# Chapter 8

# Dictionaries & Hashing

You manage a library and want to be able to quickly tell whether you carry a given book or not. We need the capability to *insert*, *delete*, and *search* books.

**Definition 8.1** (Dictionary)**.** *A **dictionary** is a data structure that manages a set of **objects**. Each object is uniquely identified by its **key**. The relevant operations are*

- *search: find an object with a given key*
- *insert: put an object into the set*
- *delete: remove an object from the set*

**Remarks:**

- There are alternative names for dictionary, e.g. key-value store, associative array, map, or just set.
- If the dictionary only offers `search`, it is called *static*; if it also offers `insert` and `delete`, it is *dynamic*.
- For our purposes, we will ignore that we actually have a set of objects, each of which is identified by a unique key, and just talk about the set of keys. With regard to the library example, books are globally uniquely identified by a key called `ISBN`. Whenever we say we insert/delete/search a key, we can just drag the key's object along.
- The classic data structure for dictionaries is a binary search tree.

## 8.1 Search Trees

**Definition 8.2** (Binary search tree)**.** *A **binary search tree** is a rooted tree (Definition 2.7), where each node stores a key. Additionally, each node may have a pointer to a left and/or a right child tree. For all nodes, if existing, the nodes in the left child tree store smaller keys, and those in the right child tree store larger keys.*





**Algorithm 8.3** Search Tree: Search

**Input :** key $k$, root $r$ of search tree
**Output:** $k$ if it is in the tree, else $\perp$

If $r$ contains key $k$, return $k$
If $k$ is smaller than the key of $r$, set $r$ to left child and recurse
If $k$ is larger than the key of $r$, set $r$ to right child and recurse
Return $\perp$

**Remarks:**

- The symbol $\perp$ ("bottom") signifies `null` or `undefined`.
- The cost of searching in a binary search tree is proportional to the *depth* of the key, which is the distance (Theorem 2.15) between the node with the key and the root.
- There are search trees called *splay trees* that keep frequently searched keys close to the root. There may be keys with linear depth in a splay tree, but on average the cost of a search is logarithmic in the number of keys.
- Using *balanced search trees*, we can maintain a dictionary with worst-case logarithmic depth for all keys, and thus worst-case logarithmic cost per insert/delete/search operation.
- Is there a way to build a dictionary with less than logarithmic cost and with keys that cannot be ordered?

## 8.2 Hashing

**Definition 8.4** (Universe, Key Set, Hash Table, Buckets)**.** *We consider a **universe** $U$ containing all possible keys. We want to maintain a subset of this universe, the **key set** $N \subseteq U$ with $|N| =: n$, where $|N| \ll |U|$. We will use a **hash table** $M$, i.e. an array $M$ with $m$ **buckets** $M[0], M[1], \ldots, M[m-1]$.*

**Remarks:**

- The standard library of almost every widely used programming language provides hash tables, sometimes by another name. In C++, they are called `unordered_map`, in Python `dictionary`, in Java `HashMap`.
- The translation from virtual memory to physical memory uses a piece of hardware called *translation lookaside buffer* (TLB), which is a hardware implementation of a hash table. It has a fixed size and acts like a cache for frequently looked up virtual addresses.
- Compilers make use of hash tables to manage the symbol table.

**Definition 8.5** (Hash Function)**.** *Given a universe $U$ and a hash table $M$, a **hash function** is a function $h : U \to M$. Given some key $k \in U$, we call $h(k)$ the **hash** of $k$.*

**Remarks:**

- A hash function should be efficiently computable, e.g. $h(k) = k \bmod m$ for a key $k \in \mathbb{N}$.
- If we use `ISBN` mod $m$ as our library hash function, can we insert/delete/search books in constant time?!
- What if two keys $k \neq l$ have $h(k) = h(l)$?

**Definition 8.6** (Collision)**.** *Given a hash function $h : U \to M$, two distinct keys $k, l \in U$ produce a* ***collision*** *if $h(k) = h(l)$.*

**Remarks:**

- There are competing objectives we want to optimize for when hashing. On the one hand, we want to make the hash table small since we want to save memory. On the other hand, small tables will have more collisions. How likely is it to get a collision for a given $n$ and $m$?

