. *IBM Journal Of Res. Dev.* **1957**, *1*, 130–146.
2. Gonnet, G.H.; Baeza-Yates, R. *Handbook of Algorithms and Data Structures*; Addison-Wesley: Workingham, UK, 1991.
3. Knuth, D.E. *The Art of Computer Programming, Vol. 3: Sorting and Searching*; Addison-Wesley: Reading, UK, 1973.
4. Vitter, J.S.; Flajolet, P. Average-case analysis of algorithms and data structures. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*; van Leeuwen, J., ed.; MIT Press: Amsterdam, The Netherlands, 1990; pp. 431–524.
5. Janson, S.; Viola, A. A unified approach to linear probing hashing with buckets. *Algorithmica* **2016**, *75*, 724–781.
6. Pagh, A.; Pagh, R.; Ružić, M. Linear probing with 5-wise independence. *SIAM Rev.* **2011**, *53*, 547–558.
7. Richter, S.; Alvarez, V.; Dittrich, J. A seven-dimensional analysis of hashing methods and its implications on query processing. *Porc. Vldb Endow.* **2015**, *9*, 96–107.
8. Thorup, M.; Zhang, Y. Tabulation-based 5-independent hashing with applications to linear probing and second moment estimation. *SIAM J. Comput.* **2012**, *41*, 293–331.
9. Pittel, B. Linear probing: The probable largest search time grows logarithmically with the number of records. *J. Algorithms* **1987**, *8*, 236–249.
10. Azar, Y.; Broder, A.Z.; Karlin, A.R.; Upfal, E. Balanced allocations. *SIAM J. Comput.* **2000**, *29*, 180–200.
11. Vöcking, B. How asymmetry helps load balancing. *J. ACM* **2003**, *50*, 568–589.
12. Malalla, E. Two-Way Hashing with Separate Chaining and Linear Probing. Ph.D. Thesis, School of Computer Science, McGill University, 2004.
13. Dalal, K.; Devroye, L.; Malalla, E.; McLeish, E. Two-way chaining with reassignment. *SIAM J. Comput.* **2005**, *35*, 327–340.
14. Berenbrink, P.; Czumaj, A.; Steger, A.; Vöcking, B. Balanced allocations: the heavily loaded case. *SIAM J. Comput.* **2006**, *35*, 1350–1385.
15. Malalla, E. (2010) Two-way chaining for non-uniform distributions. *International Journal of Computer Mathematics* **2010**, *87:2*, 454–473.
16. Morris, R. Scatter storage techniques. *Commun. ACM* **1968**, *11*, 38–44.
17. de Balbine, G. Computational Analysis of the Random Components Induced by Binary Equivalence Relations. Ph.D. Thesis, California Institute of Technology, Pasadena, CA, USA, 1969.
18. Ullman, J.D. A note on the efficiency of hashing functions. *J. ACM* **1972**, *19*, 569–575.
19. Yao, A.C. Uniform hashing is optimal. *J. ACM* **1985**, *32*, 687–693.
20. Munro, J.I.; Celis, P. Techniques for collision resolution in hash tables with open addressing. In Proceedings of the 1986 Fall Joint Computer Conference, Dallas, TX, USA, 2–6 November 1999; pp. 601–610.
21. Poblete, P.V.; Munro, J.I. Last-Come-First-Served hashing. *J. Algorithms* **1989**, *10*, 228–248.
22. Celis, P. Robin Hood Hashing. Ph.D. Thesis, Computer Science Department, University of Waterloo, Waterloo, ON, Canada, 1986.
23. Celis, P.; Larson, P.; Munro, J.I. Robin Hood hashing (preliminary report). In Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science (FOCS), Portland, OR, USA, 21–23 October 1985; pp. 281–288.
24. Larson, P. Analysis of uniform hashing. *J. ACM* **1983**, *30*, 805–819.
25. Bollobás, B.; Broder, A.Z.; Simon, I. The cost distribution of clustering in random probing. *J. ACM* **1990**, *37*, 224–237.
26. Knuth, D.E. Notes on "open" Addressing. Unpublished Notes. 1963. Available online: https://jeffe.cs.illinois.edu/teaching/datastructures/2011/notes/knuth-OALP.pdf (accessed on 31 August 2023).
27. A. G. Konheim and B. Weiss An occupancy discipline and applications. *SIAM Journal on Applied Mathematics* 1966, 14, 1266–1274.
28. Mendelson, H.; Yechiali, U. A new approach to the analysis of linear probing schemes. *J. ACM* **1980**, *27*, 474–483.
29. Guibas, L.J. The analysis of hashing techniques that exhibit K-ary clustering. *J. ACM* **1978**, *25*, 544–555.
30. Lueker, G.S.; Molodowitch, M. More analysis of double hashing. *Combinatorica* **1993**, *13*, 83–96.
31. Schmidt, J.P.; Siegel, A. *Double Hashing Is Computable and Randomizable with Universal Hash Functions*; Submitted. A Full Version Is Available as Technical Report TR1995-686; Computer Science Department, New York University, New York, NY, USA, 1995.
32. Siegel, A.; Schmidt, J.P. *Closed Hashing Is Computable and Optimally Randomizable with Universal Hash Functions*; Submitted. A Full Version Is Available as Technical Report TR1995-687; Computer Science Department, New York University, New York, NY, USA, 1995.
33. Flajolet, P.; Poblete, P.V.; Viola, A. On the analysis of linear probing hashing. *Algorithmica* **1998**, *22*, 490–515.
34. D. E. Knuth Linear probing and graphs, average-case analysis for algorithms. *Algorithmica* 1998, 22, 561–568.
35. Viola, A.; Poblete, P.V. The analysis of linear probing hashing with buckets. *Algorithmica* **1998**, *21*, 37–71.
36. Janson, S. Asymptotic distribution for the cost of linear probing hashing. *Random Struct. Algorithms* **2001**, *19*, 438–471.
37. Gonnet, G.H. Open addressing hashing with unequal-probability keys. *J. Comput. Syst. Sci.* **1980**, *20*, 354–367.

