CSE 1320 - Intermediate Programming

Hash Maps

Alex Dillhoff

University of Texas at Arlington

Hash Maps

A **hash map** is an abstract **unordered, associative array**.

It represents a mapping between a unique **key** and some associated **value**.

Hash Maps

By **unordered**, we mean that the data is represented in no particular ordering.

An **associative array** is an array consisting of key-value pairs for which each key is unique (e.g. IDs, Addresses, etc.).

Hash Maps

A **key** in a hash map need not be a numerical value.

It can be any variable sequence of characters.

A **hash function** converts the sequence of character to a numerical value which is used to determine the index.

Hash Maps

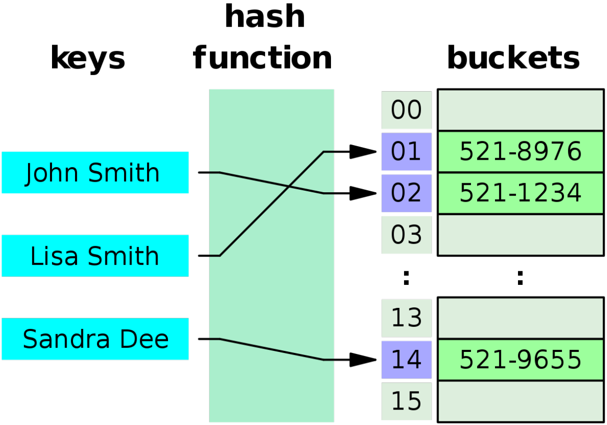

Image from Wikipedia - Hash table.

Hash Function

Ideally, a hash function would assign every key to a unique key.

However, the size of the array (or number of **buckets**) is finite.

Basic Implementation

To implement a basic hash map, you only need two components:

1. An array
2. A hash function

Example: Hash Map for strings

Resolving Collisions

A **collision** occurs if multiple keys map to the same bucket.

This is more common than you might think, regardless of array size.

The Birthday Paradox

A popular problem in probability theory is called **The Birthday Problem**.

What is the probability that some pair of people will share the same birthday in a set of n randomly sampled individuals?

The result is usually surprising at first.

The Birthday Paradox

If only 23 people are randomly sampled, there is an approximately 50% chance that two people will share the same birthday.

If 60 people are sampled, there is a 99.4% chance.

The Birthday Paradox

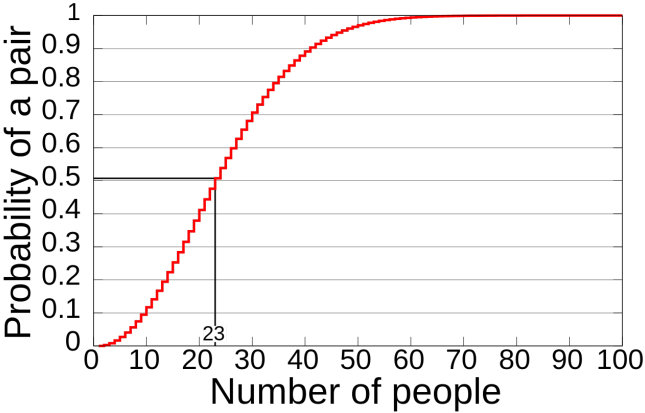

Image from Wikipedia - Birthday problem

Resolving Collisions

How does this apply to hash maps?

With a bucket size of 1,000,000, there is a 95% chance that a collision will occur if only 2,450 keys are hashed.

Resolving Collisions

There are many ways to resolve collisions. They can be broadly grouped into one of two types:

1. Separate Chaining
2. Open Addressing

Separate Chaining

Solutions based on **separate chaining** use an additional data structure for each entry in the array, such as a linked list.

If a collision occurs at the same index, the item is added to a list associated with that bucket.

Separate Chaining

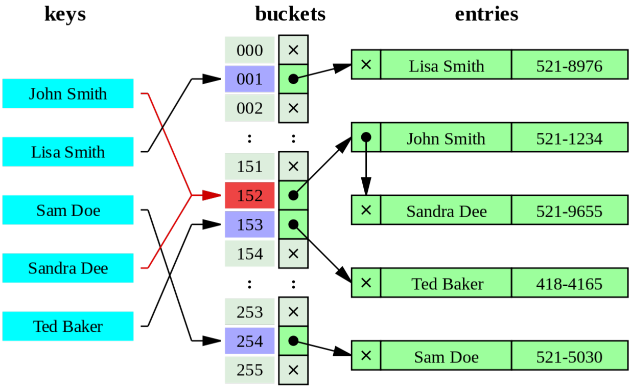

Image from Wikipedia - Hash table

Separate Chaining

There are many data structure choices for implementing separate chaining.

The decision is usually once of memory and compute time.

Usually, the number of items that collide will be relatively small.

Separate Chaining

What are the pros and cons of using each of the following for separate chaining?