**Theorem 8.7** (Birthday Problem)**.** *If we throw a fair $m$-sided dice $n \leq m$ times, let $D$ be the event that all throws show different numbers. Then $D$ satisfies*

$$\Pr[D] \leq \exp\left(-\frac{n(n-1)}{2m}\right).$$

*Proof.* We have that

$$\Pr[D] = \frac{m}{m} \cdot \frac{m-1}{m} \cdot \ldots \cdot \frac{m-(n-1)}{m} = \prod_{i=0}^{n-1} \frac{m-i}{m}$$
$$= \prod_{i=0}^{n-1}\left(1 - \frac{i}{m}\right) = \exp\left(\sum_{i=0}^{n-1} \ln\left(1 - \frac{i}{m}\right)\right)$$

We can use that $\ln(1+x) \leq x$ for all $x > -1$ and the monotonicity of $e^x$:

$$\Pr[D] = \exp\left(\sum_{i=0}^{n-1} \ln\left(1 - \frac{i}{m}\right)\right) \leq \exp\left(\sum_{i=0}^{n-1} -\frac{i}{m}\right) = \exp\left(-\frac{n(n-1)}{2m}\right)$$

□

**Remarks:**

- Theorem 8.7 is called the "birthday problem" since traditionally, people use birthdays for illustration: In order to have a chance of at least 50% that two people in a group share a birthday, we only need 23 people.
- If we insert more than roughly $n \approx \sqrt{m}$ keys into a hash table, the probability of a collision approaches 1 quickly. In other words, unless we are willing to use at least $m \approx n^2$ space for our hash table, we will need a good strategy for resolving collisions.



- Theorem 8.7 assumes totally random hash functions — for non-random distributions of hashes, we might have more collisions. In particular, if we fix a hash function, then we can always end up with a key set $N$ that suffers from many collisions. E.g., if many books have an `ISBN` that ends in 000, then `ISBN` mod 1,000 is a terrible hash function.
- Maybe we can use modulo, but with a different $m$?
- However, for any hash function there are bad key sets.
- On the other hand, for every key set there are good hash functions! How do we efficiently pick a good hash function, i.e. one that is likely to distribute hashes well?

**Definition 8.8** (Universal Family)**.** *Let $\mathcal{H} \subseteq \{h : U \to M\}$ be a family of hash functions from $U$ to $M$. If for all pairs of distinct keys $k \neq l \in U$, the probability of a collision is $\Pr[h(k) = h(l)] \leq \frac{1}{m}$ when we choose $h \in \mathcal{H}$ uniformly, then $\mathcal{H}$ is called a* ***universal family (of hash functions)****.*

**Remarks:**

- In other words: if we choose a hash function from a universal family, we can expect the hashes to be distributed well, regardless of the key set.
- We cannot just pick a random function from $U$ to $M$ because there are $|M|^{|U|}$ many, so we need $|U| \log |M|$ bits to encode such a random function. That is even more bits than keys in our huge universe $U$.

**Lemma 8.9.** *For prime $m$, $a \in \{0, \ldots, m-1\}$, $\delta \in \{1, \ldots, b-1\}$, and $b \leq m$, the linear function $f_\delta(a) := a \cdot \delta \bmod m$ is a bijection.*

*Proof.* Suppose there are integers $a, a'$ with $0 < a < a' < m$ such that $f_\delta(a) = f_\delta(a') \bmod m$. Then, there exists an integer $k$ such that $a' \cdot \delta = a \cdot \delta + km \Leftrightarrow \delta(a' - a) = km$, which implies that $m \mid \delta(a' - a)$. Since $\delta$ is relative prime to $m$, we have $m \mid (a' - a)$, thus $a = a' \bmod m \Rightarrow a = a'$. □

**Theorem 8.10** (Universal Hashing)**.** *Let $m$ be prime and $s \in \mathbb{N}$. Let $U = \{0, \ldots, b-1\}^s$ and let $M = \{0, \ldots, m-1\}$ with $b \leq m$. For a key $k = (k_0, \ldots, k_{s-1}) \in U$ and coefficient tuple $a = (a_0, \ldots, a_{s-1}) \in \{0, \ldots, m-1\}^s$, define*

$$h_a(k_0, \ldots, k_{s-1}) = \sum_{i=0}^{s-1} a_i \cdot k_i \bmod m.$$

*Then $\mathcal{H} := \{h_a : a \in \{0, \ldots, m-1\}^s\}$ is a universal family of hash functions.*