38.　Aldous, D. Hashing with linear probing, under non-uniform probabilities. *Probab. Eng. Inform. Sci.* **1988**, *2*, 1–14.

39.　Pflug, G.C.; Kessler, H.W. Linear probing with a nonuniform address distribution. *J. ACM* **1987**, *34*, 397–410.

40.　Poblete, P.V.; Viola, A.; Munro, J.I. Analyzing the LCFS linear probing hashing algorithm with the help of Maple. *Maple Tech. Newlett.* **1997**, *4*, 8–13.

41.　Janson, S. *Individual Displacements for Linear Probing Hashing with Different Insertion Policies*; Technical Report No. 35; Department of Mathematics, Uppsala University, Uppsala, Sweden, 2003.

42.　Viola, A. Exact distributions of individual displacements in linear probing hashing. *ACM Trans. Algorithms* **2005**, *1*, 214–242.

43.　Chassaing, P.; Louchard, G. Phase transition for parking blocks, Brownian excursion and coalescence. *Random Struct. Algorithms* **2002**, *21*, 76–119.

44.　Gonnet, G.H. Expected length of the longest probe sequence in hash code searching. *J. ACM* **1981**, *28*, 289–304.

45.　Devroye, L.; Morin, P.; Viola, A. On worst-case Robin Hood hashing. *SIAM J. Comput.* **2004**, *33*, 923–936.

46.　Gonnet, G.H.; Munro, J.I. Efficient ordering of hash tables. *SIAM J. Comput.* **1979**, *8*, 463–478.

47.　Brent, R.P. Reducing the retrieval time of scatter storage techniques. *Commun. ACM* **1973**, *16*, 105–109.

48.　Madison, J.A.T. Fast lookup in hash tables with direct rehashing. *Comput. J.* **1980**, *23*, 188–189.

49.　Mallach, E.G. Scatter storage techniques: a uniform viewpoint and a method for reducing retrieval times. *Comput. J.* **1977**, *20*, 137–140.

