TWO-WAY CHAINING WITH REASSIGNMENT

KETAN DALAL, LUC DEVROYE, EBRAHIM MALALLA AND ERIN MCLEISH*

Abstract. We present an algorithm for hashing $\lfloor \alpha n \rfloor$ elements into a table with n separate chains that requires $O(1)$ deterministic worst-case insert time, and $O(1)$ expected worst-case search time for constant α. We exploit the connection between two-way chaining and random graph theory in our techniques.

Key words. hashing, two-way chaining, worst-case search time, random graphs, probabilistic analysis of algorithms

AMS subject classifications. 68Q25, 68M20, 68P10, 60G99, 05C80

1. Introduction. In classical uniform hashing with chaining, a set of s keys are inserted into a hash table with n separate chains (or linked lists) via a uniform hash function. The insertion time is constant, and the average search time is proportional to the load factor of the hash table $\alpha := s/n$. However, even for constant load factor, the worst-case search time (the length of the longest chain) is asymptotic to $\log n/\log\log n$, in probability [18, 27].

Azar et al. [3] suggested a novel approach called the *greedy two-way chaining* paradigm. It uses two independent uniform hash functions to insert the keys where each key is inserted *on-line* into the shorter chain, with ties broken randomly. The insertion time is still constant, while the average search time cannot be more than twice the average search time of classical uniform hashing. However, the expected maximum search time is only $2\log_2\log n + 2\alpha + O(1)$ [3, 4, 24]. The two-way chaining paradigm has been effectively used to derive many efficient algorithms [5, 6, 7]. A further variant of on-line two-way chaining [28] improves the maximum search time by a constant factor.

On the other hand, one can show that the *off-line* version of two-way chaining, where all the hashing values of the keys are known in advance, yields better worst-case performance [3, 8, 25]. Czumaj and Stemann [8] proved that if $s \leq 1.67545943... \times n$, one can find an assignment for the keys such that the maximum chain length is at most 2, w.h.p. (with high probability, i.e., with probability tending to one as $n \to \infty$). In general, for any integer $k \geq 2$, it is known [23] that there is a threshold $c_k \sim k$ such that if $s \leq c_k n$, one can assign the keys such that the maximum chain length is at most k, w.h.p. The insertion time, however, is proportional to s. This shows that there is a large gap between the worst-case performances of the on-line and off-line versions of two-way chaining. One wonders if it is possible to design an efficient on-line two-way chaining algorithm whose worst-case search time is close enough to its off-line one, while preserving constant insertion time and $O(\alpha)$ average search time. Our goal here is to obtain constant expected maximum search and deterministic $O(1)$ insertion times when the load factor of the hash table is constant.

Many hashing schemes that achieve constant worst-case search time have been developed [11, 12, 13, 16]. However, these schemes use a large number of hash functions, sometimes employ rehashing techniques, and have insertion times that are constant

*Research of the authors was supported by NSERC Grant A3456, and NSERC Centre of Excellence IRIS grant "Learning machines," and the National Science Foundation Graduate Research Fellowship. Emails: kdalal@cs.mcgill.ca, luc@cs.mcgill.ca, emal-a@cs.mcgill.ca, and mcleish@cs.mcgill.ca. School of Computer Science, McGill University, Montreal, Canada.

only in an expected amortized sense. The closest to our work is a new hashing scheme called cuckoo hashing [26, 10] which utilizes the two-choice paradigm to improve the worst-case performance, but it relies also on the idea of reallocation of the inserted keys. It inserts n keys into a hash table that is partitioned into two parts, each of size $\lceil (1+\epsilon)n \rceil$, for some constant $\epsilon > 0$. It uses two independent hash functions chosen from an $O(\log n)$-universal class—one function only for each sub-table. Each key is hashed initially by the first function to a cell in the first sub-table. If the cell is full, then the new key is inserted there anyway, and the old key is kicked out to the second sub-table to be hashed by the second function. The same rule is applied in the second sub-table. Keys are moved back and forth until a key moves to an empty location or a limit of $O(\log n)$ moves is reached. When the limit is reached, new independent hash functions are chosen, and the whole table is rehashed. The worst-case search time is at most two, but the insertion time is constant only in an amortized expected sense. An off-line and static version of this algorithm previously appeared in [25].

In this paper, we present a two-way chaining algorithm that is close to cuckoo hashing but it achieves constant worst-case insertion time, deterministically, and constant worst-case search time asymptotically almost surely, when the load factor is constant. The space consumption is also linear. The idea is based on the structure of a random multi-graph, a key reassignment technique, and a deamortization method. The algorithm is divided into stages where at each stage the hash table is modelled by a random graph with n vertices representing the chains and m edges denoting the keys inserted during the stage. Inserting keys into chains corresponds to orienting edges towards vertices. Our goal then is to minimize the maximum out-degree. This model has been used earlier to analyze the off-line version of two-way chaining [8]. When the graph is a forest, it is easy to orient the edges such that the maximum out-degree is one. In order to keep the maximum out-degree as low as possible, some edges need to be reoriented when two trees are joined during the hashing process, and this means that the corresponding keys also need to be reassigned. Furthermore, cycles could occur in the random graph. Since the hashing process is on-line, we use a queue to control the orientation process, thereby ensuring that every insertion operation takes only a constant time of work. This leads us to the elegant deamortization method introduced by Gajewska and Tarjan [17]. In the next section we describe the algorithm precisely, and assure that an insert takes $O(1)$ deterministic worst-case time. We analyze the worst-case search time in Section 3.

2. The Algorithm.

We start by presenting a simplified algorithm that requires $\omega(1)$ insertion time in the worst-case and then using a standard deamortization trick to reduce the insertion time to $O(1)$.

Our algorithm inserts $s = \lfloor \alpha n \rfloor$ keys into a hash table $\mathcal{T}$ with n separate chains (implemented as doubly-linked lists) denoted by $\mathcal{T}[1], \ldots, \mathcal{T}[n]$ by using two independent uniform hash functions f and g. We assume throughout that f and g map the space of the keys to $\{1, \ldots, n\}$ such that if $x_1, \ldots, x_s$ are different keys, then $f(x_1), g(x_1), \ldots, f(x_s), g(x_s)$ are independent and uniformly distributed on $\{1, \ldots, n\}$. So, a key x is inserted into one of the chains $\mathcal{T}[f(x)]$ or $\mathcal{T}[g(x)]$. To search for any key, we only examine the two possible hashing chains available to it. Thus, the worst-case search time is at most twice the length of the longest chain plus the time needed to compute the hashing values. For simplicity, we ignore the time for evaluating the hash functions.

