# Fundamental Algorithms, Lecture Notes. Hash Tables

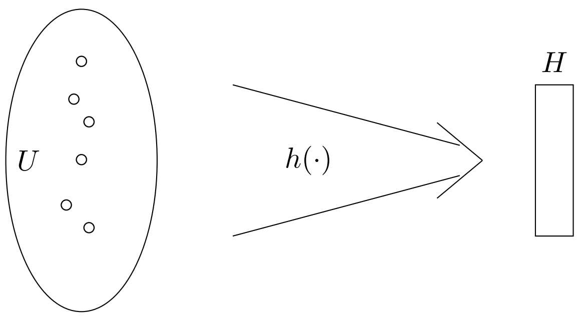
Hash tables are used to store sets while supporting membership queries, i.e. answering the question is $x$ in set $S$. This is called the dictionary data structure. It also supports insertion and deletion operations, though sometimes the set we want to store is fixed, for example, the key words in a programming language.

The obvious way to solve this problem is to use an array of bits indexed by the possible set elements. So suppose the set is drawn from a universe $U$ of items, say $0, 1, 2, \ldots, u-1$. The elements in the set do not have to be integers; this is just for illustration. Then a Boolean array $B[0 : u-1]$, has entry $B[x] =$ **True** if and only if $x \in S$. This is not a great solution when $|U| \gg |S|$, for it may not fit in memory, or even if it does, it still uses too much of the memory.

Instead, we use randomization to allow us to answer membership queries in expected (i.e. average) $O(1)$ time, while using $O(|S|)$ space in memory (space for short).

Typically, an item consists of a key and additional data. For example, in NYU's student database, the key is the unique number given to each student, and the data is whatever other information the university stores (name, address, etc.). Searches use the key value to identify an item. Consequently, our discussion will focus on the key values.

The goal in hashing is to store the set S in a table, the hash table, $H$, of size $O(S)$. We use an easily computed hash function $h$ to quickly locate an item. Specifically, $h(x)$ provides the array index in $H$ where $x$ should be stored.



Given a specific function $h$, inevitably many items of $U$ will be mapped (hashed) to the same location in $H$, for $H$ is much smaller than $|U|$ by design. We say there is a collision when two items in $S$ are mapped to the same location. The whole issue in designing hash functions is to minimize the number of collisions.

We introduce some further notation. $s$ denotes the size of $S$ and $n$ the size of the array $H$. When $S$ can change, we have to be more careful in defining $s$. We then define $s$ to be to number of items that are ever in the set $S$.

## Resolving Collisions

There are two main approaches to handling collisions: chaining and open addressing.

In chaining, each table location stores a pointer to a linked list holding all items that hash to that location. In open addressing, all items are stored in the table, in a way that we explain later. Chaining is easier to analyze and that is what we will focus on.

We now explain how to perform the dictionary operations, search, insert and delete with chaining.

Search($x$): Compute $h(x)$, and then search the list stored at $H[h(x)]$ for the item $x$, returning the item if present in the list, and returning **nil** otherwise.

Insert($x$): Compute $h(x)$, and then search the list stored at $H[h(x)]$ for the item $x$; if $x$ is not found, add item $x$ to the end of this list; otherwise, if $x$ is already in the dictionary, then report this. One can also imagine that in some applications one allows multiple items with the same key, in which case one would add the item, but the focus here is on sets with unique keys.
Delete($x$): Compute $h(x)$, and then search the list stored at $H[h(x)]$ for the item $x$; if $x$ is found, then remove it from this list; otherwise, report that $x$ was not in the dictionary.

Clearly, if the lists being searched are of length $O(1)$ then these operations will take $O(1)$ time. As already mentioned, we cannot guarantee this in the worst case, but there are ways of ensuring this happens on average.

But we want to achieve this good average performance for every possible set we might store. We don't want to take the approach we used in analyzing quicksort, where we supposed all input orderings were equally likely. The analog here would be to assume that all sets of a given size are equally likely.

We have seen that any given hash function will have sets on which it works badly. For given a hash function $h$, which maps into a table $H$ of size $n$, there must be some $i$, such that at least $|U|/n$ items $x$ have hash value $h(x) = i$. Any set $S$ drawn from this (large) subset of $|U|/n$ items will hash to a single location, and then the hash operations take $\Theta(s)$ time rather than the desired $O(1)$ time.

So the only way to get good average performance is to have a collection $\mathcal{H}$ of hash functions, $\mathcal{H} = \{h_1, h_2, \ldots, h_r\}$ say, and to choose one of these functions uniformly at random. We then need the property that for any set $S$, most of these hash functions work well. More precisely, the average performance of these hash functions on set $S$ is $O(1)$ time per operation, and this is true for every possible set $S$ of size $s$.