50.　Rivest, R.L. Optimal arrangement of keys in a hash table. *J. ACM* **1978**, *25*, 200–209.

51.　Pagh, R.; Rodler, F.F. Cuckoo hashing. In Proceedings of the European Symposium on Algorithms; LNCS 2161, Springer-Verlag: Berlin/Heidelberg, Germany, 2001; pp. 121–133.

52.　Devroye, L.; Morin, P. Cuckoo hashing: further analysis. *Inf. Process. Lett.* **2003**, *86*, 215–219.

53.　Östlin, A.; Pagh, R. Uniform hashing in constant time and linear space. In Proceedings of the 35th Annual ACM Symposium on Theory of Computing (STOC), San Diego, CA, USA, 9–11 June 2003; pp. 622–628.

54.　Dietzfelbinger, M.; Wolfel, P. Almost random graphs with simple hash functions. In Proceedings of the 35th Annual ACM Symposium on Theory of Computing (STOC), San Diego, CA, USA, 9–11 June 2003; pp. 629–638.

55.　Fotakis, D.; Pagh, R.; S.; ers, P.; Spirakis, P.G. Space efficient hash tables with worst case constant access time. In Proceedings of the 20th Symposium on Theoretical Aspects of Computer Science; LNCS 2607, Springer-Verlag: Berlin/Heidelberg, Germany, 2003; pp. 271–282.

56.　Fountoulakis, N.; Panagiotou, K. Sharp Load Thresholds for Cuckoo Hashing. *Random Struct. Algorithms* **2012**, *41*, 306–333.

57.　Lehman, E.; Panigrahy, R. 3.5-Way Cuckoo Hashing for the Price of 2-and-a-Bit. In Proceedings of the 17th Annual European Symposium, Copenhagen, Denmark, 7–9 September 2009; pp. 671–681.

58.　Dietzfelbinger, M.; Weidling, C. Balanced allocation and dictionaries with tightly packed constant size bins. *Theor. Comput. Sci.* **2007**, *380*, 47–68.

59.　Fountoulakis, N.; Panagiotou, K.; Steger, A. On the Insertion Time of Cuckoo Hashing. *SIAM J. Comput.* **2013**, *42*, 2156–2181.

60.　Frieze, A.M.; Melsted, P. Maximum Matchings in Random Bipartite Graphs and the Space Utilization of Cuckoo Hash Tables. *Random Struct. Algorithms* **2012**, *41*, 334–364, .

61.　Walzer, S. Load thresholds for cuckoo hashing with overlapping blocks. In Proceedings of the 45th International Colloquium on Automata, Languages, and Programming, Prague, Czech Republic, 9–13 July 2018; pp. 102:1–102:10.

62.　Frieze, A.M.; Melsted, P.; Mitzenmacher, M. An analysis of random-walk cuckoo hashing. *SIAM J. Comput.* **2011**, *40*, 291–308.

63.　Pagh, R. Hash and displace: Efficient evaluation of minimal perfect hash functions. In *Proceedings of the 6th International Workshop on Algorithms and Data Structures*; LNCS 1663; Springer-Verlag: Berlin/Heidelberg, Germany, 1999; pp. 49–54.

64.　Pagh, R. On the cell probe complexity of membership and perfect hashing. In Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC), Crete, Greece, 6–8 July 2001; pp. 425–432.

65.　Dietzfelbinger, M.; auf der Heide, F.M. High performance universal hashing, with applications to shared memory simulations. In *Data Structures and Efficient Algorithms*; LNCS 594; Springer-Verlag: Berlin/Heidelberg, Germany, 1992; pp. 250–269.

66.　Fredman, M.; Komlós, J.; Szemerédi, E. Storing a sparse table with $O(1)$ worst case access time. *J. ACM* **1984**, *31*, 538–544.