- Linked Lists
- Binary Search Trees
- Arrays

Separate Chaining - Linked Lists

If the data that is being stored is small, a linked list might not be a good choice as the overhead of the next pointer will add up.

Separate Chaining - Binary Search Trees

Using a Binary Search Tree has the advantage of searching much more efficient

However, this relies on the tree being balanced. This coupled with other operations like deletion can cause slowdowns depending on the implementation.

Separate Chaining - Arrays

Searching between multiple elements in the same bucket is linear with arrays.

When items are added or removed, the array needs to be resized.

This requires moving elements in the case of removing an item.

Separate Chaining

Example: Separate Chaining with Linked Lists

Open Addressing

Open addressing is arguable more common as it is much easier to implement.

The simplest version of this is called **linear probing**.

Open Addressing - Linear Probing

If a collision occurs in **linear probing**, the table is linearly scanned until the desired entry is found OR an empty entry is found.

In the latter case, no such element exists.

Open Addressing - Linear Probing

Example: Linear Probing

Load Factor

What is the most efficient bucket size to select?

This is an important question when implementing a hash map and can be informed by a property called the **load factor**.

Load Factor

The **load factor** is computed as

$$\textbf{load factor} = \frac{n}{k},$$

where

- n is the number of entries in the hash table.
- k is the number of buckets.

Load Factor

As the load factor increases, there are less buckets available per entry.

Thus, the hash map becomes slower as the lookup operation depends more on collision resolution.

Load Factor

A very low load factor is not necessarily better than a finely tuned one.

If there are many more buckets available than number of entries, this is simply wasted space.

Load Factor

What is the best choice then?

It is usually dependent on your project requirements, but the goal is to keep the expected **constant time** search property.

Some common implementations pick a load factor of around 0.7.

Load Factor

When the upper bound of the load factor is exceeded based on some insertion operation, the map must be resized and **rehashed**.

Rehashing

Rehashing is required when the array is resized.

The naive approach is to simply create a new array (usually double the size of the original) and copy all entries from the original array.

Rehashing

A glaring issue with this method is that, if the size of the array is large, it can be very costly to write a table.

Rehashing

Example: All-at-once Rehashing

Incremental Rehashing

An alternative to creating a new array each time it needs to be rehashed is to resize **incrementally**.

Incremental Rehashing

The first step is to create a new array.

The original map should stay in memory and any new insertions should occur in the new array.

Incremental Hashing

When an insertion occurs, move r elements from the original array to the new one.

If the new array is $\frac{r+1}{r}$ times larger than the original, this will ensure the old array is completely copied before the new array needs to be resized.

Incremental Hashing

During a search or deletion, each array is checked for the entry.

This must happen until all entries from the original have been moved over.

Once the original array is empty, it can be released.

Computing the Resize Factor

When performing an incremental rehash, we want to make sure all data is moved from the old map to the new one before another rehash is required.

As we saw previously, this only requires that we make the new map $\frac{r+1}{r}$ times larger.

Computing the Resize Factor

If we wish to move only a single item at a time, then the new map should be twice as large.

$$\frac{1+1}{1} = 2$$

Computing the Resize Factor

For example, let's say that our original hash map has size 4 with a load boundary of 0.7 and already has 2 entries.

Adding another sample will increase the load factor to 0.75, triggering a rehash.

Computing the Resize Factor

If we use $r = 1$, then the new map size will be 8.

We insert the new entry into the new map while taking one from the old map.

The new map has 2 entries with a load factor of 0.25 while the old map only has a single entry left.

Computing the Resize Factor

We then add one more entry into the new map.

Incremental rehashing will take the last entry from the old map and rehash it into the new one.

We now have 4 samples in the new map with a load factor of $\frac{4}{8} = 0.5$.

Computing the Resize Factor

Since the old map is empty, we would release its memory.

We were able to make sure that all samples from the old map were transferred before hitting the load boundary in the new map.

Computing the Resize Factor

What if we need to bulk import n samples of data into the map at once?

It would not be optimal to insert each new entry one at a time, possibly rehashing multiple times in the process.

Computing the Resize Factor

Consider a hash map of size 4 with 2 samples already in it.

We wish to load in 4 new samples.

A load boundary of 0.7 means that rehashing begins on the first sample added.

If we only doubled the size of the map, we would require another rehash once the last sample is added.

Computing the Resize Factor

We need to consider the relationship between the number of total samples that will be in the map and the load boundary.

$$\frac{n}{k} < b,$$

where n is the number of samples, k is the map size, and b is the load bound.

Computing the Resize Factor

In our example, $n = 6$ and $b = 0.7$.

The chosen value of k should satisfy

$$\begin{aligned} k &> \left\lceil \frac{n}{b} \right\rceil \\ k &> \left\lceil \frac{6}{0.7} \right\rceil \\ k &> \lceil 8.57 \rceil \\ k &\geq 9 \end{aligned}$$