Universal hash functions, defined below, achieve this performance.

## Universal Hash Functions

Next we explore what constraints we should impose on $\mathcal{H}$. It seems reasonable to require that each item in the universe $U$ be approximately equally likely to be mapped to each location in $H$, when the hash function is chosen uniformly at random from $\mathcal{H}$. More formally:

**Property 1.** $\Pr[h(x) = i] = O(\frac{1}{n})$, *where* $|H| = n$, *and* $h$ *is chosen uniformly at random from* $F$.

However, this is inadequate to ensure good performance. For example, suppose $h_i(x) = i \bmod n$. The family $h_1, h_2, \cdots, h_r$ obeys Property 1, and yet each $h \in \mathcal{H}$ stores any set $S$ in a single location of $H$, resulting in time $\Theta(s)$ for each dictionary operation.

Thus the next constraint is to demand that pairs of items do not cluster; in more technical language, each pair of items is distributed independently. Formally:

**Property 2.** *1. (weak version)* $\Pr[h(x) = h(y)] = O(1/n)$.

*2. (strong version)* $\Pr[h(x) = i$ *and* $h(y) = j] = O(1/n^2)$ *for all* $i$ *and* $j$, $0 \le i, j < |H|$.

**Lemma 1.** *Let set* $S$ *of size* $n$ *be stored in* $H$. *Let* $x \in U$. *The expected size of bucket* $H[h(x)]$ *is* $O(1 + s/n)$ *if* $x \in S$ *and* $O(s/n)$ *if* $x \notin S$, *assuming the weak form of Property 2.*

*Proof.* Let $y_1, y_2, \cdots, y_m$ be the items in $S$. Let $Y_i = 1$ if $y_i$ is in bucket $H[h(x)]$ and let $Y_i = 0$ otherwise. Then the expected number of items in $H[h(x)]$, aside from $x$, is

$$E\Big[\sum_{y_i \neq x} Y_i\Big] = \sum_{y_i \neq x} E[Y_i] = \sum_{y_i \neq x} O(1/n) = O(s/n).$$

□

**Corollary 2.** *If the weak form of Property 2 holds, each dictionary operation takes time $O(1+s/n)$.*

**Comment**. As a rule one chooses $n = \Theta(s)$, which ensures the expected or average time for each of these operations is constant.

It remains to exhibit a class of universal hash functions.

We start with a solution with $n = p$, where $p$ is a prime number. In addition, we suppose that each item being hashed lies in the range $[0, p-1]$. This is an unrealistic assumption and we will remove it. But bear with us for now.

Our hash functions are quite simple. We define $h_a(x) = ax + b \bmod p$, where $0 \le a, b < p$. Let $\mathcal{H}$ be the set $\{h_{a,b} \,|\, 0 \le a, b < p\}$; i.e. all $p^2$ hash functions of this form.

**Lemma 3.** *The set $\mathcal{H}$ defined above satisfies the strong form of Property 2. In fact, it safisfies the condition with the best constant possible, namely: if $x \ne y$, then for any pair $(c,d)$ of integers with $0 \le c, d < p$,*

$$\Pr\left[h_{a,b}(x) = c \text{ and } h_{a,b}(y) = d\right] = \frac{1}{p^2}.$$

*Proof.* For the condition to hold we need that $ax + b = c \bmod p$ and $ay + b = d \bmod p$. Taking their difference, this implies $a(x-y) = c - d \bmod p$, or $a = (x-y)^{-1}(c-d) \bmod p$. Now every non-zero integer has an inverse mod $p$ when $p$ is a prime, and therefore there is a single value of $a$ that satisfies this condition. As the choice of $a$ is random, this choice is made with probability $1/p$.

Suppose $a$ has been chosen to satisfy the condition $a(x-y) = c - d \bmod p$. Then what is the probability that $ax+b = c \bmod p$? We would need that $b = c - ax \bmod p$. Again, this choice occurs with probability $1/p$. So the probability of choosing the right $a$ and $b$ is $1/p^2$. □

However, the interesting case in when the range of the items being hashed is much larger than the hash table size. Then we will choose a prime $p$ that is larger than this range, while using essentially the same set of hash functions. Now, we define $h_{a,b}(x) = (ax + b \bmod p) \bmod n$; i.e. we start our hash function computation as before, and then do a further mod $n$ computation to get the range of values down to $[0 : n-1]$. The resulting hash function is still universal, as we show next.