67.　Dietzfelbinger, M.; Karlin, A.; Mehlhorn, K.; auf, der Heide, F.M.; Rohnert, H.; Tarjan, R. Dynamic perfect hashing: upper and lower bounds. *SIAM J. Comput.* **1994**, *23*, 738–761.

68.　Dietzfelbinger, M.; auf der Heide, F.M. A new universal class of hash functions and dynamic hashing in real time. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming*; LNCS 443; Springer-Verlag: Berlin/Heidelberg, Germany, 1990; pp. 6–19.

69.　Broder, A.Z.; Karlin, A. Multilevel adaptive hashing. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*; ACM Press: New York, NY, USA, 2000 ; pp. 43–53.

70.　Dietzfelbinger, M.; Gil, J.; Matias, Y.; Pippenger, N. Polynomial hash functions are reliable (extended abstract). In *Proceedings of the 19th International Colloquium on Automata, Languages and Programming*; LNCS 623; Springer-Verlag: Berlin/Heidelberg, Germany, 1992; pp. 235–246.

71.　Johnson, N.L.; Kotz, S. *Urn Models and Their Application: An Approach to Modern Discrete Probability Theory*;, John Wiley: New York, NY, USA , 1977.

72.　Kolchin, V.F.; Sevast'yanov, B.A.; Chistyakov, V.P. *Random Allocations*; V. H. Winston & Sons: Washington, DC, USA, 1978.

73. Devroye, L. The expected length of the longest probe sequence for bucket searching when the distribution is not uniform. *J. Algorithms* **1985**, *6*, 1–9.

74. Raab, M.; Steger, A. "Balls and bins"—A simple and tight analysis. In Proceedings of the 2nd Workshop on Randomization and Approximation Techniques in Computer Science; LNCS 1518; Springer-Verlag: Berlin/Heidelberg, Germany, 1998; pp. 159–170.

75. Mitzenmacher, M.D. The Power of Two Choices in Randomized Load Balancing. Ph.D. Thesis, Computer Science Department, University of California at Berkeley, Berkeley, CA, USA, 1996.

76. Karp, R.; Luby, M.; auf der Heide, F.M. Efficient PRAM simulation on a distributed memory machine. *Algorithmica* **1996**, *16*, 245–281.

77. Eager, D.L.; Lazowska, E.D.; Zahorjan, J. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Softw. Eng.* **1986**, *12*, 662–675.

78. Vöcking, B. Symmetric vs. asymmetric multiple-choice algorithms. In Proceedings of the 2nd ARACNE Workshop, Aarhus, Denmark, 27 August 2001; pp. 7–15.

79. Adler, M.; Berenbrink, P.; Schroeder, K. Analyzing an infinite parallel job allocation process. In Proceedings of the 6th European Symposium on Algorithms, Venice, Italy, 24–26 August 1998; pp.417–428.

80. Adler, M.; Chakrabarti, S.; Mitzenmacher, M.; Rasmussen, L. Parallel randomized load balancing. In Proceedings of the 27th Annual ACM Symposium on Theory of Computing (STOC), Las Vegas, NV, USA, 29 May–1 June 1995; pp. 238–247.

81. Czumaj, A.; Stemann, V. Randomized Allocation Processes. *Random Struct. Algorithms* **2001**, *18*, 297–331.

82. Berenbrink, P.; Czumaj, A.; Friedetzky, T.; Vvedenskaya, N.D. Infinite parallel job allocations. In Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), Bar Harbor, ME, USA, 9–13 July 2000; pp. 99–108.

83. Stemann, V. Parallel balanced allocations. In Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), Padua, Italy, 24–26 June 1996; pp. 261–269.

84. Mitzenmacher, M. Studying balanced allocations with differential equations. *Comb. Probab. Comput.* **1999**, *8*, 473–482.

