Chapter 6

Dictionaries

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 6.1 (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, or map.
- 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 via encapsulation.
- The classic data structure for dictionaries is a binary search tree.

6.1 Search Trees

Definition 6.2 (Binary search tree)**.** *A **binary search tree** is a rooted tree (Definition 1.7), where each node stores a key. Additionally, each node may have a pointer to a left and/or a right child node. For all nodes, if existing, the left child stores a smaller key, and the right stores a larger key.*

Algorithm 6.3 Search Tree Search

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

1: If r contains key k, return k
2: If k is smaller than the key of r, set r to left child and recurse
3: If k is larger than the key of r, set r to right child and recurse
4: 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 (Definition 1.15) 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?

6.2 Hashing

Definition 6.4 (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 **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 `dict`, 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 6.5 (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. $h(k) = k \mod m$ 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?!
- But what if two keys k and l have $h(k) = h(l)$?

Definition 6.6 (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 with regard to the size of a hash table. 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 many collisions will we get for a given n and m?

Theorem 6.7 (Birthday Problem)**.** *If we throw a fair m-sided dice n 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

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

We can use $\ln(1+x) \leq x$ for all $x > -1$:

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

□

Remarks:

- Theorem 6.7 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 6.7 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? In particular, we might apply a simple function to the `ISBN` first to introduce some randomness, then use a moderately large prime number for m since primes are less likely to cause collisions?
- 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 evenly?

Definition 6.8 (Universal Hashing)**.** *Let $\mathcal{H} \subseteq \{h : U \to M\}$ be a family of hash functions from U to M. $\mathcal{H}$ is called* ***universal*** *if for any two distinct keys $k, l \in U$, if we choose $h \in \mathcal{H}$ uniformly, then the probability of a collision is* $\Pr[h(k) = h(l)] = \frac{1}{m}$.

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.

Theorem 6.9 (Universal Hashing)**.** *Let m be prime and $r \in \mathbb{N}$. For $U = [b]^{r+1}$ where $[b] = \{0, \dots, b-1\}$ and $M = [m]$ with $b \leq m$ and $a = (a_0, \dots, a_r) \in [m]^{r+1}$, define*

$$h_a(k_0, \dots, k_r) = \sum_{i=0}^{r} a_i \cdot k_i \mod m.$$

Then $\mathcal{H} := \{h_a : a \in [m]^{r+1}\}$ is a universal family of hash functions.

Proof. Let $(k_0, \dots, k_r) = k \neq l = (l_0, \dots, l_r) \in U$. Since k and l are different, there must be a smallest index $0 \leq j \leq r$ such that $k_j \neq l_j$. For a given $a \in [m]^{r+1}$, consider

$$\begin{aligned}
h_a(k) = h_a(l) \Leftrightarrow \quad & \sum_{i=0}^{r} a_i \cdot k_i \equiv \sum_{i=0}^{r} a_i \cdot l_i && \mod m \\
\Leftrightarrow \quad & a_j(k_j - l_j) \equiv \sum_{i \neq j} a_i \cdot (l_i - k_i) && \mod m \\
\Leftrightarrow \quad & a_j \equiv (k_j - l_j)^{-1} \cdot \sum_{i \neq j} a_i \cdot (l_i - k_i) && \mod m
\end{aligned}$$

where $(k_j - l_j)^{-1}$ exists because $k_j - l_j \neq 0$. There are m^r choices for all a_i with $i \neq j$; for each of those choices, there is a unique a_j for the hashes to be equal since $[m]$ is a field, and in fields linear equations have unique solutions. Since there are a total of m^{r+1} choices for $(a_0, \ldots, a_r)$, this gives us a probability of $\frac{m^r}{m^{r+1}} = \frac{1}{m}$ for the hashes to be equal. $\square$

Remarks:

- Theorem 6.9 gives us a general method for constructing universal hash functions in an efficient manner. We simply choose a prime number m and uniformly at random some factors $a_0, \ldots, a_r$. Thus, we can represent our hash function as the tuple $(m, a_0, \ldots, a_r)$.

