Hash Tables

A **hash table** is a datastructure designed for O 1 expected time operations for Find, Insert, and Delete by indexing data within an array by making use of a **hash code**.

The following table shows the expected time for each one of these operations.

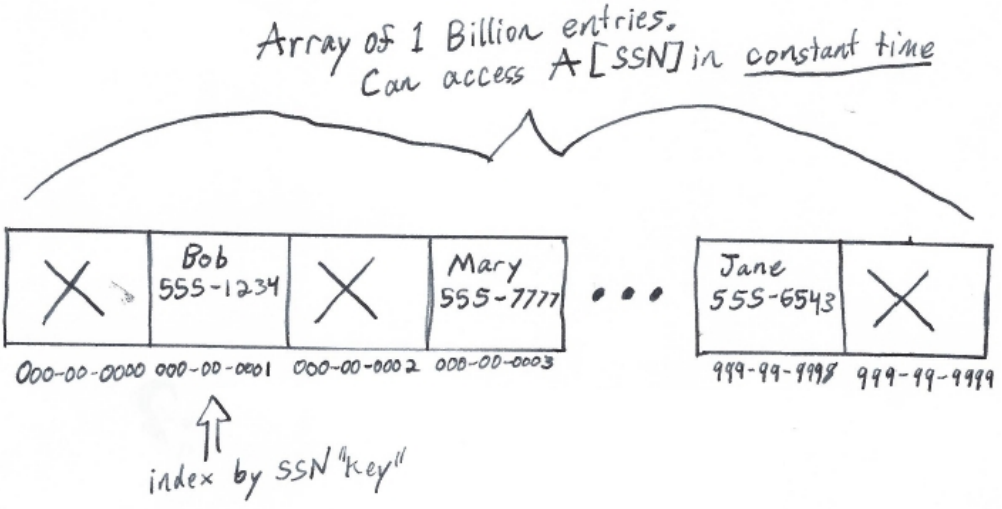
	Best Case	Worst Case	Average Case
Find	O1	ON	O1
Insert	O1	ON	O1
Delete	O1	ON	O1

These **hash tables** and **maps** are a special type of **associative arrays**. **Associative arrays** map associate keys with a set of values.

General idea

The basic idea behind hash tables to to take advantage of the O(1) FindIdx operation in Arrays. Given an index, we can find any index into the array in constant time.

For example, imagine if we were indexing every USA citizen by their social security number. As a SSN is a 10 digit number, we would be able to store every citizen without collision in an array with 1 billion entries

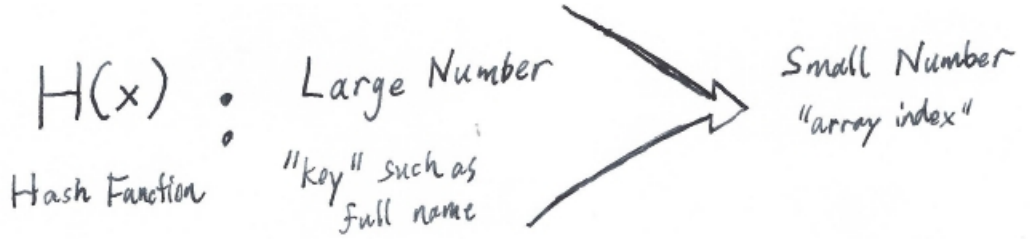


As we can see, we can index records by SSN “key” using an array of 1 billion (10^10) entries. This array would take gigabytes of RAM and is already possibly too large to implement without special server hardware.

However, if we wanted to index records by a larger number, such as a person’s full name (first and last), if the longest name has 32 characters, and we allow 26 letters + 1 spaces, we could require an array with 27^32 entries, or ~ 6.35 * 10^45 entries, each one multiple bytes. Most of these entries would be empty.

Hash Function

A Hash function H(x) is a function that maps a large number (key) to a smaller number (array index). The ideal hash function would randomly distribute the data uniformly across all of the values within it's range.