The hashing process is described as follows, (see Figure 1). In addition to the hash table, the algorithm maintains a directed graph $\mathcal{G}(V, A)$, where V is a set of n

vertices and A is a set of arcs. Each vertex of $\mathcal{G}$ corresponds to a chain of $\mathcal{T}$. An arc $\langle u, v\rangle \in A$ implies that there exists some key x whose hash values are u and v. The direction of this arc indicates that x is located in the chain corresponding to u, i.e. $\mathcal{T}[u]$. (With some abuse of notation, we will refer to an edge (u, v) of $\mathcal{G}$ to indicate either $\langle u, v\rangle$ or $\langle v, u\rangle$.) In addition, let `X` be a pointer to the linked-list node that contains x, and let `*X` be the node itself. An important property of the arcs is that they correspond to a *subset* of the keys contained in $\mathcal{T}$, i.e., some keys are *dropped*.

To insert a key x into $\mathcal{T}$, see Pseudocode 1. Notice that initially any key x is always inserted into the chain $\mathcal{T}[f(x)]$. However, during the hashing process the key x may be reassigned back and forth between the two chains $\mathcal{T}[f(x)]$ and $\mathcal{T}[g(x)]$. This could happen by edge reversals as is shown, e.g., in Pseudocode 5.

Pseudocode 1 `Insert(x)`

u $\leftarrow f$(x)
v $\leftarrow g$(x)
Create new linked-list node, `*X` containing key `x`
Insert `*X` into $\mathcal{T}$[u].
`UpdateGraph(u, v, X)`.

The `UpdateGraph` operation enforces that $\mathcal{G}$ is acyclic and that every vertex has out-degree at most one. This means that the graph is simply a forest of parent-pointer trees. We represent the graph in an array where the array element corresponding to a vertex u contains a parent pointer $P[u]$ and a pointer `X`$[u]$ to the linked-list node. Thus the array element for vertex u represents the arc, $\langle u, P[u]\rangle$. The `UpdateGraph` operation is described in Pseudocode 2. Note that in some cases, no edge is inserted into $\mathcal{G}$ at all.

Pseudocode 2 `UpdateGraph(u, v, X)`

$r_1 \leftarrow$ `FindRoot(u)`
$r_2 \leftarrow$ `FindRoot(v)`
if $r_1 \neq r_2$ **then**
  `ReverseRoot(u)`
  `Link(u, v, X)`.
end if

The `FindRoot(u)` operation starts at u and takes parent pointers until the root is found. See Pseudocode 3.

Pseudocode 3 `FindRoot(u)`

$r_1 \leftarrow$ `u`
while `P`[r_1] $\neq$ `nil` **do**
  $r_1 \leftarrow$ `P`[r_1]
end while
return r_1

The `Link(u, v, X)` operation creates the arc $\langle u, v\rangle$ and updates $\mathcal{T}$ accordingly. Recall that the chains of $\mathcal{T}$ are implemented as doubly-linked lists. Thus, the updates to $\mathcal{T}$ can be performed in $O(1)$ time. Additionally, `Link(u, v, X)` requires that u be a root. See Pseudocode 4.

Pseudocode 4 `Link(u, v, X)`

```
P[u] ← v
X[u] ← X
Move *X to 𝒯[u]
```

The `ReverseRoot(u)` function reverses the sequence of pointers from u to the root of u's component. See Pseudocode 5. At the end of this operation, u is the new root of u's component. Note that each reversal will update $\mathcal{T}$ as part of the `Link` operation.

Pseudocode 5 `ReverseRoot(u)`

```
Ensure: P[u] = nil
if P[u] ≠ nil then
  ReverseRoot(P[u])
  Temp ← P[u]
  P[u] ← nil
  Link(Temp, u, X[u])
end if
```

The following two facts are easy to see.

LEMMA 1. *At any point of time, the graph $\mathcal{G}$ is a forest of parent-pointer trees with no self-loops or multiple edges.*

LEMMA 2. *The* `Insert` *operation requires worst-case time not exceeding $4M + O(1)$, where M is the maximum size of any tree in $\mathcal{G}$.*

We use two simple techniques to reduce the cost of the insertions. First, to reduce the size of the trees, after each $m = \lfloor \beta n \rfloor$ inserts, where $\beta < 1/2$ is some constant to be picked later, we destroy $\mathcal{G}$ which amounts to simply zeroing the array representation of $\mathcal{G}$.

Next, following [17], we use a queue $\mathcal{Q}$ (implemented as a linked-list) to defer some work to reduce the cost per insert. We define κ to be a (constant) parameter of the algorithm that indicates the maximum number of operations that may take place as part of each insert to process work items in $\mathcal{Q}$. (The dependence of κ on β will be made clear later on.) The algorithm is modified as follows. After every hash-table insert, where a new list node is created for key x and inserted to the chain $\mathcal{T}[f(x)]$, we append a *graph-insert request* to the *end* of the queue, $\mathcal{Q}$. This is a request for adding the edge $(f(x), g(x))$ to the graph $\mathcal{G}$ and orienting that edge appropriately. Additionally, κ extra units of work will be performed on the request at the front of the queue where one unit of work can be used to traverse or reverse an edge in the graph. Once the request is completed, it is deleted from the queue, and the remaining time is spent on the next element of the queue, until either $\mathcal{Q}$ is empty or the κ available time steps are depleted.

To combine both techniques, we keep an extra graph structure $\mathcal{G}'$ that is zeroed incrementally as elements are inserted into $\mathcal{G}$. After $\lfloor \beta n \rfloor$ inserts, we will simply swap $\mathcal{G}$ and $\mathcal{G}'$. This allows us to reduce the cost of zeroing the graph structure every $\lfloor \beta n \rfloor$ inserts. There are some minor technicalities in implementing this approach and the complete pseudocode for `ConstantInsert` is given in Pseudocode 6 in the appendix. Essentially, the approach is to break down the operation of `UpdateGraph` into constant time pieces.

