# Hash Tables

January 24, 2018

## Contents

# 1 The Beginning: Direct Addressing

At the beginning, the number of elements in a set of numbers to be stored in a computer system used to be not so large or having a wide range. Then, a simple table $T[0, 1, ..., m-1]$ called, *direct-address table,* could be used to store those numbers. As the situation became more and more complex, and a new idea came to be:

**Definition** An associative array, map, symbol table, or dictionary is an abstract data type composed of a collection of tuples $\{(key, value)\}$

This can bee seen in the example of dictionaries in any spoken language. The problem became more complex when the range of the possible values for the keys at the tuples became unbounded. Therefore a new type of data structure needed to be invented to avoid the inherent sparsity in the new data structure.

# 2 The Solution: Hash Tables

An initial solution, given a small storage table for our tuples, can be designed around the following idea:

- Try to map the keys using some function, hash function $h : U \to \{0, 1, 2, ..., m-1\}$, so the generated new indexes for storing the tuples are bounded then they can be stored to the hash table $h[0, 1, ..., m-1]$.

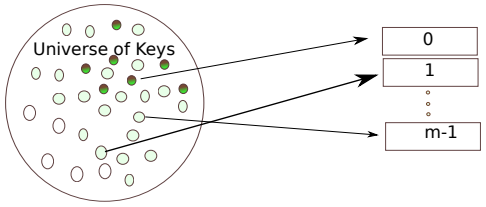


Figure 1: From a huge volume of keys to a small storage.

Here is where a new problem arises: Sooner of later certain keys will mapped, collide, to the same slot.

Thus, it is necessary to minimize the number of collisions. There are tow ways of handling this problem:

1. Device a hash function $h$ that appears random. Therefore, the possible collision of the $m$ keys is minimized to $\frac{1}{m}$. This needs to be done under the constraint that given a key $k$ you always get the same $h(k)$ index.

2. The other option is to extend the data structure to add some kind of collision policy.

# 3 Chaining

A simple collision policy is to use a linked list at each slot, thus each time a key maps into a slot the tuple is stored at the head of the list. There are three operations to support this new extended data structure:

**CHAINED-HASH-INSERT(T,x)** Insert at the head of the list $T[h(x.key)]$

**CHAINED-HASH-SEARCH(T,k)** Search for an element with key $k$ in list $T[h(k)]$

**CHAINED-HASH-DELETE(T,x)** Delete $x$ from the list $T[h(x.key)]$

Here, we have that the worst case of insertion is O(1). In the case of search and deletion, we have that we depend on the size of the list at the bucket $T[h(k)]$ or $T[h(x.key)]$. To obtain a more realistic analysis of the search and deletion complexities.

# 4 Expected Analysis: Simple Uniform Hashing Assumption

In order to do the expected analysis, it is necessary to assume the following assumption

- Any given key is equally likely to hash into any of the m slots, independently of where any other key has hashed to.

Then, if you have $n$ tuples to be stored in $j = 0, 1, 2, ..., m-1$ buckets, we can say that $|T[j]| = n_j$, thus

$$n = n_0 + n_1 + ... + n_{m-1}.$$

Then, if we apply the expected value to each of the $n_j$, we have that $E[n_j] = \alpha = \frac{n}{m}$.

The analysis can be simplified by realizing that any deletion has two operations

1. A search.
2. The deletion itself.

And if the deletion takes $O(1)$, then we have two cases: Successful search and unsuccessful search.

**Theorem** 11.1

In a hash table in which collisions are resolved by chaining, an unsuccessful search takes average-case time $\Theta(1 + \alpha)$, under the assumption of simple uniform hashing.

**Proof:**

If the key $k$ is not stored in the table, we can assume that it can hash in any bucket $T[h(k)]$ with length $E\left[n_{h(k)}\right] = \alpha$. In order to find that the key is not in the bucket, it is necessary to examine the entire list. Then, we have that total time required is $\Theta(1+\alpha)$.

**Theorem** 11.2

> In a hash table in which collisions are resolved by chaining, a successful search takes average-case time $\Theta(1+\alpha)$ under the assumption of simple uniform hashing.

**Proof:**

First, we have the following assumptions:

- The key being searched can be any of the keys stored in the table.
- The number of elements examined during a successful search for an element $x$ is one more that the number of elements in front of $x$. This elements where inserted after $x$ was inserted.