A hash table is where we use the hash values as indices into an array. There are several hash functions that we might consider:

Division Method $H(x) = x \% k$

Multiplication Method $H(x) = \lfloor k(xA \% 1) \rfloor$ A is irrational

Cryptographic Hash MD5, SHA1

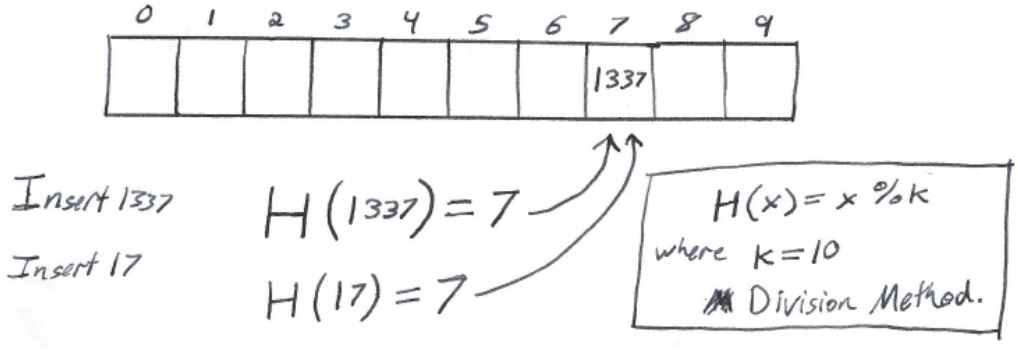
Of these methods, the Division method is the simplest and the easiest to calculate on paper and pencil. The Division method, however suffers from an issue that if k has any common factors with the distribution of keys, then this can cause a lot of collisions. As such, it is best practice to make "k" (the capacity of the table) to be a prime number such as to reduce the possibility of collisions with most datasets.

The multiplication method also care must be taken to avoid collisions and ensure that the data is uniformly distributed throughout the table. The multiplcation method we multiply key x by a factor of A, and it is best practice that A be an irrational number such as the golden ration such as to ensure that multiplication is likely to distributed entries throughout the table. One advantage of the multiplcation method is that k (the table capacity) does not need to be prime.

Only cryptographic hash functions ensure that the data will be uniformly distributed for any reasnable dataset. Cryptographic hash functions are much more complex and computationally expensive. The typically involve using lookup tables to scramble data as part of the calculation.

Collisions

A collision is when two keys map have the same hash value. No matter how ideal the hash code, collisions can always occur.

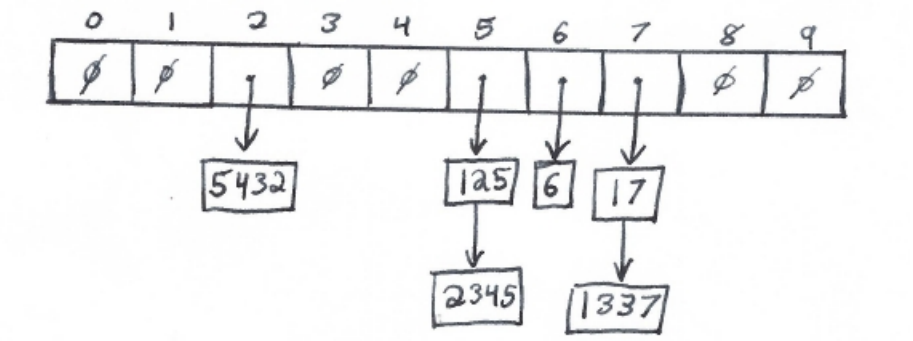


There are two primary methods to resolve collisions:

Closed Addressing where every table entry has a linked list
Open Addressing where we use a "probing" function p(x) to address other indices