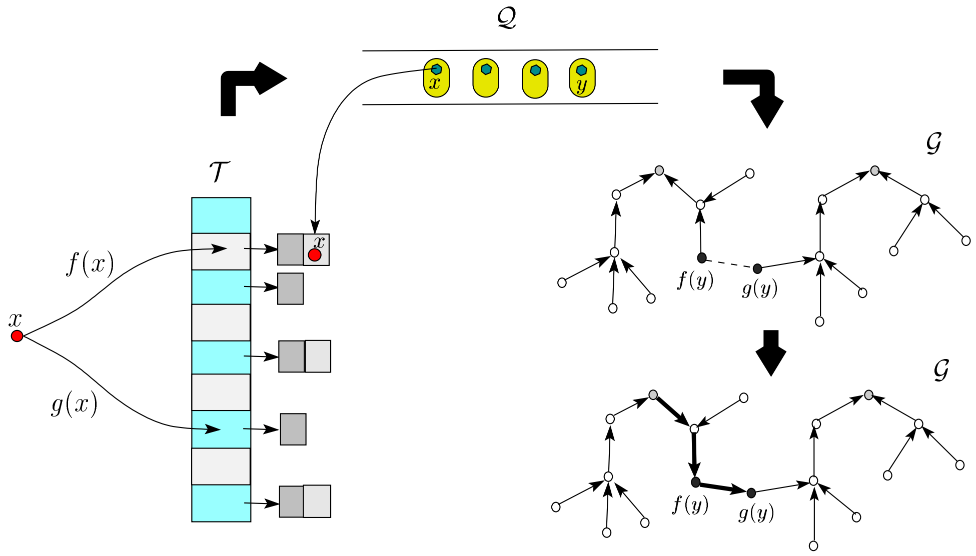

FIG. 2.1. *Upon arrival of key x, it is inserted into $T[f(x)]$, and then a request for adding a corresponding arc is appended to the queue $\mathcal{Q}$. Next, κ extra units of work are performed to process the requests at the front of the queue. The figure also illustrates the* `ReverseRoot` *operation and the insertion of $\langle f(y), g(y)\rangle$.*

We will write $\text{HASH}(n, s, m, \kappa)$ to refer to this modified process of hashing s keys into a hash table of size n where m keys are inserted in each stage and κ is the constant parameter mentioned above. We omit the details for initializing the data structures and keeping track of an edge counter.

LEMMA 3. *Using the queue, the* `ConstantInsert` *operation requires time proportional to κ.*

The deferral described above creates a potential inconsistency between the state of the hash-table and the state of the graph. Specifically, at the completion of a particular insertion request, the graph represents the state of the hash-table at an earlier point in time, i.e., prior to the requests that are still pending in the queue. Only when the queue is empty will the graph represent the state of the hash table. Additionally, because some requests are dropped and on occasion the graph is destroyed, the graph may only contain a subset of the state of the hash-table.

3. The Worst-case Search Time. We shall prove the following theorem.

THEOREM 1. *There is a constant $\kappa > 0$ such that at any point of time during the hashing process $\text{HASH}(n, s, m, \kappa)$, where $n, s, m \in \mathbb{N}$, and $m = \lfloor \beta n \rfloor \le s = O(n \log n)$, for some constant $\beta < 1/2$, the maximum search time is at most $2\lceil s/m \rceil + 6$, w.h.p.*

The theorem confirms that if the load factor of the hash table $s/n = O(1)$, then asymptotically almost surely the maximum search time is constant. Since there is a trivial lower bound of $2s/n$, we see that we are roughly within $1/\beta$ of the best possible, recalling that β can be picked arbitrarily close to $1/2$. Before we proceed with the proof, we need some facts. We write $\text{Bin}(n, p)$ to denote a binomial random variable with parameters $n \in \mathbb{N}$ and $p \in [0, 1]$. We recall the following binomial tail inequalities.

LEMMA 4 (Angluin and Valiant [2]). *For $n \in \mathbb{N}$, $p \in [0, 1]$, and constant $\epsilon \in (0, 1)$,* *we have*

$$\mathbb{P}\left\{\mathrm{Bin}(n,p) \geq (1+\epsilon)np\right\} \leq e^{-np\epsilon^2/3},$$

and

$$\mathbb{P}\left\{\mathrm{Bin}(n,p) \leq (1-\epsilon)np\right\} \leq e^{-np\epsilon^2/2}.$$

Let $\mathbb{G}(n,m)$ denote a random graph with n vertices and m multiedges that may include loops where each edge connects two vertices chosen—one after another—independently and uniformly at random, with replacement, from the set of all n vertices. This means that any loop is realized with probability of $1/n^2$ and any undirected non-loop edge is realized with probability of $2/n^2$. Recall that the classical model $G(n,p)$ of Erdös and Rényi [14, 15], has no loops or multiedges and each edge is realized with a fixed probability $p \in (0,1)$. Throughout, we write $[n]$ to denote the set $\{1,\dots,n\}$.

LEMMA 5. *Let $\mathcal{C}(u)$ be the number of vertices in the connected component containing a fixed vertex u from the random graph $\mathbb{G}(n,m)$, where $n \in \mathbb{N}$, and $m = \lfloor \beta n \rfloor$, for some constant $\beta < 1/2$. Then for any $t \in [n]$, we have $\mathbb{P}\{\mathcal{C}(u) > t\} \leq 2e^{-\gamma t}$, where $\gamma = (1/2-\beta)^2/(2+4\beta)$. Thus, if M is the size of the largest connected component, then $\mathbb{P}\left\{M > (1+\epsilon)\gamma^{-1}\log n\right\} \leq 2n^{-\epsilon}$, for any fixed $\epsilon > 0$.*

Proof. We first need to distinguish between the components of the classical $G(n,p)$ and those of $\mathbb{G}(n,m)$. Let $\mathcal{R}_p(u)$ denote the number of vertices in the component containing vertex u in $G(n,p)$ and use $C_k(u)$ for our model $\mathbb{G}(n,k)$. Next let $|G(n,p)|$ be the number of edges in $G(n,p)$. Notice that

$$\mathbb{P}\left\{\mathcal{R}_p(u) > t \mid |G(n,p)| = k\right\} \geq \mathbb{P}\left\{C_k(u) > t\right\},$$

because conditional on having k edges, components in $G(n,p)$ are stochastically larger than those in $\mathbb{G}(n,k)$, since the latter includes multiedges and loops. This leads to the following relationship between $\mathcal{R}_p(u)$ and $C_m(t)$:

$$\begin{aligned}
\mathbb{P}\left\{\mathcal{R}_p(u) > t\right\} &= \sum_k \mathbb{P}\left\{\mathcal{R}_p(u) > t \mid |G(n,p)| = k\right\}\mathbb{P}\left\{|G(n,p)| = k\right\} \\
&\geq \sum_k \mathbb{P}\left\{C_k(u) > t\right\}\mathbb{P}\left\{|G(n,p)| = k\right\} \\
&\geq \sum_{k\geq m} \mathbb{P}\left\{C_k(u) > t\right\}\mathbb{P}\left\{|G(n,p)| = k\right\} \\
&\geq \sum_{k\geq m} \mathbb{P}\left\{C_m(u) > t\right\}\mathbb{P}\left\{|G(n,p)| = k\right\} \\
&= \mathbb{P}\left\{C_m(u) > t\right\}\mathbb{P}\left\{|G(n,p)| \geq m\right\} \\
&\geq \mathbb{P}\left\{C_m(u) > t\right\} - \mathbb{P}\left\{|G(n,p)| < m\right\}.
\end{aligned}$$