85. Mitzenmacher, M.D.; Richa, A.; Sitaraman, R. The power of two random choices: A survey of the techniques and results. In *Handbook of Randomized Computing*; Pardalos, P., Rajasekaran, S., Rolim, J., Eds.; Kluwer Press: London, UK, 2000; pp. 255–305.

86. Broder, A.; Mitzenmacher, M. Using multiple hash functions to improve IP lookups. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2001)*; Full version available as Technical Report TR–03–00; Department of Computer Science, Harvard University: Cambridge, MA, USA, 2000; pp. 1454–1463.

87. Mitzenmacher, M.; Vöcking, B. The asymptotics of Selecting the shortest of two, improved. In Proceedings of the 37th Annual Allerton Conference on Communication, Control, and Computing, Monticello, IL, USA, 22–23 September 1998; pp. 326–327.

88. Wu, J.; Kobbelt, L. Fast mesh decimation by multiple-choice techniques. In Proceedings of the Vision, Modeling, and Visualization, Erlangen, Germany, 20–22 November 2002; pp. 241–248.

89. Siegel, A. On universal classes of extremely random constant time hash functions and their time-space tradeoff. Technical Report TR1995-684, Computer Science Department, New York University, 1995. A previous version appeared under the title "On universal classes of fast high performance hash functions, their time-space tradeoff and their applications. In Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science (FOCS), Triangle Park, NC, USA, 30 October–1 November 1989; pp. 20–25.

90. auf der Heide, F.M.; Scheideler, C.; Stemann, V. Exploiting storage redundancy to speed up randomized shared memory simulations. *Theoretical Computer Science*. **1996**, *162*, 245–281.

91. Schickinger, T.; Steger, A. Simplified witness tree arguments. In Proceedings of the 27th Annual Conference on Current Trends in Theory and Practice of Informatics; LNCS 1963; Springer-Verlag: Berlin/Heidelberg, Germany, 2000; pp. 71–8.

92. Cole, R.; Maggs, B.M.; auf der Heide, F.M.; Mitzenmacher, M.; Richa, A.W.; Schroeder, K.; Sitaraman, R.K.; Voecking, B. Randomized protocols for low-congestion circuit routing in multistage interconnection networks. In Proceedings of the 29th Annual ACM Symposium on the Theory of Computing (STOC), El Paso, TX, USA, 4–6 May 1998; pp. 378–388.

93. Cole, R.; Frieze, A.; Maggs, B.M.; Mitzenmacher, M.; Richa, A.W.; Sitaraman, R.K.; Upfal, E. On balls and bins with deletions. In *Proceedings of the 2nd International Workshop on Randomization and Approximation Techniques in Computer Science*; LNCS 1518; Springer-Verlag: Berlin/Heidelberg, Germany, 1998; pp. 145–158.

94. Swain, S.N.; Subudhi, A. A novel RACH scheme for efficient access in 5G and Beyond betworks using hash function. In Proceedings of the 2022 IEEE Future Networks World Forum (FNWF), Montreal, QC, Canada, 10–14 October 2022; pp. 75–82.

95. Guo, J.; Liu, Z.; Tian, S.; Huang, F.; Li, J.; Li, X.; Igorevich, K.K.; Ma, J. TFL-DT: A trust evaluation scheme for federated learning in digital twin for mobile networks. *IEEE J. Sel. Areas Commun.* **2023**. https://doi.org/10.1109/JSAC.2023.3310094.

96. Okamoto, M. Some inequalities relating to the partial sum of binomial probabilities. *Ann. Math. Stat.* **1958**, *10*, 29–35.

97. Dubhashi, D.; Ranjan, D. Balls and bins: a study in negative dependence. *Random Struct. Algorithms* **1998**, *13*, 99–124.

98. Esary, J.D.; Proschan, F.; Walkup, D.W. Association of random variables, with applications. *Ann. Math. Stat.* **1967**, *38*, 1466–1474.

99. Joag-Dev, K.; Proschan, F. Negative association of random variables, with applications. *Ann. Stat.* **1983**, *11*, 286–295.