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 via encapsulation.
- 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.

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 \( |N| \approx n \), where \( |N| \ll |U| \). We will use a hash table \( M \), i.e., an array with buckets \( M[0], M[1], \ldots, M[n−1] \).

Remarks:

- The standard library of almost every widely used programming language provides hash tables, sometimes by another name. In C++, they are called \texttt{unordered_map}, in Python \texttt{dict}, in Java \texttt{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 \( h : U \rightarrow M \). 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?!
- 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 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 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

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

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.

8.2. HASHING

Remarks:
- 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 $m$? 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 8.8 (Universal Family). Let $H \subseteq \{h : U \to M\}$ 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 $h \in H$ uniformly, then $H$ is called a universal family (of hash functions).

Remarks:
- This means: 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 $\lvert U \rvert^m$ many, so we need $\lvert U \rvert \log \lvert M \rvert$ bits to encode such a random function. That is even more bits than keys in our huge universe $U$.

Theorem 8.9 (Universal Hashing). Let $m$ be prime and $r \in \mathbb{N}$. Let $U = \{0, \ldots, m+1\}$ and let $M = \{0, \ldots, m\}$ with $b \leq m$. For a key $k = (k_0, \ldots, k_r) \in U$ and coefficient tuple $a = (a_0, \ldots, a_r) \in [m]^{r+1}$, define

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

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

Proof. For prime $m$ and $\delta \in \{1, \ldots, m-1\}$, any linear function $f_\delta : [m] \to [m]$ $f_\delta(x) := x \cdot \delta \mod m$ is a bijection. This means that all $x \in [m]$ have different images under $f_\delta$, and every element of $[m]$ is the image of some $x \in [m]$.

Let $(\hat{k}_0, \ldots, \hat{k}_r) = k \neq l = (l_0, \ldots, l_r) \in U$, and consider

$$h_a(k) = h_a(l) \iff \sum_{i=0}^{r} a_i \cdot k_i \equiv \sum_{i=0}^{r} a_i \cdot l_i \mod m$$

$$\iff 0 \equiv \sum_{i=0}^{r} a_i \cdot (l_i - k_i) \mod m$$

$$\iff 0 \equiv \sum_{k_i \neq l_i} a_i \cdot (l_i - k_i) \mod m.$$
The terms where $k_i = l_i$ are 0 and so we can ignore them. Now define $\delta_i := l_i - k_i$, and we get

$$0 = \sum_{i \neq j} a_i \cdot \delta_j \mod m$$

Let $S := \{i \in [m] : \delta_i \neq 0\}$ be the set of the indices of the non-vanishing terms. There are $m^{|S|}$ possibilities to choose the factors $a_i : j \in S$. If we choose the first $|S| - 1$ factors, then due to the expression being linear, we have exactly 1 choice left for the last $a_j$ to satisfy the equation. Altogether, we have $m^{|S|-1}$ choices for all $a_j$ to satisfy the equation, and so our chance of picking an $a$ that produces a collision is $\frac{m^{|S|-1}}{m} = \frac{1}{m}$.

Remarks:

- Theorem 8.9 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 $a_0, \ldots, a_r$. 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., `Strings` 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 8.9 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. `Strings` 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 $a_0 \in \mathbb{Z}$ and uses $(a_0, a_1^r, a_2^r, \ldots, a_r^r)$ 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 `Strings` with different lengths in the same hash table.

### 8.3 Static Hashing

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

**Definition 8.10 (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 8.11 (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 $C(h_b, N) \leq b$ and uses a hash table of size $m = \lceil \frac{n(b+1)}{b} \rceil$ 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

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

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,

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

and so

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

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

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

Remarks:

- According to Lemma 8.11, 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 8.12 defines perfect static hashing, which applies the result of Lemma 8.11.
Algorithm 8.12 Perfect Static Hashing

Input: fixed set of keys \( N \)
Output: hash table \( M \) and secondary hash tables \( M_i \)

Function: \( N_i := \{ k \in N : h(k) = i \} \)
Function: \( n_i := |N_i| \)