*Proof.* Let $(k_0, \ldots, k_{s-1}) = k \neq k' = (k'_0, \ldots, k'_{s-1}) \in U$. Using $\delta_i = k'_i - k_i$ we get

$$h_a(k') - h_a(k) = \sum_{i=0}^{s-1} a_i \cdot k'_i - \sum_{i=0}^{s-1} a_i \cdot k_i = \sum_{i=0}^{s-1} a_i \cdot \delta_i \qquad \bmod m.$$

The terms with $\delta_i = 0$ are 0, and so we can ignore them. Let $X := \{i \in \{0, \ldots, m-1\} : \delta_i \neq 0\}$ be the non-empty set of the indices of the non-zero terms, with $x = |X|$. There are $m^x$ possibilities to choose the coefficients $\{a_i : i \in X\}$. So how many of these possibilities will have a conflict, i.e., a sum of 0? If we choose $x-1$ coefficients arbitrarily, then the last coefficient can always render the sum 0, as according to Lemma 8.9 the last coefficient can add any value $\{0, \dots, m-1\}$ to the sum. In other words, $m^{x-1}$ sums are 0. Therefore, our chance of randomly picking an $a$ that produces a collision is $\frac{m^{x-1}}{m^x} = \frac{1}{m}$. □

**Remarks:**

- Theorem 8.10 gives us a general method for picking hash functions from a universal family in an efficient manner. We simply choose a prime number $m$ and uniformly at random some factors $a_0, \dots, a_r$. Thus, we can represent our hash function as the tuple $(m, a_0, \dots, a_r)$.
- In practice, hash tables perform really well, and if we detect that we had bad luck in choosing our hash function, we just choose a new one and rebuild our table with the new function — this is called *rehashing*.

## 8.3 Static Hashing

How can we state the tradeoff between space and collisions more precisely?

**Definition 8.11** (Number of Collisions)**.** *Given a hash function $h : U \to M$ and a key set $N \subseteq U$, define the number of collisions that $h$ produces on $N$ as*

$$C(h, N) := |\{\{k, l\} \subseteq N : k \neq l, h(k) = h(l)\}|.$$

**Lemma 8.12** (Space vs. Collisions)**.** *Let $b$ be an upper bound on the number of collisions we want a hash function $h_b$ to produce on a given key set $N$ of size $|N| = n$. If we sample from a universal family, we can find an $h_b$ that satisfies $C(h_b, N) < b$ and $m = \lceil \frac{n(n-1)}{b} \rceil$ by sampling a constant number of times in expectation.*

*Proof.* There are $\binom{n}{2}$ pairs of distinct keys in $N$, and each of those produces a collision with probability at most $1/m$ since $h$ is chosen from a universal family. Together, using the linearity of expectation we get

$$\mathbb{E}[C(h, N)] \le \binom{n}{2} \cdot \frac{1}{m} = \frac{n(n-1)}{2m}.$$

The Markov inequality states that for any random variable $X$ that only takes on non-negative integer values, we have $\Pr[X \ge k \cdot \mathbb{E}[X]] \le \frac{1}{k}$. Hence,

$$\Pr[C(h, N) \ge 2 \cdot \mathbb{E}[C(h, N)]] \le \frac{1}{2}$$

and so

$$\Pr[C(h, N) < 2 \cdot \mathbb{E}[C(h, N)]] \ge \frac{1}{2}$$

If we choose $m$ such that $2 \cdot \mathbb{E}[C(h, N)] \le b$, then we only need to sample 2 hash functions in expectation. Solving for $m$, we get

$$2 \cdot \mathbb{E}[C(h, N)] \le b \Leftarrow \frac{n(n-1)}{m} \le b \Leftrightarrow \frac{n(n-1)}{b} \le m.$$

□



**Remarks:**

- According to Theorem 8.12, if we want no collisions, we set $b = 1$ and choose $m = \lceil \frac{n(n-1)}{1} \rceil = n(n-1)$.
- Similarly, if we can tolerate $n$ collisions, we find that a hash table of size $m = n - 1$ suffices.
- Algorithm 8.13 defines perfect static hashing, which applies the result of Theorem 8.12.