Then, we define:

1. $x_i$ the ith element inserted into the table, for $i = 1, 2, ..., n$.
2. $k_i = x_i.key$.

Thus, we can define the following indicator function $X_{ij} = I\{h(k_i) = h(k_j)\}$, which has $P\{h(k_i) = h(k_j)\} = \frac{1}{m}$. Finally, $\mathrm{E}[X_{ij}] = \frac{1}{m}$. Finally, we count all the elements before each particular $x_i$ and $x_i$ itself. This means that for a particular $i$, we have that

$$1 + \sum_{j=i+1}^{n} X_{ij}$$

is the total numbers of element being examined for $i$. Then the total count for all the elements is

$$\sum_{i=1}^{n} \left[ 1 + \sum_{j=i+1}^{n} X_{ij} \right]$$

Then, taking the average and the expected time

$$
\begin{aligned}
E\left[\frac{1}{n}\sum_{i=1}^{n}\left[1+\sum_{j=i+1}^{n}X_{ij}\right]\right] &= \\
&= \frac{1}{n}\sum_{i=1}^{n}\left[1+\sum_{j=i+1}^{n}E[X_{ij}]\right] \\
&= \frac{1}{n}\sum_{i=1}^{n}\left[1+\sum_{j=i+1}^{n}\frac{1}{m}\right] \\
&= 1+\frac{1}{nm}\sum_{i=1}^{n}[n-i] \\
&= 1+\frac{1}{nm}\left[\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right] \\
&= 1+\frac{n-1}{2m} \\
&= 1+\frac{\alpha}{2}-\frac{\alpha}{2n}.
\end{aligned}
$$

Thus, $T(1+\frac{\alpha}{2}-\frac{\alpha}{2n})=\Theta(1+\alpha)$, and if $n=O(m)\Longrightarrow\alpha=\frac{n}{m}=\frac{O(n)}{n}=1$. Thus, searching takes constant time on average $O(1)$, and it is the same for insertion. Now for deletion, using a double linked list, we can achieve constant time too. Then, we can support the dictionary operations in $O(1)$ time on average.

# 5 Hash Functions

## 5.1 What is a good hash function?

It is simple, we would like to have a hash function that can hash any key into any slot with equally likely probability, the assumption of simple uniform hashing. For example, if we know that the keys are uniformly distributed in the range , something as simple as $h(k)=\lfloor mk\rfloor$ satisfies the assumption.

In reality, we often use heuristic methods to create hash functions that try to be independent of any pattern that exist in the data.

There are three strategies for hash functions that are explored in the Cormen's textbook:

1. The division method.
2. The multiplication method.
3. Universal Hashing.

## 5.2 Interpreting keys as a natural number

Given any key, it is possible to interpret them as. For example, it is possible to interpret a character string as an integer by using the position of the character in the ASCII character set. For example, $pt \approx 112 * 128^1 + 116 * 128^0$.

### 5.2.1 The division method

In this method, we take the reminder of the division of a key as our hash value:

$$h(k) = k \mod m$$

When using this method, we avoid certain values of $m$:

1. $m$ should not be a power of 2.

This is because given that $k = a_t 2^t + ... + a_m 2^p + a_{p-1} 2^{p-1} + ... + a_o 2^0$, then $h(k) = a_{p-1} 2^{p-1} + ... + a_0 2^0$ , and unless you have a good distribution of the elements in the range $\left[0, 2^{p-1} - 1\right]$, it is not a good idea. It is better that the hash depends on all the bits. Another problem of using hash values a power of 2 is the fact that in the case of keys as natural numbers is that for character with bit values larger than $p$ exchanging them do not change the value of the hashing if you exchange the position of the characters. Therefore, a possible value of $m$ is a prime number not near to an exact power of 2.

### 5.2.2 The multiplication method

The multiplication method for creating hash functions has two steps

1. Multiply the key $k$ by a constant $A$ in the range $0 < A < 1$ and extract the fractional part of $kA$.

2. Then, you multiply the value by $m$ an take the floor.

$$h(k) = \lfloor m \left(kA \mod 1\right) \rfloor$$

The good part of this method is that $m$ is not critical, normally $m = 2^p$. This makes easier to implement the method in a computer using the following steps:

1. First, imagine that the word in a machine has $w$ bits size and $k$ fits on those bits.

2. Then, select an $s$ in the range $0 < s < 2^w$. Then assume $A = \frac{s}{2^w}$.

3. Now, we multiply $k$ that fits in the word size $w$ by the number $s = A2^w$.

4. The result of that is $r_1 2^w + r_0$, a $2w$-bit value word, where the first $p$-most significative bits of $r_0$ are the desired hash value.