Thus,

$$\mathbb{P}\left\{C_m(u) > t\right\} \leq \mathbb{P}\left\{\mathcal{R}_p(u) > t\right\} + \mathbb{P}\left\{|G(n,p)| < m\right\}.$$

Bounding $\mathbb{P}\left\{\mathcal{R}_p(u) > t\right\}$ in the classical model is done in the usual manner, see for example Janson [21]. Imagine that u's component grows out from vertex u, picking up neighbors according to a binomial distribution. This certainly overestimates the number of vertices in u's component. Now suppose that the component containing vertex u contains more than t vertices. This implies that the sum of t binomial random variables is at least t. Denote these random variables by X_i, for $i = 1..t$, and for an upper bound on $|\mathcal{R}_p(u)|$ assume that each X_i is distributed as $\mathrm{Bin}(n,p)$ and that they are independent. The sum of t independent $\mathrm{Bin}(n,p)$ is itself a $\mathrm{Bin}(nt,p)$ random variable. So we have that

$$\mathbb{P}\left\{|\mathcal{R}_p(u)| > t\right\} \le \mathbb{P}\left\{\mathrm{Bin}(nt,p) \ge t\right\}.$$

Using Lemma 4, with $p = (\beta + 1/2)/n$ and $\epsilon = \frac{1/2-\beta}{1/2+\beta} \in (0,1)$, we get

$$\mathbb{P}\left\{\mathrm{Bin}(nt,(\beta+1/2)/n) \ge t\right\} \le \exp\left(\frac{-t(1/2-\beta)^2}{3(1/2+\beta)}\right).$$

Now it only remains to bound $\mathbb{P}\left\{|G(n,p)| < m\right\}$. Notice that $|G(n,p)|$ is distributed as $\mathrm{Bin}(N,p)$, where $N = \binom{n}{2}$, and

$$\begin{aligned} m - 1 \le \beta n - 1 &= (n-1)(\beta/2 + 1/4) - (n/4 - \beta n/2 + 1 - \beta/2 - 1/4) \\ &\le Np - x, \end{aligned}$$

where $x = (1/4 - \beta/2)n$. Using the lower tail bound of Lemma 4, we get

$$\begin{aligned} \mathbb{P}\left\{|G(n,p)| < m\right\} = \mathbb{P}\left\{\mathrm{Bin}(N,p) \le m-1\right\} &\le \mathbb{P}\left\{\mathrm{Bin}(N,p) \le Np - x\right\} \\ &\le \exp\left(\frac{-x^2}{2Np}\right) = \exp\left(\frac{-n^2(1/2-\beta)^2}{4(n-1)(1/2+\beta)}\right) \\ &\le \exp\left(\frac{-n(1/2-\beta)^2}{4(1/2+\beta)}\right). \end{aligned}$$

Putting everything together the resulting bound for the component size in our model is

$$\mathbb{P}\left\{C(u) > t\right\} \le \exp\left(\frac{-n(1/2-\beta)^2}{4(1/2+\beta)}\right) + \exp\left(\frac{-t(1/2-\beta)^2}{3(1/2+\beta)}\right).$$

Since the component size $t \le n$,

$$\mathbb{P}\left\{C(u) > t\right\} \le 2\exp\left(\frac{-t(1/2-\beta)^2}{4(1/2+\beta)}\right) = 2e^{-\gamma t},$$

where $\gamma = (1/2-\beta)^2/(2+4\beta)$. ◻

The next lemma, which is included to make the paper self-contained, assures us that the asymptotic structure of $\mathbb{G}(n,m)$ is not complex when $m < n/2$. Further details can be found in [20]. An edge is said to *complete* a cycle if both of its vertices are chosen from the same connected component before its insertion.

LEMMA 6. *Let $n \in \mathbb{N}$, and $m = \lfloor \beta n \rfloor$, for some constant $\beta < 1/2$. In the random graph $\mathbb{G}(n,m)$, the expected number of edges that complete cycles is $O(\log n)$. Furthermore, the probability that $\mathbb{G}(n,m)$ contains a connected component with more than one cycle is $o(1/\log n)$.*

Proof. Let Y_i be the number of edges that complete cycles in $\mathbb{G}(n,m)$ after $(i-1)$ edges have been inserted. Let D_i be the event that the i-th edge completes a cycle. Let

M_i be the random variable corresponding to the size of the largest component after $i-1$ edges have been inserted. Using the fact that the sequence $\{M_i\}$ is increasing,

$$\begin{aligned}
\mathbf{E}\,[\,Y_{m+1}\,] &= \sum_{i=1}^{m} \mathbb{P}\,\{D_i\} \\
&= \mathbf{E}\left[\sum_{i=1}^{m} \mathbb{I}_{[D_i]}\right] = \mathbf{E}\left[\mathbf{E}\left[\sum \mathbb{I}_{[D_i]} \mid M_i\right]\right] \\
&\le \sum_{i=1}^{m} \mathbf{E}\,[\,M_i/n\,] \le \mathbf{E}\,[\,M_{m+1}\,] \\
&\le \frac{2}{\gamma}\log n + m\mathbb{P}\left\{M_{m+1} > \frac{2}{\gamma}\log n\right\} \\
&= O(\log n)\,,
\end{aligned}$$

which follows from Lemma 5. Next, we show that it is unlikely for a component to contain more than one cycle.