**Algorithm 8.13** Perfect Static Hashing

**Input** : fixed set of keys $N$
**Output** : Primary hash table $M$ and secondary hash tables $M_i$
**Function:** $N_i := \{k \in N : h(k) = i\}$
**Function:** $n_i := |N_i|$

```
 1: M := a table with n buckets
 2: repeat
 3:    h := hash function N → M (sampled from universal family)
 4: until C(h, N) < n
 5: for i ∈ M do
 6:    M_i := a table with n_i(n_i − 1) = 2(n_i choose 2) buckets
 7:    repeat
 8:       h_i := hash function N_i → M_i (sampled from universal family)
 9:    until C(h_i, N_i) < 1
10: end for
11: return (M, h, (M_i)_{i∈{0,...,m−1}}, (h_i)_{i∈{0,...,m−1}})
```

**Remarks:**

- In a first stage (Lines 1 to 4), we find a hash function $h$ with at most $n$ collisions in linear space according to Theorem 8.12.
- In a second stage (Lines 5 to 10), we find a hash function $h_i$ per bucket $i$ without collisions by using an amount of space that is quadratic in the number of keys in the bucket $n_i$ as per Theorem 8.12.

**Theorem 8.14** (Perfect Static Hashing)**.** *Algorithm 8.13 returns a collision-free data structure of with a total size ($M$ and all $M_i$) less than $3n$.*

*Proof.* The data structure is collision-free because of Line 9. Bucket $i$ is of size $n_i(n_i - 1) = 2\binom{n_i}{2}$, holds $n_i$ keys that produced $\binom{n_i}{2}$ collisions. Since the total number of collisions is less than $n$ (Line 4), and each bucket is of size twice its collisions, the total size of all buckets is less than $2n$. Because of Line 1, the size of $M$ is $n$, so together the total size of the data structure is less than $n + 2n = 3n$. □

**Remarks:**

- We now have a hashing algorithm that can be built in linear space and expected linear time, and offers worst-case constant time search for a static set $N$.
- But what about a dynamic dictionary?

## 8.4 Collisions

**Definition 8.15** (Hashing with Chaining)**.** *In* ***hashing with chaining****, every bucket $M[i]$ stores a pointer to a secondary data structure that manages all keys $k$ with $h(k) = i$. Insertion, search, and deletion of $k$ are all relegated to those data structures. In the simplest implementation, we use a simple linked list for each bucket.*

**Remarks:**

- Algorithm 8.13 is an instance of hashing with chaining with the $M_i$ being the secondary data structures managing the buckets.
- The Java standard library uses hashing with chaining to resolve collisions.

**Definition 8.16** (Load Factor)**.** *The fraction $\frac{n}{m} =: \alpha$ is called the* ***load factor*** *of the hash table.*

**Remarks:**

- The performance of all three operations (insert/delete/search) depends on the load factor for all collision resolution strategies.
- Hashing with chaining allows for a load factor $\alpha > 1$ since the size of the table is the number of secondary data structures; performance deteriorates with growing $\alpha$.
- If we use linked lists as secondary structures and use a hash function chosen from a universal family, the cost for an unsuccessful search is $1+\alpha$ in expectation, while that for a successful search is roughly $1+\frac{\alpha}{2}$ in expectation.
- If we use one of the strategies of this section and $\alpha$ grows too large, we should rehash with a bigger $m$ in order to remain efficient. In the Java standard library, if a hash table surpasses a load factor of 0.75, it is rehashed into a hash table with twice the size of the old one.

**Definition 8.17** (Hashing with Probing)**.** *In* ***hashing with probing****, keys are stored directly in the hash table. The sequence $(h_i(k) \bmod m)_{i\geq 0}$ is called the* ***probing sequence*** *of $k$, and each step of the iteration is a* ***probe****.*

**Remarks:**

- Algorithm 8.18 defines how to search for a key in hashing with probing. Line 5 is a *successful search*, and Lines 7 and 11 are the two cases of *unsuccessful searches*.
- To insert a key, we adapt Algorithm 8.18: with an unsuccessful search in Line 7 we insert in the empty bucket. Therefore, the cost of an insert is roughly the cost of an unsuccessful search. An unsuccessful search in Line 11 triggers a rehash.