**Lemma 4.** *The set $\mathcal{H}$ defined above satisfies the strong form of Property 2. More precisely: if $x \ne y$, then for any pair $(c,d)$ of integers with $0 \le c, d < n$,*

$$\Pr\left[h_{a,b}(x) = c \text{ and } h_{a,b}(y) = d\right] \le \frac{1}{n^2} + \frac{1}{p^2}.$$

*Proof.* We know that the probability of $ax + b = c' \bmod p$ and $ay + b = d' \bmod p$ is $1/p^2$. We ask for which $c'$ and $d'$ is it the case that $c' = c \bmod n$ and $d' = d \bmod n$, which we call the good events, for this tells us how many probability $1/p^2$ events have $x$ hashing to $c$ and $y$ hashing to $d$. Well, there are at most $\lceil p/n \rceil$ such $c'$ and at most $\lceil p/n \rceil$ such $d'$. Therefore the probability that a good event occurs is at most

$$\frac{1}{p^2}\left(\frac{p}{n} + 1\right)^2 = \frac{1}{n^2} + \frac{1}{p^2}.$$

□

### Open Addressing

Remember that in open addressing there are no linked lists: items are being directly stored in the table $H$. Thus we need a different way of handling collisions.

As with chaining, to search for item $x$, we start by computing $h_1(x)$. If $H[h_1(x)] = x$ then the item is found, and if $H[h_1(x)] =$ **nil** then the item is not present in the dictionary. However, if $H[h_1(x)] = y$, then we need to look elsewhere for item $x$.

To this end, we use a sequence of $m = |H|$ hash functions, $h_1, h_2, \cdots, h_{|H|}$. This sequence of hash functions is not the collection of universal hash functions we just saw. If $H[h_1(x)] \neq x$, then we compute $h_2(x)$, and proceed as before: If $H[h_2(x)] = x$ then the item is found, if $H[h_2(x)] =$ **nil** then the item is not present in the dictionary, and if $H[h_2(x)] \neq x$ or **nil**, then the search needs to continue. In general, if the search has reached the $i$-th iteration, we compute $h_i(x)$, and proceed as before: If $H[h_i(x)] = x$ then the item is found, if $H[h_i(x)] =$ **nil** then the item is not present in the dictionary, and if $H[h_i(x)] \neq x$ or **nil**, the search continues. The goal is to have each of the values $h_1(x), h_2(x), \ldots$ be distinct, so that new locations keep being examined. If every location in $H$ is examined then we know that $x$ is not in the dictionary and the search can end at that point.

Of course, we don't want to search a lot of locations. So one should have the table be larger than the maximum set size, say twice as large, so that one is likely to quickly find an empty location in $H$.

The dictionary operations proceed as follows. In turn, $h_1(x), h_2(x), \ldots$ are computed, until an empty table location or the item $x$ itself is found. An empty location indicates the item is not present in the table, and if the operation is an insertion, it is added in this location. Deletions complicate the situation. In order for searches to work correctly following a deletion, once filled (but now empty) locations must be marked to indicate this; a search for item $x$ will not stop at a marked location, but will only stop on reaching either $x$ or an unmarked empty location.

In double hashing, $h_i(x) = h_1(x) + (i-1)g(x)$, where $h_1$, and $g$ are suitable functions. A particularly simple choice for $g(x)$ is $g(x) = 1$; this is called linear probing.

The analysis of open addressing hashing is non-trivial and we will not address it in this course.

### Hashing multi-field items

Often we wish to hash long strings or other large structures. We view them as a sequence of words, say $x = (x_1, x_2, \ldots, x_k)$. We then use $k$ hash functions, $h_1, h_2, \ldots, h_k$. The hash $h(x)$ is defined as $\sum_{i=1}^{k} h_i(x_i) \bmod n$.

**Lemma 5.** *If each $h_i$ is drawn uniformly from a set $\mathcal{H}$ of universal hash functions, then the resulting hash function is also universal.*

*Proof.* We need to show that the probability that both $h(x) = c$ and $h(y) = d$ is $O(n^2)$. Suppose that $x_i \neq y_i$ (note that there must be some coordinate on which $x$ and $y$ differ as $x \neq y$). Then we want $h_i(x_i) = c - \sum_{1 \leq j \leq k, j \neq i} h_j(x_j) \bmod n$ and $h_i(y_i) = d - \sum_{1 \leq j \leq k, j \neq i} h_j(y_j) \bmod n$. But the probability of this event is $O(n^2)$ by the universality of the $h_i$. □

## Bloom Filters

Bloom filters use multiple hash functions to answer set membership queries, but with a modest probability of a one-sided error. That is, a Bloom filter may report incorrectly that an item is in the set, but it will always report an item is present if it is.