Let A_i be the event that $M_i \le a\log n$ where a is chosen such that $\mathbb{P}\left\{A^c_{m+1}\right\} = O(1/n)$. Let B_i be the event that $Y_i \le \log^3 n$. Using $\mathbf{E}\,[\,Y_{m+1}\,] = O(\log n)$ and Markov's inequality, we have $\mathbb{P}\left\{B^c_{m+1}\right\} = O(1/\log^2 n)$. Let C_i be the event that the i-th edge causes the creation of a component that contains two cycles. Equivalently, C_i is the event that the i-th edge connects two (not necessarily distinct) cyclic components. Treating these events as sets, we obtain

$$\begin{aligned}
C_i &= (C_i \cap A_i \cap B_i) \cup (C_i \cap B_i^c \cap A_i) \cup (C_i \cap A_i^c) \\
&\subseteq (C_i \cap A_i \cap B_i) \cup B_i^c \cup A_i^c\,.
\end{aligned}$$

Since A_i^c and B_i^c are increasing events, then $\cup_{i=1}^{m+1} A_i^c = A^c_{m+1}$, and similarly, $\cup_{i=1}^{m+1} B_i^c = B^c_{m+1}$. Consequently,

$$\begin{aligned}
\mathbb{P}\left\{\bigcup_i C_i\right\} &\le \left(\sum_{i=1}^{m} \mathbb{P}\,\{C_i, A_i, B_i\}\right) + \mathbb{P}\left\{B^c_{m+1}\right\} + \mathbb{P}\left\{A^c_{m+1}\right\} \\
&\le \left(\sum_{i=1}^{m} \mathbb{P}\,\{C_i \mid A_i, B_i\}\right) + O(1/\log^2 n) + O(1/n) \\
&\le m\left(\frac{(a\log n)(\log^3 n)}{n}\right)^2 + O\left(1/\log^2 n\right) = O\left(1/\log^2 n\right)\,,
\end{aligned}$$

as the maximum number of 'bad' vertices that the i-th edge can choose from is at most Y_i times M_i which is not more than $a\log^4 n$. □

Recall that the hashing process $\textsc{Hash}(n, s, m, \kappa)$ is divided into $N := \lceil\, s/m \,\rceil$ different stages where at each stage $m\,\lfloor\,\beta n\,\rfloor$ keys are inserted into the hash table. Consider only the first stage. Recall that the graph $\mathcal{G}$ does not fully represent the hash table because first, the keys are inserted into the hash table without any delay, while the edges are enqueued in $\mathcal{Q}$ for what might be a long time before they are actually inserted into the graph $\mathcal{G}$, and secondly, any edge that completes a cycle is not added to the graph. For convenience, we will write $\mathcal{G}(m)$ to denote the graph $\mathcal{G}$ at the end of the first stage, i.e., after having fully processed m edges, and $\mathcal{G}(m)^+$ to denote the complete graph of $\mathcal{G}(m)$ plus all dropped edges that complete cycles.

Observe that the undirected version of the graph $\mathcal{G}(m)^+$ is stochastically equivalent to the random graph $\mathbb{G}(n, m)$. The following lemma shows that the dropped edges of the whole hashing process are disjoint. For any multiset of edges $\mathcal{E}$, and for any vertex u in the graph, let $\mathcal{V}(u, \mathcal{E})$ be the multiset of all vertices v such that $(u, v) \in \mathcal{E}$. Let $\deg(u, \mathcal{E}) = |\mathcal{V}(u, \mathcal{E})|$ be the degree of u in $\mathcal{E}$.

LEMMA 7. *Let $\mathcal{D}$ be the multiset of dropped edges during all stages of the hashing process* $\mathrm{HASH}(n, s, m, \kappa)$*, where n, s and m are as defined in Theorem 1. Then* $\max_u \deg(u, \mathcal{D}) = 1$*, w.h.p.*

Proof. Recall that the number of stages is $N := \lceil s/m \rceil = O(\log n)$. For $i = 1, \ldots, N$, let $\mathcal{D}_i$ be the multiset of all dropped edges in stage i. Since the dropped edges are the ones that complete cycles, then Lemma 6 implies that $\mathbf{E}[\,|\mathcal{D}_1|\,] = O(\log n)$, and

$$\mathbb{P}\left\{\max_u \deg(u, \mathcal{D}_1) > 1\right\} = o(1/\log n),$$

because $\deg(u, \mathcal{D}_1) > 1$ implies that the component containing u has more than one cycle. Clearly, $\sum_u \deg(u, \mathcal{D}_1) \le 2\,|\mathcal{D}_1|$. Since we have n vertices in the graph, then

$$\mathbf{E}[\,\deg(u, \mathcal{D}_1)\,] = \mathbf{E}[\,\mathbf{E}[\,\deg(u, \mathcal{D}_1) \mid |\mathcal{D}_1|\,]\,] \le 2\mathbf{E}[\,|\mathcal{D}_1|\,]/n.$$

For $i \ne j$, let $A_{i,j}$ be the event that there is a vertex u appearing in two dropped edges in $\mathcal{D}_i$ and $\mathcal{D}_j$, i.e., there are vertices v and w such that $(u, v) \in \mathcal{D}_i$, and $(u, w) \in \mathcal{D}_j$. Since $\mathcal{D}_1, \ldots, \mathcal{D}_N$ are independent and identically distributed, thence,

$$\begin{aligned}
\mathbb{P}\left\{\max_u \deg(u, \mathcal{D}) > 1\right\} &\le N\mathbb{P}\left\{\max_u \deg(u, \mathcal{D}_1) > 1\right\} + \binom{N}{2}\mathbb{P}\{A_{1,2}\} \\
&\le o(1) + N^2 n\left(\mathbb{P}\{\deg(u, \mathcal{D}_1) \ge 1\}\right)^2 \\
&\le o(1) + N^2 n\left(\mathbf{E}[\,\deg(u, \mathcal{D}_1)\,]\right)^2 \\
&\le o(1) + N^2 n\left(\frac{2\mathbf{E}[\,|\mathcal{D}_1|\,]}{n}\right)^2 \\
&= o(1) + O((\log n)^4/n) = o(1).
\end{aligned}$$

□

Finally, we recall the following inequality.

LEMMA 8 (Hoeffding [19]). *Let S be a set of m balls where ball i has a value x_i. Let $X_1, \ldots, X_\nu$ be the values of ν balls chosen from S independently and uniformly at random without replacement. Let $Y_1, \ldots, Y_\nu$ be the values of ν balls chosen from S independently and uniformly at random with replacement. Then for any continuous convex function f, we have*

$$\mathbf{E}\left[f\left(\sum_{i=1}^{\nu} X_i\right)\right] \le \mathbf{E}\left[f\left(\sum_{i=1}^{\nu} Y_i\right)\right].$$

Proof of Theorem 1. Recall that $\mathcal{G}(m)$ denotes the graph $\mathcal{G}$ at the end of the first stage, and $\mathcal{G}(m)^+$ denotes the complete graph of $\mathcal{G}(m)$ plus all dropped edges that complete cycles. First of all, Lemma 7 says that the vertices of all dropped edges of the whole hashing process $\mathrm{HASH}(n, s, m, \kappa)$ are distinct, w.h.p. That is, the corresponding keys of these edges are inserted into distinct chains, or in other words, any chain harbors at most one key that corresponds to a dropped edge. Therefore, upon termination of the hashing process, the dropped edges may contribute at most one to the maximum chain length.