**Algorithm 8.18** Hashing with Probing: Search

**Input** : key $k$ to search for
**Output** : key $k$ if found, else $\perp$
**Function:** parametrized hash function $h_i$

```
1: i := 0
2: while i < m do
3:    j := h_i(k) mod m
4:    if M[j] = k then
5:       return M[j]
6:    else if M[j] = ⊥ then
7:       return ⊥
8:    end if
9:    i := i + 1
10: end while
11: return ⊥
```

- Table 8.19 describes three different types of hashing with probing, each together with the approximate time that a successful or unsuccessful search takes in expectation. More generally, linear probing uses some linear function $h_i(k) = h(k)+ci$ for some $c \neq 0$, and quadratic probing uses some quadratic function $h_i(k) = h(k) + ci + di^2$ with $d \neq 0$. As long as we guarantee that $h_i(k)$ is integer for all $i \in [m]$, the constants $c$ and $d$ can be rational.

| Type | $h_i(k)$ | $\approx$ cost successful | $\approx$ cost unsuccessful |
|---|---|---|---|
| Linear probing | $h(k) + i$ | $\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$ | $\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$ |
| Quadratic probing | $h(k) + i^2$ | $\frac{1}{1-\alpha} + \ln\frac{1}{1-\alpha} - \alpha$ | $1 + \ln\frac{1}{1-\alpha} - \frac{\alpha}{2}$ |
| Double hashing | $h_1(k) + i \cdot h_2(k)$ | $\frac{1}{1-\alpha}$ | $\frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$ |

Table 8.19: Different types of hashing with probing together with the expected number of probes per search, where $\alpha$ is the load factor of the table. For hashing with probing, we need $\alpha \leq 1$ since we must have $n \leq m$. Each of $h, h_1, h_2$ is a hash function drawn from a universal family.

**Remarks:**

- What is the reason for the different costs in Table 8.19?
- Linear probing suffers from so-called *primary clustering*: the probing sequences are simplistic; if two probing sequences meet in the same bucket $j$, they will continue to collide in buckets $j+1, j+2, \ldots$
- Quadratic probing does not suffer from primary clustering, but it is subject to *secondary clustering*: if two keys have the same hash, then their probing sequences are identical.
- The form of quadratic probing defined in Table 8.19 has one additional issue: the probing sequence of a key does not necessarily cover the

whole table. Assume $m = 7$ buckets and $h(k) = 0$, then the probing sequence of $k$ is $(0, 1, 4, 2, 2, 4, 1)$ — buckets $3, 5, 6$ do not appear.

- Double hashing does not suffer from either version of clustering. One can show that if the hash functions $h_1$ and $h_2$ used in double hashing are independently drawn from a universal family, then double hashing performs as well as an idealized hash function that assigns hashes uniformly at random.

## 8.5 Worst Case Guarantees

So far, the cost of all operations for dynamic key sets has been given in expected time cost. There are algorithms that allow us to do better and give us worst case guarantees on some of the operations. Two widely known possibilities to achieve this are called *dynamic perfect hashing* and *cuckoo hashing*.

**Algorithm 8.20** Cuckoo Hashing: Insert

1: Cuckoo hashing uses two hash tables, and as such provides two possible buckets $M_1[h_1(k)]$ or $M_2[h_2(k)]$ for each key $k$.
2: When inserting a new key $k$, and one of the two buckets is empty, we simply place $k$ there.
3: If both buckets are occupied by other keys, the new key is anyway inserted in one of its two possible buckets, "kicking out" the key $k'$ that currently resides in this bucket.
4: The kicked out key $k'$ moves to its other bucket, potentially kicking out the currently resident key $k''$; this process is repeated recursively until an empty bucket is found.
5: If this recursion loops or takes too long (logarithmic in the table size), the hash table is rebuilt using two new hash functions.