Let us explain with an example:

1. First select $s = 2^t = A2^w$ with $t < w$, then $A = 2^{t-w}$.
2. In our case, we could say $t = \lfloor \frac{w}{2} \rfloor$.
3. Then, the values of $A$ are stored now in the first half part of the $w$ word right starting next at the $\lfloor \frac{w}{2} \rfloor$-bit.
4. Now if $k = 2^{\lfloor \frac{w}{2} \rfloor}$, then $k * s = r_1 2^w + r_2$ or the values are stored starting next to the $w$-bit.

## 5.3 The problem of clustering in hash functions

It is possible to see that sooner or latter we can pick up a hash function that does not give us the desired uniform randomized property. Then, it is necessary to introduce some measurement of clustering

**Definition** If bucket $i$ contains $n_i$ elements, then we have:

$$C = \frac{m}{n-1}\left[\frac{\sum_{i=1}^{m} n_i^2}{n} - 1\right].$$

An estimate of the variance.

This formula give us a way to measure how good are our hash function is. That has the following properties:

1. If $C = 1$, then you have uniform hashing.
2. If $C > 1$, it means that the performance of the hash table is slowed down by clustering by approximately a factor of $C$.
3. If $C < 1$, , the spread of the elements is more even than uniform!!! Not going to happen!!!

For example, if $m = n$ and all elements are hashed to a single bucket, we have that $C = n$. In the case they hash to their own bucket $C = 0$. Then, if $C < 1$ we have a hash function that spreads the values better that a random hash function, ok this does not going to happen.

### 5.3.1 Probabilistic Analysis of the Clustering

Consider bucket $i$ containing $n_i$ elements, with $X_{ij}$= I{element $j$ lands in bucket $i$}, then $n_i = \sum_j X_{ij}$. Thus, if we assume the uniform hash assumption:

$$n_i = \sum_{j=1}^{n} X_{ij} \tag{1}$$

We have that

$$E[X_{ij}] = \frac{1}{m},\ E\left[X_{ij}^2\right] = \frac{1}{m} \tag{2}$$

We look at the dispersion of $X_{ij}$

$$Var\left[X_{ij}\right] = E\left[X_{ij}^2\right] - \left(E\left[X_{ij}\right]\right)^2 = \frac{1}{m} - \frac{1}{m^2} \tag{3}$$

Because independence of $\{X_{ij}\}$, the scattering of $n_i$ $\left(Var\left(\sum_{j=1}^{n} X_{ij}\right) = \sum_{j=1}^{n} Var\left(X_{ij}\right)\right)$

$$\begin{aligned} Var\left[n_i\right] &= Var\left[\sum_{j=1}^{n} X_{ij}\right] \\ &= \sum_{j=1}^{n} Var\left[X_{ij}\right] \\ &= nVar\left[X_{ij}\right] \end{aligned}$$

What about the **range** of the possible number of elements at each bucket?

$$\begin{aligned} Var\left[n_i\right] &= \frac{n}{m} - \frac{n}{m^2} \\ &= \alpha - \frac{\alpha}{m} \end{aligned}$$

But, we have that

$$E\left[n_i^2\right] = E\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{j=1}^{n} \sum_{k=1,k\neq j}^{n} X_{ij}X_{ik}\right] \tag{4}$$

Or

$$E\left[n_i^2\right] = \frac{n}{m} + \sum_{j=1}^{n} \sum_{k=1,k\neq j}^{n} \frac{1}{m^2} \tag{5}$$

We re-express the range on term of expected values of $n_i$

$$E\left[n_i^2\right] = \frac{n}{m} + \frac{n\left(n-1\right)}{m^2} \tag{6}$$

Then

$$\begin{aligned} E\left[n_i^2\right] - E\left[n_i\right]^2 &= \frac{n}{m} + \frac{n\left(n-1\right)}{m^2} - \frac{n^2}{m^2} \\ &= \frac{n}{m} - \frac{n}{m^2} \\ &= \alpha - \frac{\alpha}{m} \end{aligned}$$