- 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*.

- In Java, creating an `int` as the hash of an `Object` is the job of the JVM. In OpenJDK for example, the first time `hashCode()` is called on an `Object`, the JVM creates a random number as its hash and stores it with the `Object`.

- Hash functions are usually defined on classes, not by the hashing structures themselves. For classes in the Java standard library that have fields (e.g., `String`s have a `char[]` as a field), `hashCode()` is implemented such that the hash is derived from the fields that are considered when deciding whether one instance `equals()` another. This is called the contract between `hashcode()` and `equals()`: if two instances of the same class are equal, then they have to have the same hash. On the other hand, two objects with the same hash need not be equal.

- In Theorem 6.9 we assume that $U = [m]^{r+1}$. In applications, we often want to find hashes for keys that are not numbers, and keys of different "sizes", e.g. `String`s of different lengths.

- The Java standard library uses a fixed version of a weaker form of this type of hashing for `String`. Instead of choosing $(a_0, \ldots, a_r) \in [m]^{r+1}$, Java fixes a value $a_0 \in$ `int` and uses $(a_0^0, a_0^1, .., a_0^r)$ instead, where r is the number of characters in the `String`. In Java, $a_0 = 31$ was chosen since it produced comparatively few collisions on English language test data. Also, this hash function can be represented as a single value a_0, regardless of how long the strings we want to hash are, and it will also work to manage `String`s with different lengths in the same hash table.

6.3 Static Hashing

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

Definition 6.10 (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 6.11 (Space vs. Collisions)**.** *If we uniformly sample a hash function from a universal family, given an upper bound b on the number of collisions, we only need to sample a constant number of times in expectation to find a hash function h_b that satisfies $C(h_b, N) < b$ in $m = \lceil \frac{n(n-1)}{b} \rceil$ space for a fixed key set N of size n.*

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

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

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

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

and so

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

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

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

□

Remarks:

- According to Lemma 6.11, 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 6.12 defines perfect static hashing, which applies the result of Lemma 6.11.

Algorithm 6.12 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: $M :=$ hash table with n buckets
2: **repeat**
3: $\quad h :=$ hash function $N \to M$
4: **until** $C(h, N) < n$
5: **for** $i \in M$ **do**
6: $\quad M_i :=$ hash table with $2\binom{n_i}{2} = n_i(n_i - 1)$ buckets
7: $\quad$ **repeat**
8: $\quad\quad h_i :=$ hash function $N_i \to M_i$
9: $\quad$ **until** $C(h_i, N_i) < 1$
10: **end for**
11: **return** $(M, h, (M_i)_{i\in[m]}, (h_i)_{i\in[m]})$

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 Lemma 6.11.
- 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 Lemma 6.11.

Theorem 6.13 (Perfect Static Hashing)**.** *When Algorithm 6.12 returns, the size of M and all M_i together is less than $3n$.*

Proof. Due to Line 1, the size of M is exactly n.

The number of collisions produced by the keys in bucket i is $\binom{n_i}{2}$ since any two of them produce one. We know that $2\binom{n_i}{2} = n_i(n_i - 1)$. As two keys in different buckets cannot produce a collision, we can sum the number of collisions per bucket over all buckets to get the number of all collisions, and so

$$\sum_{i=0}^{m-1} n_i(n_i - 1) = \sum_{i=0}^{m-1} 2\binom{n_i}{2} = 2\sum_{i=0}^{m-1}\binom{n_i}{2} = 2C(h, N) < 2n.$$

We used that $C(h, N) < n$ due to Line 4. Because of the choice of the size of M_i in Line 6, all buckets M_i together use less than $2n$ space. In total, M and all M_i together have a size of 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?