We shall prove that w.h.p., during any interval of time (measured with respect to the number keys inserted) of length $\nu := \lfloor n^{1/4} \rfloor$ (at any stage), the queue $\mathcal{Q}$ must be empty at least once, and no connected component is chosen more than twice. This means that w.h.p., any set of requests that exist in the queue at some point of time could have at most two requests for inserting edges in the same connected component, that is, we could have at most two keys that are inserted into the same chain before the final positions of the related keys are corrected. Assume this is true for the time being. Then clearly the length of any chain in the hash table during the first stage is not more than the out-degree of the corresponding vertex in $\mathcal{G}$ plus three: one for any possible dropped edge contribution, and two for the two requests in the queue that have chosen the same component. However, the maximum out-degree of $\mathcal{G}$ is ensured to be one all the time. Hence, the maximum chain length at any point of time during the first stage is at most 4, w.h.p. Since we follow the same strategy at each stage, it is not difficult to see that the chain length increases by at most one per stage. Note that over all stages, each chain has at most one dropped edge contribution and a single component is present at most twice in the queue. Consequently, the maximum chain length at any point of time during the hashing process is at most $\lceil s/m \rceil + 3$, w.h.p., and hence the worst-case search time is at most $2\lceil s/m \rceil + 6$, w.h.p., as we have to search two chains for each key.

Now we prove our assumption. We say "at time i" to mean at the insertion time of the i-th key. Thus, an interval of time of length ν means a period of time into which exactly ν keys are inserted. Let M be the size of the largest connected component in $\mathcal{G}(m)$. Since $\beta < 1/2$, then by Lemma 5, $\mathbb{P}\left\{M > 2\gamma^{-1}\log n\right\} = O(1/n)$.

Let A_{ij} be the event that the j-th component is hit by the i-th request and is hit at least twice more in the subsequent $\nu - 1$ requests. Note that when the insertion request at the front of the queue is being processed, the j-th component is fixed. Let $A = \cup_{i,j} A_{ij}$. Let M_{ij} be the number of vertices in the j-th component just before the i-th insertion. Thus $\mathbb{P}\left\{\exists\, i, j, M_{ij} > 2\gamma^{-1}\log n\right\} \le \mathbb{P}\left\{M > 2\gamma^{-1}\log n\right\} = O(1/n)$. Since we have fewer than m time intervals of length ν during the first stage, and at most n connected components in $\mathcal{G}$, then the binomial tail inequality yields that

$$
\begin{aligned}
\mathbb{P}\{A\} &\le \mathbb{P}\left\{A \cap \left[\, M \le 2\gamma^{-1}\log n \,\right]\right\} + \mathbb{P}\left\{M > 2\gamma^{-1}\log n\right\} \\
&\le mn \max_{i,j} \mathbb{P}\left\{A_{ij} \cap \left[\, M \le 2\gamma^{-1}\log n \,\right]\right\} + O(1/n) \\
&\le mn \max_{i,j} \mathbb{P}\left\{A_{ij} \cap \left[\, M_{ij} \le 2\gamma^{-1}\log n \,\right]\right\} + O(1/n) \\
&\le mn \max_{i,j} \mathbb{P}\left\{A_{ij} \mid \left[\, M_{ij} \le 2\gamma^{-1}\log n \,\right]\right\} + O(1/n).
\end{aligned}
$$

The probability that the i-th request hits the j-th component is at most $2M_{ij}/n$, since there are 2 vertices involved in the request. The number of times that the j-the component is hit in the subsequent $\nu - 1$ requests is distributed as $\mathrm{Bin}(2\nu - 2, M_{ij}/n)$.

Thus, using $\mathbb{P}\{\mathrm{Bin}(n,p) \geq 2\} \leq (np)^2/2$, we have

$$\begin{aligned}
\mathbb{P}\{A\} &\leq mn\left(\frac{4\log n}{\gamma n}\right)\mathbb{P}\left\{\mathrm{Bin}(2\nu, 4\gamma^{-1}\log n/n) \geq 2\right\} + O(1/n)\\
&\leq mn\left(\frac{4\log n}{\gamma n}\right)\left(\frac{8\nu\log n}{\gamma n}\right)^2 + O(1/n)\\
&= O\left(\frac{\log^3 n}{n^{1/2}}\right).
\end{aligned}$$

The event A refers to the first stage only. Multiplying the bound by the number of stages $N = O(\log n)$, we can see that the probability that during some stage there is a connected component which is chosen more than twice in some interval of time of length ν goes to zero as $n \to \infty$.

Next, let $[a,b] = \{a, a+1, \ldots, b\}$ where $|b-a+1| = \nu$ be a time interval of length ν, and let B be the event that $[a,b]$ is the first interval of time of length ν in which the queue $\mathcal{Q}$ is never empty. Observe that if B is true, then the queue was empty at time $a-1$. Let $e_a, \ldots, e_b$ be the edges associated with the ν requests appended to $\mathcal{Q}$ during $[a,b]$. For $i \in [a,b]$, let T_i be the computational time needed for the algorithm to fully process the edge e_i, that is, the number of edges traversed or reversed during the whole process of serving the request, plus one if the edge is inserted. Let R_i be the number of vertices in the connected component to which the edge e_i belongs. Thus, $T_i \leq 4R_i$, for all $i \in [a,b]$. Using Chernoff's bounding method (see e.g., [19]), we see that for parameter $\lambda > 0$,

$$\begin{aligned}
\mathbb{P}\{B\} &\leq \mathbb{P}\{T_a > \kappa, T_a + T_{a+1} > 2\kappa, \ldots, T_a + \cdots + T_b > \kappa\nu\}\\
&\leq \mathbb{P}\{T_a + \cdots + T_b > \kappa\nu\}\\
&\leq \mathbb{P}\{R_a + \cdots + R_b > \kappa\nu/4\}\\
&\leq e^{-\lambda\kappa\,\nu/4}\,\mathbf{E}\left[\exp\left(\sum_{i=a}^{b}\lambda R_i\right)\right].
\end{aligned}$$