A Bloom filter stores a set $S$ using a collection $h_1, h_2, \cdots, h_m$ of hash functions to access a table of bits $B[0:n-1]$. For each $x \in S$, the entries $B[h_i(x)]$ are all set to 1. All other entries remain at 0.

To test if $x \in S$, the entries $B[h_i(x)]$, $1 \le i \le m$, are checked; if any is 0, $x \notin S$. If they are all equal to 1, then $x$ is reported to be in $S$; this is where an error may occur, because the fact that these bits are all equal to 1 just means that the entries $B[h_i(x)]$, $1 \le i \le m$, were each set to 1 as the result of the hashes of possibly different items. But Lemma 6 below, shows that with high probability, the answer is correct.

Deletion of an item $x$ is more problematic, as we cannot simply set $B[h_i(x)]$ to 0. It should only be set to 0 if this was the only item mapping to this location. One solution is to keep counters recording how many items are in each bucket, but this takes much more space.

We will analyze Bloom filters using a heuristic analysis, meaning that it will be based on an unrealistic assumption.

**Assumption 1.** *Suppose $x \ne y$. Then the probability that $h_i(x) = h_j(y)$ is $1/n$. (Or more generally, the bound is at most $c/n$ for some constant $c \ge 1$; but we will only analyze the case $c = 1$.)*

**Lemma 6.** *If Assumption 1 holds, then we can choose $m$ so that the probability that an item $x \notin S$ is reported as being in $S$ is at most $\frac{1}{2^{n/s}}$ (to a close approximation).*

*Proof.* Consider the location $h_i(x)$. The probability that $h_j(y) \ne h_i(x)$ is $1 - \frac{1}{n}$. Therefore the probability that there are no collisions with $h_i(x)$ is $\left(1 - \frac{1}{n}\right)^{ms}$. The probability that there is a collision is $1 - \left(1 - \frac{1}{n}\right)^{ms}$. Therefore the probability that there is a collision with every $h_i(x)$ is $\left(1 - \left(1 - \frac{1}{n}\right)^{ms}\right)^m$; this is the probability of $x$ being reported as being in $S$ when it is not in $S$.

To bound this expression we use a Taylor series expansion for $\frac{1}{e}$:

$$\left(\frac{1}{e}\right)^{ms/n} = 1 - \frac{ms}{n} + \frac{1}{2!}\left(\frac{ms}{n}\right)^2 - \cdots.$$

We also observe that

$$\left(1 - \frac{1}{n}\right)^{ms} = 1 - \frac{ms}{n} + \frac{1}{2!}\left(\frac{ms(ms-1)}{n^2}\right) - \cdots.$$

We deduce that

$$\left(\frac{1}{e}\right)^{ms/n} - \frac{1}{2n^2} \le \left(1 - \frac{1}{n}\right)^{ms} \le \left(\frac{1}{e}\right)^{ms/n}.$$

We choose $m = \ln 2 \cdot \frac{n}{s}$ (this would be the optimal choice if the $1/2n^2$ term in the above expression was not present). Substituting in the probability expression we obtain an upper bound on the misreporting probability of

$$\begin{aligned}
\left(1-\left(1-\tfrac{1}{n}\right)^{ms}\right)^{m} &\le \left(1-\left(\frac{1}{e}\right)^{\ln 2}+\frac{1}{2n^2}\right)^{\ln 2\cdot n/s}\\
&\le \left(\frac{1}{2}+\frac{1}{2n^2}\right)^{\ln 2\cdot n/s}\\
&\le \left(\frac{1}{2}\right)^{\ln 2\cdot n/s}\cdot\left(1+\frac{1}{n^2}\right)^{\ln 2\cdot n/s}\\
&\le \frac{1}{2^{n/s}}\cdot\left(1+\frac{1}{n^2}\right)^{\ln 2\cdot n/s}\\
&\le \frac{1}{2^{n/s}}\cdot\left(1+\frac{\ln 2}{ns}\right)\\
&\approx \frac{1}{2^{n/s}}.
\end{aligned}$$

□

## Passwords

Passwords are also stored using hash functions. Here though there is a further requirement: it needs to be hard to determine $x$ from $h(x)$, i.e. hard to invert the hash function. Then the gatekeeper, on receiving the hash of a password, has high confidence that the provider indeed knows $x$.

This is not the case for the family of universal hash functions we considered. There are hash functions that are used in practice and they are considered hard to invert. But the evidence is just the belief that this has not happened despite considerable effort directed at breaking the hash functions (i.e. finding an inversion algorithm). There are no proofs.