Finally, we have that

$$E\left[n_i^2\right] = \alpha\left(1 - \frac{1}{m}\right) + \alpha^2 \tag{7}$$

Now we build an estimator of the mean of $n_i^2$ which is part of $C$

$$\frac{1}{n}\sum_{i=1}^{m} n_i^2 \tag{8}$$

Thus

$$\begin{aligned} E\left[\frac{1}{n}\sum_{i=1}^{m} n_i^2\right] &= \frac{1}{n}\sum_{i=1}^{m} E\left[n_i^2\right] \\ &= \frac{m}{n}\left[\alpha\left(1 - \frac{1}{m}\right) + \alpha^2\right] \\ &= \frac{1}{\alpha}\left[\alpha\left(1 - \frac{1}{m}\right) + \alpha^2\right] \\ &= 1 - \frac{1}{m} + \alpha \end{aligned}$$

We can plug back on $C$ using the expected value

$$\begin{aligned} E\left[C\right] &= \frac{m}{n-1}\left[E\left[\frac{\sum_{i=1}^{m} n_i^2}{n}\right] - 1\right] \\ &= \frac{m}{n-1}\left[1 - \frac{1}{m} + \alpha - 1\right] \\ &= \frac{m}{n-1}\left[\frac{n}{m} - \frac{1}{m}\right] \\ &= \frac{m}{n-1}\left[\frac{n-1}{m}\right] \\ &= 1 \end{aligned}$$

Thus, using a hash table that enforce a uniform distribution in the buckets, we get that $C = 1$ or the best distribution of keys. A similar analysis can be done for $C > 1$ and $C < 1$.

# 6 Universal Hashing

You could have a nasty hacker trying to make that her/his keys hashes to the same slot, making the retrieval of the hash key $\Theta(n)$. Then, we require for each hash table to select the hash in a random way, which is totally independent of the set of keys. This technique is called Universal Hashing, and it provides with a good performance on average. For this, we require a well selected set of hash functions with the following property:

**Definition** Universal Collection $\mathcal{H} = \{h : U \to \{0, 1, ..., m-1\}\}$ is a set such that each pair of distinctions keys $k, l$ the number of hash functions for which $h(k) = h(l)$ is at most $\frac{|\mathcal{H}|}{m}$. Thus, when picking at random a hash function the chance of coalition for $k, l$ is at most $\frac{1}{m}$.

Then we have the following theorem.

**Theorem** 11.3

Suppose that a hash function $h$ is chosen randomly from a universal collection of hash functions and has been used to hash $n$ keys into a table $T$ of size $m$, using chaining to resolve collisions. If key $k$ is not in the table, then the expected length $E\left[n_{h(k)}\right]$ of the list that key $k$ hashes to is at most the load factor $\alpha = \frac{n}{m}$ If key $k$ is in the table, then the expected length $E\left[n_{h(k)}\right]$ of the list containing key $k$ is at most $1 + \alpha$.

**Proof:** Here, we can notice that everything depends on the hash function not the keys themselves. Then, for each $k, l$ we have $X_{kl} = I\{h(k) = h(l)\}$. Then, by definition

$$Pr\{h(k) = h(l)\} \leq \frac{1}{m}.$$

Then, we define for each key $k$, $Y_k = \sum_{l \in T, l \neq k} X_{kl}$. Thus

$$E[Y_k] = E\left[\sum_{l \in T, l \neq k} X_{kl}\right] \leq \sum_{l \in T, l \neq k} \frac{1}{m}$$

We have two cases:

**Case** 1. If $k$ is not in the table, then $n_{h(k)} = Y_k$ and $|\{l | l \in T \text{ and } l \neq k\}| = n$. Thus, $E\left[n_{h(k)}\right] = E[Y_k] \leq \frac{n}{m} = \alpha$.

**Case** 2. If $k$ is in the table, then $k$ appears in list $T[h(k)]$ and the count $Y_k$ does not include key $k$, thus $n_{h(k)} = Y_k + 1$ and $|\{l | l \in T \text{ and } l \neq k\}| = n-1$. Therefore, $E\left[n_{h(k)}\right] = E[Y_k] + 1 \leq \frac{n-1}{m} + 1 = 1 + \alpha - \frac{1}{m} < 1 + \alpha$.

From this you can prove the following corollary.

**Corollary** 11.4

Using universal hashing and collision resolution by chaining in an initially empty table with $m$ slots, it takes expected time $\Theta(n)$ to handle any sequence of $n$ INSERT, SEARCH, and DELETE operations containing $O(m)$ INSERT operations.