Recall that $\mathcal{G}(m)^+$ denotes the union of $\mathcal{G}(m)$ and the edges that completed cycles. Suppose that we choose ν edges from the set of all m edges in the graph $\mathcal{G}(m)^+$ independently and uniformly without replacement. Let $V_1, \ldots, V_\nu$ be the sizes of the components containing these edges. Notice that $V_1, \ldots, V_\nu$ stochastically dominate $R_a, \ldots, R_b$. On the other hand, suppose that we repeat the experiment of choosing ν edges from the set of all m edges in the graphs $\mathcal{G}(m)^+$ independently and uniformly but *with* replacement. Let $V_1^*, \ldots, V_\nu^*$ be the values of these edges which are plainly independent and identically distributed. Thence, by Lemma 8, we see that

$$\begin{aligned}
\mathbb{P}\{B\} &\leq e^{-\lambda\kappa\,\nu/4}\,\mathbf{E}\left[\exp\left(\sum_{i=1}^{\nu}\lambda V_i\right)\right]\\
&\leq e^{-\lambda\kappa\,\nu/4}\,\mathbf{E}\left[\exp\left(\sum_{i=1}^{\nu}\lambda V_i^*\right)\right]\\
&= e^{-\lambda\kappa\,\nu/4}\left(\mathbf{E}\left[e^{\lambda V_1^*}\right]\right)^\nu.
\end{aligned}$$

By definition, V_1^* is distributed as the size of the component containing a uniformly chosen edge of $\mathbb{G}(n,m)$. In distribution, this is the same as the size of the component containing the *last* edge inserted in $\mathbb{G}(n,m)$, which is in turn stochastically dominated by $C(u)+C(v)$, where u and v are uniformly chosen vertices and $C(u)$ is the size of the component containing u in $\mathbb{G}(n,m-1)$. Using Lemma 5 for $t \in [m]$,

$$\mathbb{P}\{V_1^* \geq t\} \leq \mathbb{P}\{C(u)+C(v) \geq t\} \leq 2\,\mathbb{P}\{C(u) \geq t/2\} \leq 2\left(2\,e^{-\gamma(t/2-1)}\right),$$

where $\gamma = \frac{(1/2-\beta)^2}{2+4\beta} > 0$. Therefore,

$$\mathbf{E}\left[e^{\lambda V_1^*}\right] \leq 4\sum_{t=1}^{m} e^{\lambda t} e^{-\gamma(t/2-1)} \leq 4\,e^{\lambda+\gamma/2}\sum_{t=0}^{\infty} e^{(\lambda-\gamma/2)t} \leq \frac{4\,e^{\lambda+\gamma/2}}{1-e^{\lambda-\gamma/2}},$$

provided that $\lambda < \gamma/2$. Finally, we get

$$\mathbb{P}\{B\} \leq e^{-\lambda\kappa\,\nu/4}\left(\mathbf{E}\left[e^{\lambda V_1^*}\right]\right)^{\nu} \leq e^{-p\nu},$$

where $p := \lambda\kappa/4 - \lambda - \gamma/2 + \log\left(1-e^{\lambda-\gamma/2}\right) - \log 4$, which is positive if we choose $\kappa > 4 + 4\left(-\gamma/2 - \log(1-e^{\lambda-\gamma/2}) + \log 4\right)/\lambda$. For example, if we put $\lambda = \gamma/4$, then $\kappa = \lceil -40\log(1-e^{-\gamma})/\gamma \rceil$ will suffice. With this choice, and since there are $N = \lceil s/m \rceil$ stages, and in each stage there are at most m intervals of time of length ν, we see that the probability that at some stage there is an interval of time of length ν in which the queue is never empty is at most $Nme^{-p\nu} = o(1)$. Indeed, this is true even if the interval is as short as $2\log_p s = O(\log n)$. The proof now is complete. $\square$

Some Remarks.

1. The last step of the proof reveals that asymptotically almost surely the space consumed by the queue is indeed $O(\log n)$, because the queue is always empty at least once during any interval of $O(\log n)$ insertions.
2. Notice that as β approaches $1/2$, γ goes to zero, and hence, the constant κ increases to infinity.
3. As β increases to $1/2$, the worst-case search time of our algorithm is close to $4\alpha+6$, w.h.p., where $\alpha := s/n$ is the load factor of hash table. This performance can be beaten by other hashing algorithms when α is large enough. For example, when $\alpha \geq \log n/\log\log n$, the classical hashing with chaining where only a single uniform hash function is utilized achieves better performance than our algorithm: the maximum search time is known [27] to be at most $\alpha + \Theta(\sqrt{\alpha\log n})$, w.h.p. Also, when $\log_2\log n \leq \alpha \leq (1/3)\log n/\log\log n$, the worst-case search time of greedy two-way chaining, which is $2\log_2\log n + 2\alpha + O(1)$, w.h.p., is the best known [4]. However, for $\alpha < \log_2\log n$, our algorithm has the best worst-case search time among all known hashing with chaining algorithms that have constant worst-case insertion time.
4. We did not try to optimize the constant insertion time κ, or the total space consumed by the algorithm. We believe that some aspects of the algorithm can be modified to improve its performance by a constant factor. For example, the use of the graph $\mathcal{G}$ is probably unnecessary, and one can implement the operations on the graph directly on the hash table. It is also necessary to generalize the algorithm for the dynamic case where deletions are allowed.

Appendix. In this appendix, we give the detailed pseudocode for the procedure `ConstantInsert`. Each element of the queue is the 5-tuple [`X, Root1, Root2, IndexFrom, IndexTo`] defined as follows:

`X`	The pointer to the linked-list node that is currently in some chain of $\mathcal{T}$.
`Root1`	Current computation of the root of vertex $f(x)$'s tree.
`Root2`	Current computation of the root of vertex $g(x)$'s tree.
`IndexFrom`	If `Root1` and `Root2` are roots, then the new arc should point from `IndexFrom`
`IndexTo`	If `Root1` and `Root2` are roots, then the new arc should point to `IndexTo`

The last two fields are computable from each other; so only four elements are actually needed. Additionally, recall that `X` points to a structure that contains the key which we will refer to as `X.key`.