6.4 Collisions

Definition 6.14 (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 can use linked lists.*

Remarks:

- Algorithm 6.12 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.
- From Java 7 to Java 8, the standard library changed from `HashMap` always using a linked list for a bucket to using a linked list as long as the bucket contains less than a certain number of keys, and building a search tree once the bucket reaches that number.
- More concretely: `HashMap` applies its own hash function to the hash supplied by the keys (remember, each class defines `hashCode()`, either by overriding it or by inheriting it from `Object`) to determine each key's bucket. For the ordering within the trees, there are two possibilities: the class implements `Comparable` or it does not.
- If the class of the keys implements `Comparable`, then the natural ordering of the keys is used.
- If the keys are not `Comparable`, then the tree uses the values returned by `System.identityHashCode(Object x)` to order keys; this method returns the same value that the default implementation of `Object.hashCode()` returns. This means that if your class is not `Comparable` and does not override `hashCode()`, then `System.identityHashCode(Object x)` is equal for all keys within a given tree; this makes the trees degenerate to lists.

Definition 6.15 (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 discussed in this section.
- 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 α.
- 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 successul search is roughly $1+\frac{\alpha}{2}$ in expectation.

- If we use one of the strategies of this section and α grows too large, we should rehash with a bigger m in order to maintain expected constant time cost. 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 6.16 (Hashing with Probing)**.** *Algorithm 6.17 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 an* ***unsuccessful search****. We call the sequence* $(h_i(k) \mod m)_{i\geq 0}$ *the* ***probing sequence*** *of* k*, and each step of the iteration a* ***probe****.*

Algorithm 6.17 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: $\quad j := h_i(k) \mod m$
4: $\quad$ **if** $M[j] = k$ **then**
5: $\qquad$ **return** $M[j]$
6: $\quad$ **else if** $M[j] = \perp$ **then**
7: $\qquad$ **return** $\perp$
8: $\quad$ **end if**
9: $\quad i := i + 1$
10: **end while**
11: **return** $\perp$

Remarks:

- To insert a key, we adapt Algorithm 6.17: 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.

- Table 6.18 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 6.18: Different types of hashing with probing together with the expected number of probes per search. α is the load factor of the table, and for hashing with probing, it has to satisfy $\alpha < 1$ since we cannot store more keys in the table than it has buckets. Each of h, h_1, h_2 is a hash function drawn from a universal family.

Remarks:

- The main reason for the differences in access times is *clustering*.
- Linear probing suffers from *primary clustering*: from some point on, the probing sequences of any two keys will become identical.
- 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 will still be identical.
- The form of quadratic probing defined in Table 6.18 has one additional issue: the probing sequence of a key does not necessarily cover the whole table. Assume the size of the table is $m = 7$ 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.

6.5 Worst Case Guarantees

So far, the cost of all operations 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*.

Remarks:

- 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 6.12, 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

Algorithm 6.19 Cuckoo Hashing Insert

Input : key $k \in U$ we want to insert; counter `limit` specifying the maximum number of tries

Data Structures: arrays M_1, M_2 of equal size

Functions : hash functions $h_1 : U \to M_1, h_2 : U \to M_2$; chosen independently and uniformly at random from universal families

if $M_1[h_1(k)] = k$ or $M_2[h_2(k)] = k$ **then**
    return
end if
$t := 1$
while $t \leq$ `limit` **do**
    swap k with $M_1[h_1(k)]$
    if $k = \perp$ **then**
        return
    end if
    swap k with $M_2[h_2(k)]$
    if $k = \perp$ **then**
        return
    end if
    $t := t + 1$
end while
`rehash()`
`CuckooHashingInsert(k, limit)`

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).

- Another option is *cuckoo hashing*, which is described in Algorithm 6.19. 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.
- The counter `limit` used in Algorithm 6.19 has to be chosen carefully to guarantee the expected insert cost is constant. Specifically, one can show that we get this guranatee if we choose `limit` $\approx \log m$.
- Search and delete only need to check $M_1[h_1(k)]$ and $M_2[h_2(k)]$ to figure out whether a given key k is in the table, and so those operations are worst case constant time.
- 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.

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 at all 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: 2016-04-05.

[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.