1. \( M := \text{hash table with } n \text{ buckets} \)
2. \( h := \text{hash function } N \rightarrow M \) (sampled from universal family)
3. \( M_i := \text{hash table with } 2^i \cdot n_i \text{ buckets} \)
4. \( \text{until } C(h, N) < n \)
5. \( \text{for } i \in M \text{ do} \)
6. \( M_i := \text{hash table with } 2^i \cdot n_i \text{ buckets} \)
7. \( \text{repeat} \)
8. \( h_i := \text{hash function } N_i \rightarrow M_i \) (sampled from universal family)
9. \( \text{until } C(h_i, N_i) < 1 \)
10. \( \text{end for} \)
11. \( \text{return } (M, h, (M_i)_{i \in [m]}; (h_i)_{i \in [m]}) \)

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 8.11.
- 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 8.11.

Theorem 8.13 (Perfect Static Hashing). When Algorithm 8.12 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 \( 2^i \binom{n_i}{2} = n_i(n_i - 1) \). 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

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

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:

- Note one caveat: Lines 3 and 8 of Algorithm 8.12 require sampling from a universal family. Theorem 8.9 gives us universal families for hash tables with a prime number of buckets. For non-prime hash table sizes \( n_i \), there are constructions for families of hashes where any two keys have a chance of \( \leq \frac{2}{n} \) when sampling a hash function from the family uniformly. To account for this higher chance of collision, we need to increase the hash table size by a factor of 2 (compare the proof of Lemma 8.11).
- 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 8.14 (Hashing with Chaining). In hashing with chaining, every bucket \( M[k] \) 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 8.12 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 \( \text{hashCode()}, \) either by overriding it or by inheriting it from \( \text{Object} \)) to determine each key’s bucket. For the ordering within the trees, there are two possibilities: the class implements \( \text{Comparable} \) or it does not.
- If the class of the keys implements \( \text{Comparable} \), then the natural ordering of the keys is used.
- If the keys are not \( \text{Comparable} \), then the tree uses the values returned by \( \text{System.identityHashCode} (\text{Object} \ x) \) to order keys; this method returns the same value that the default implementation of \( \text{Object.} \text{hashCode()} \) returns. This means that if your class is not \( \text{Comparable} \) and does not override \( \text{hashCode()} \), then \( \text{System.identityHashCode} (\text{Object} \ x) \) is equal for all keys within a given tree; this makes the trees degenerate to lists.

Definition 8.15 (Load Factor). The fraction \( \frac{n}{m} =: \alpha \) is called the load factor of the hash table.
8.5. WORST CASE GUARANTEES

Table 8.18: Different types of hashing with probing together with the expected number of probes per search. \( \alpha \) 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 8.18 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 \( (0, 1, 4, 2, 1, 4) \) — 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 8.19 Cuckoo Hashing: Insert

Input : key \( k \in U \) we want to insert; counter \( \text{limit} \) specifying the maximum number of tries

Data Structures : arrays \( M_1, M_2 \) of equal size

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

1. if \( M_1(h_1(k)) = k \) or \( M_2(h_2(k)) = k \) then
2. return
3. end if
4. \( t := 1 \)
5. while \( t \leq \text{limit} \) do
6. swap \( k \) with \( M_1[h_1(k)] \)
7. if \( k = \perp \) then
8. return
9. end if
10. swap \( k \) with \( M_2[h_2(k)] \)
11. if \( k = \perp \) then
12. return
13. end if
14. \( t := t + 1 \)
15. end while
16. \text{rehash}()
17. \text{CuckooHashingInsert}(k, \text{limit})

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 8.12, 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 \( n_i^2 = |M_i| \) 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).
- Another option is cuckoo hashing, which is described in Algorithm 8.19. 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 \( \text{limit} \) used in Algorithm 8.19 has to be chosen carefully to guarantee the expected insert cost is constant. Specifically, one can show that we get this guarantee if we choose \( \text{limit} \approx \log m \).
- 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.

BIBLIOGRAPHY

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-Velskii 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].

Chapter Notes