Pseudocode 6 `ConstantInsert(x,` $\mathcal{G}$`,` $\mathcal{T}$`)`

```
Create new linked-list node, *X containing key x
Insert *X into T[f(x)]
Create new request, R_new = [X, f(x), g(x), f(x), g(x)]
Append R_new to Q
Zero ⌈1/β⌉ more elements of G'
i ← 1
repeat
  i ← i + 1, R ← Q.Peek()
  if P[R.Root1] ≠ nil then
    R.Root1 = P[R.Root1]
  else if P[R.Root2] ≠ nil then
    R.Root2 = P[R.Root2]
  else if R.Root1 = R.Root2 then
    Q.Pop()
  else
    Y ← X[R.IndexFrom]
    Link(R.IndexFrom, R.IndexTo, R.X)
    if Y = nil then
      Q.Pop() {This will happen once for every new arc in G.}
      if |G| = ⌊βn⌋ then swap(G, G')
    else if f(Y.key) = R.IndexFrom then
      R ← [Y, R.Root1, R.Root2, g(Y.key), f(Y.key)]
    else
      R ← [Y, R.Root1, R.Root2, f(Y.key), g(Y.key)]
    end if
  end if
until i = κ OR Q = ∅
```

REFERENCES

[1] N. Alon and J. H. Spencer, *The Probabilistic Method*, 2nd ed., John Wiley, New York, 2000.

[2] D. Angluin and L. G. Valiant, "Fast probabilistic algorithms for hamiltonian paths and matchings," *Journal of Computer and Systems Science*, vol. 18, pp. 155–193, 1979.

[3] Y. Azar, A. Z. Broder, A. R. Karlin and E. Upfal, "Balanced allocations," *SIAM Journal on Computing*, vol. 29 (1), pp. 180–200, 2000. A preliminary version of this paper appeared in *Proceedings of the 26th ACM Symposium on Theory of Computing (STOC)*, pp. 593–602, 1994.

[4] P. Berenbrink, A. Czumaj, A. Steger, and B. Vöcking, "Balanced allocations: the heavily loaded case," in: *Proceedings of the 32nd ACM Symposium on Theory of Computing (STOC)*, pp. 745–754, 2000.

[5] A. Broder and M. Mitzenmacher, "Using multiple hash functions to improve IP lookups," in: *Proceedings of the IEEE INFOCOM 2001 Conference*, Anchorage, Alaska USA , April 2001. Full version available as Technical Report TR–03–00, Department of Computer Science, Harvard University, Cambridge, MA, 2000.

[6] J. Byers, J. Considine, and M. Mitzenmacher, "Simple load balancing for distributed hash tables," in: *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems*, pp. 80–87, 2003.

[7] A. Czumaj, F. Meyer auf der Heide, and V. Stemann, "Contention resolution in hashing based shared memory simulations," *SIAM Journal on Computing*, vol. 29, No. 5, pp. 1703–1739, 2000.

[8] A. Czumaj and V. Stemann, "Randomized allocation processes," *Random Structures and Algorithms*, vol. 18, Issue 4, pp. 297–331, June 2001.

[9] L. Devroye, "Branching processes and their applications in the analysis of tree structures and tree algorithms," in: *Probabilistic Methods for Algorithmic Discrete Mathematics*, ed. M. Habib, C. McDiarmid, J. Ramirez-Alfonsin and B. Reed, pp. 249–314, 1998.

[10] L. Devroye and P. Morin, "Cuckoo hashing: further analysis," *Information Processing Letters*, vol. 86, pp. 215–219, 2004.

[11] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. Tarjan, "Dynamic perfect hashing: upper and lower bounds," *SIAM Journal on Computing*, vol. 23 (4), pp. 738–761, 1994. A preliminary version appeared in: *Proceedings of the 29th IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 524–531, 1988.

[12] M. Dietzfelbinger and F. Meyer auf der Heide, "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, pp. 6–19, 1990.

[13] M. Dietzfelbinger and F. Meyer auf der Heide, "High performance universal hashing, with applications to shared memory simulations," in: *Data Structures and Efficient Algorithms*, LNCS 594, Springer-Verlag, pp. 250–269, 1992.

[14] P. Erdös, "Some remarks on the theory of graphs," *Bulletin of the American Mathematical Society*, vol. 53, pp. 292–294, 1947.

[15] P. Erdös and A. Rényi, "On the evolution of random graphs," *Publ. Math. Ins. Hunger. Acad. Sci.*, Vol. 5, PP. 17-61, 1960.

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

[17] H. Gajewska and R. E. Tarjan, "Deques with heap order," *Information Processing Letters*, vol. 22(4), pp. 197–200, 1986.

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

[19] W. Hoeffding, "Probability inequalities for sums of bounded random variables," *Journal of the American Statistical Association*, vol. 58, pp. 13–30, 1963.

[20] S. Janson, D. E. Knuth, T. Łuczak, and B. Pittel, "The birth of the giant component," *Random Structures and Algorithms*, vol. 4 (3), pp. 233–358, 1993.

[21] S. Janson, T. Łuczak and A. Ruciński, *Random Graphs*, John Wiley & Sons, New York, 2000.

[22] R. M. Karp, "The transitive closure of a random digraph," *Random Structures and Algorithms*, vol. 1, PP. 73-93, 1990.

[23] E. Malalla, *Two-way Hashing with Separate Chaining and Linear Probing*, Ph.D. thesis, School of Computer Science, McGill University, 2004.

[24] M. Mitzenmacher, A. Richa, and R. Sitaraman, "The power of two random choices: A survey of the techniques and results," in: *Handbook of Randomized Computing*, (P. Pardalos, S. Rajasekaran, and J. Rolim, eds.), pp. 255–305, 2000.

[25] R. Pagh, "On the cell probe complexity of membership and perfect hashing," in: *Proceedings*

of 33rd ACM Symposium on Theory of Computing (STOC), pp. 425–432, 2001.

[26] R. Pagh and F. F. Rodler, "Cuckoo hashing," in: *Proceedings of the European Symposium on Algorithms*, LNCS 2161, Springer-Verlag, pp. 121–133, 2001. A previous version is available as BRICS Report Series RS–01–32, Department of Computer Science, University of Aarhus, 2001.

[27] M. Raab and A. Steger, ""Balls into bins"—a simple and tight analysis," in: *Proceedings of the 2nd Workshop on Randomization and Approximation Techniques in Computer Science*, vol. 1518, Lecture Notes in Computer Science, Springer-Verlag, pp. 159–170, 1998.

[28] B. Vöcking, "How asymmetry helps load balancing," in: *Proceedings of the 40th IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 131–141, 1999.