## 6.1 Examples of Universal Hashing Sets

### 6.1.1 Using the Module function

Choose a primer number $p$ large enough so that every possible key $k$ is in the range $[0, ..., p-1]$

$$\mathbb{Z}_p = \{0, 1, ..., p-1\} \text{and } \mathbb{Z}_p^* = \{1, ..., p-1\}$$

Define the following hash function:

$$h_{a,b}(k) = ((ak + b) \mod p) \mod m, \forall a \in Z_p^* \text{and } b \in Z_p$$

The family of all such hash functions is:

$$H_{p,m} = \{h_{a,b} : a \in Z_p^* \text{and } b \in Z_p\}$$

where:

- $a$ and $b$ are chosen randomly at the beginning of execution.
- The class $H_{p,m}$ of hash functions is universal.

**Theorem 1.** *The class $H_{p,m}$ of hash functions defined by the module based equations is universal.*

# 7 Open Addressing

In open addressing, all elements occupy the hash table itself. Therefore, in order to do insertions, we probe the hash table until we find and empty slot. Thus, the hash function is defined as

$$h : U \times \{0, 1, 2, ..., m-1\} \to \{0, 1, ..., m-1\}$$

This generates the probe sequence $\langle h(k, 0), h(k, 1), ..., h(k, m-1)\rangle$ is a permutation of $\langle 0, 1, 2, ..., m-1\rangle$.

## 7.1 Analysis of Open Addressing

Then, we can prove the following theorem for an unsuccessful search under uniform hashing.

**Theorem** 11.6

> Given an open-address hash table with load factor $\alpha = \frac{n}{m} < 1$, the expected number of probes in an unsuccessful search is at most $\frac{1}{1-\alpha}$ assuming uniform hashing.

**Proof**

If we make the following assumption

- In an unsuccessful search, every probe access a slot that does not contain the desired key, then when probing the last slot, it finds an empty slot.
- $X$ is the number of probes made in an unsuccessful search
- $A_i$=an ith probe occurs and it is to an occupied slot.
- $\{X \geq i\} = A_1 \cap A_2 \cap ... \cap A_{i-1}$

Then:

$$Pr\{A_1 \cap A_2 \cap ... \cap A_{i-1}\} = Pr\{A_1\} Pr\{A_2|A_1\} ...P\{A_{i-1}|A_1 \cap ... \cap A_{i-2}\}$$

Thus, we have that

- $Pr\{A_1\} = \frac{n}{m}$.
- $Pr\{A_2|A_1\} = \frac{n-1}{m-1}$.
- ...
- $Pr\{A_j|A_1 \cap ... \cap A_{j-1}\} = \frac{n-j+1}{m-j+1}$.

In this way, we have

$$Pr\{X \geq i\} = \frac{n}{m} \times \frac{n-1}{m-1} \times ... \times \frac{n-i+2}{m-i+2} \leq \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1}$$

Now, we have that the expected value is:

$$E[X] = \sum_{i=1}^{\infty} Pr\{X \geq i\} \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}.$$

This can be seen in the following way: Given that we make the first probe, we have a probability $\alpha$ to find an occupied slot. In the second probe, we have probability $\alpha^2$ to find a occupied slot, and so on.

From here, we have the performance of HASH-INSERT from the following corollary.

**Corollary** 11.7

> Inserting an element into an open-address hash table with load factor requires at most $\frac{1}{1-\alpha}$ probes on average, assuming uniform hashing.

**Proof:** Quite simple you require an unsuccessful search then the insertion.

Now for a successful search.

**Theorem** 11.8

> Given an open-address hash table with load factor $\alpha < 1$, the expected number of probes in a successful search is at most $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$ assuming uniform hashing and assuming that each key in the table is equally likely to be searched for.

**Proof:**

Basically a successful search is similar than an insertion. Then if $k$ was the (i+1)st key inserted into the table, then from corollary 11.7 we have the number of possible searches is at most $\frac{1}{1-\frac{i}{m}} = \frac{m}{m-i}$. Then take the average

$$\begin{aligned}
\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} &= \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} = \frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k} \leq \frac{1}{\alpha}\int_{m-n}^{m}\left(\frac{1}{x}\right)dx \\
&= \frac{1}{\alpha}\ln\frac{m}{m-n} \\
&= \frac{1}{\alpha}\ln\frac{1}{1-\alpha}.
\end{aligned}$$