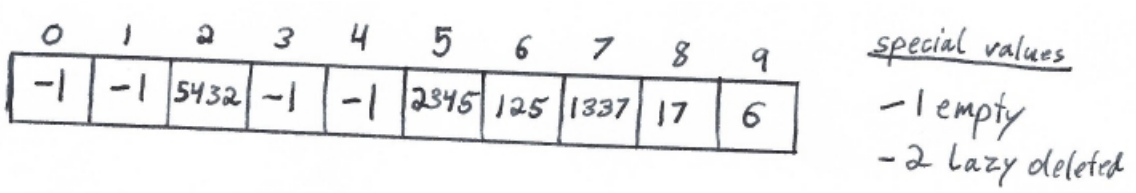
The following shows the result of a **closed addressing** table after the following insertions

Insert 5432
Insert 1337
Insert 2345
Insert 17
Insert 125
Insert 6



Notice, how with the linked list in **closed addressing,** that we insert collision entries at the beginning of the linked list.

The following shows the result of an **open addressing** table using linear probing



Notice how first 5432 was inserted into slot 2, then 1337 into slot 7 and 2345 into slot 5

Insert 17 is a collision with 1337 in slot 7, so we move to the next probing index
p(17,1) = (7 + 1) % 10 = 8 and 17 is inserted in slot 8

Insert 125 is a collision with 2345 in slot 5 so we move to the next probing indexing
p(125, 1) = (5 + 1) % 10 = 6 and 125 is inserted in slot 6

Insert 6 is a collision with 125 in slot 6 so we move to the next problem index until empty
p(6, 1) = (6 + 1) % 10 = 7 full
p(6, 2) = (6 + 2) % 10 = 8 full
p(6, 3) = (6 + 3) % 10 = 9 and 6 is inserted in slot 9

There are 3 common probing functions, linear probing, quadraic probing, and double hashing

Linear Probing

$$p(x,i) = (H(x) + i) \% k$$

Quadratic Probing

$$p(x,i) = (H(x) + ai^2 + bi) \% k$$

Double Hashing

$$p(x,i) = (H_1(x) + H_2(x)i) \% k$$

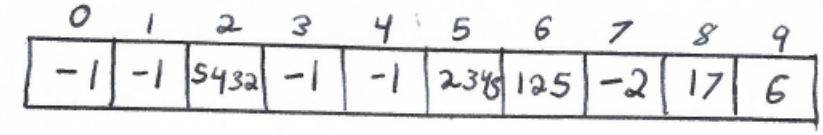
Linear probing is the most common, probing functions p(x,i) describe which slot to guess next if the first hash slot does not contain the desired entry.

Double hashing is a bit unique, because it makes use of two hash functions to determine the next index for probing.

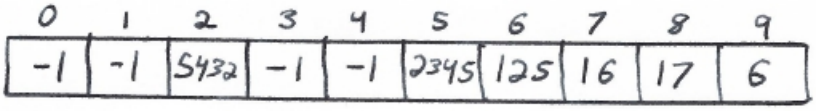
Lazy Deletion

For open addressing we typically delete elements using lazy deletion, where we replace indices with a special marker (in this case -2) to indicate that there is no element in the slot, but it does not break chains for probing.

For example, delete 1337



Followed by insert 16



The following is pseudocode for delete, find, and insert for **open addressing** using the probing functions p(x,i)

```
Delete (x, A, k)
{
    idx = Find (x, A, k)
    A[idx].key = -2
}
```

```
Find (x, A, k)
{
    idx = p(x, 0)
    i = 0
    while (A[idx].key ≠ -1)
    {
        if (A[idx].key = x)
            return idx
        i++
        idx = p(x, i)
    }
    return -1
}
```

```
Insert (x, A, k) value)
{
    idx = p(x, 0)
    i = 0
    while (A[idx].key ≠ -1 && A[idx].key ≠ -2
            && A[idx].key ≠ x)
    {
        i++
        idx = p(x, i)
    }
    A[idx].key = x
    A[idx].val = value.
}
```