**Remarks:**

- Search and delete only need to check two buckets to figure out whether a given key is in the table, and so those operations are worst case constant time.
- One can show that the expected insert cost in *cuckoo hashing* is constant as long as the load factor $\alpha$ is below 0.5.
- Cuckoo hashing gets its name from cuckoo birds: they lay their eggs into the nests of other birds, and once the cuckoo chicks hatch, they push the other eggs/chicks out of the nest.
- The idea behind cuckoo hashing is to use the *"power of two choices"*, which can be roughly described as: if you can choose between two resources and use the one that is less busy, you gain efficiency.
- To adapt perfect static hashing to a dynamic setting where we can also handle inserts and deletions, all we have to do is choose the size of $M_i$ twice as large as in Algorithm 8.13, and rehash appropriately: Whenever $C(h_i, N_i) > 0$ for some bucket $i$, we rehash that bucket until there are no collisions. Once some bucket reaches $n_i^2 \approx |M_i|$ due to insertions, we rehash the entire table. This leaves us with expected constant time insert and delete, and worst case constant time search. To keep the table linear-sized, we rehash everything after every $m$ updates (inserts or deletes).



## Chapter Notes

Dictionaries based on search trees are useful for providing additional operations such as nearest neighbor queries or range queries, where we want to find all keys in a certain range. Binary search trees were first published by three independent groups in 1960 and 1962 (for references, see Knuth [9]). The first instance of a self-balancing search tree that guarantees logarithmic cost for insert/search/delete is the AVL-tree, named so after its inventors Adelson-Velski and Landis [1]. For multidimensional keys, e.g. geometric data or images, there are specialized tree structures such as kd-trees [2] or BK-trees [3].

Hashing has a long history and was initially used and validated based on empirical results. One of the first publications was Peterson's 1957 article [11] where he defined an idealized version of probing and empirically analyzed linear probing. Universal hashing was introduced two decades later by Carter and Wegman in 1979 [4]. Perfect static hashing was invented in 1984 by Fredman et al. [7] and is sometimes also referred to as FKS hashing after its inventors. Its dynamization by Dietzfelbinger et al. took another decade until 1994 [6]. A comprehensive study on perfect hashing by Czech et al. was compiled in 1997 [5]. Cuckoo hashing is a comparatively recent algorithm; it was introduced by Pagh and Rodler in 2001 [10].

There have been a number of other developments regarding hashing since the late 1970s; for an overview, see Knuth [9], in particular the section on History at the end of chapter 6.4. For a neat visualization of hashing with probing, see [8] online.

The power of two choices paradigm has found widespread application and analysis in load balancing scenarios. It was initially studied from the perspective of a balls-into-bins game where we want to minimize the maximum number of balls in any bin, and to do this we can pick two random bins and put the next ball into the least full of the two bins. Richa et al. [12] compiled an excellent survey on the earliest sources and numerous applications of this paradigm.

This chapter was written in collaboration with Georg Bachmeier.

## Bibliography

[1] M Adelson-Velskii and Evgenii Mikhailovich Landis. An Algorithm for the Organization of Information. *Doklady Akademii Nauk USSR*, 146(2):263–266, 1962.

[2] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[3] W. A. Burkhard and R. M. Keller. Some approaches to best-match file searching. *Commun. ACM*, 16(4):230–236, 1973.

[4] J.Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143 – 154, 1979.

[5] Zbigniew J. Czech, George Havas, and Bohdan S. Majewski. Perfect hashing. *Theoretical Computer Science*, 182(1 - 2):1 – 143, 1997.

[6] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994.

[7] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, 1984.

[8] David Galles. Closed hashing. `https://www.cs.usfca.edu/~galles/visualization/ClosedHash.html`. Accessed: 2017-03-29.

[9] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[10] Rasmus Pagh and Flemming Friche Rodler. *Algorithms — ESA 2001: 9th Annual European Symposium Århus, Denmark, August 28–31, 2001 Proceedings*, chapter Cuckoo Hashing, pages 121–133. Springer Berlin Heidelberg, 2001.

[11] W. W. Peterson. Addressing for random-access storage. *IBM J. Res. Dev.*, 1(2):130–146, 1957.

[12] Andrea W Richa, M Mitzenmacher, and R Sitaraman. The power of two random choices: A survey of techniques and results. *Combinatorial Optimization*, 9:255–304, 2001.