Notes - Algorithms & Data Structures - Hash Tables - Part 2

- Dr Nick Hayward

A brief intro to *hash* tables and their general usage.

Contents

Intro

Hash tables are a particularly useful, and fast, data structure.

From a conceptual perspective, we may define a hash table data structure as follows

- store each item in an easily determined location
    - so no need to search for item
- no ordering to maintain
    - for insertion and deletion of items

As such, this data structure has impressive performance, as far as time is concerned. However, there is a tradeoff with additional memory requirements, and conceptually harder implementation for custom patterns.

Prevent duplicate entries

A key consideration for working with hash tables is the prevention of duplicate entries for data.

For example, consider the following initial scenario for user accounts and registration.

- a new user submits their preferred username
    - username is checked against existing records for user accounts
- if the username already exists - return the user to the registration page & try again...
    - otherwise, allow the user to continue registration

Whilst this sounds like an easy process, we may quickly create a large dataset of user accounts, names, &c.

Using this process, each time a new user submits a registration request the app has to scan the large, and growing, list of users to check for existing usernames.

However, there is a better option using a *hash table*. We can create a new table to keep track of the users and associated usernames.

```
user_accounts = dict()
```

and then check if a username already exists in the table

```
user = user_accounts.get("daisy")
```

In effect, we return data for the queried username

```
# create hash table for address book
user_accounts = dict()

# perform check for passed username
def check_users(name):
    if user_accounts.get(name):
        print("try again - username '" + name + "' already exists...")
    else:
        user_accounts[name] = "active"
        print("user account created...")

# check user accounts
check_users("daisy")
check_users("emma")
check_users("daisy")
```

If we were simply storing such records in a list of users, queries would become very slow as the number of users increases. In effect, we'd need to run a simple search over the entire list.

However, checking for duplicate entries in a hash table is very fast, and well-suited for this type of usage.

Hash tables and caches

Another common use case for hash tables is *caching* with applications.

If we consider a web application, which regularly receives multiple requests for pages, data, and media from both authenticated users and anonymous users.

For example, let's consider a standard usage pattern

- a user submits a request to the web application, which is sent to the defined host server
- the server processes the request, and returns the data and updated page for the web application
- the user views and interacts with this page...

This is a standard, abstracted pattern for such usage, and provides the data and page for the user.

However, we may also find that many users will submit the same requests for data and pages. In effect, the latest weather, news, photos, and so on. Such requests may take a few seconds, perhaps even minutes, to process and return.

This is a common usage scenario for website caching, effectively remembering processed data for submitted queries and requests. This saves repetitive requests and recalculations of data.

We may also see this pattern for authenticated users and anonymous users. A logged-in user may require personalised, tailored data and pages. This will be calculated and returned by the server.

However, anonymous users will see the same page structure and data. In effect, the web application will receive many repetitive requests for the same data and pages, perhaps the user's registration and login page.

To help lessen server usage, the server will simply *remember* such pages for anonymous users, and send the same page.

This *caching* of pages and data has two notable advantages

- the requested web page for the application is returned a lot faster, removing the need for repetitive requests and calculations by the server
- the server and the web application has less work to do...

Such data may be cached, of course, in a hash table. In effect, we may define mapping of URLs from a web app's pages to associated page data.

As a user visits and requests various pages and data, the web app is able to check for cached versions of the page in the hash table. If the page exists, the server will simply send the cached copy for the request to the user.

So, we can obviously see how hash tables are particularly useful for the following

- modeling relationships
- filtering duplicate entries
- caching data

Collisions

To help us better understand the relative merits and performance of hash tables, we need to consider collisions.

Whilst we might strive for an ideal solution where a hash function always maps different keys to different slots in an array, this is not always possible.

For many hash functions, this is simply not possible to achieve.

Let's consider the following initial example, where a simple hash function assigns a spot in the array alphabetically. If we know we only have single items of each letter, then this will work fine. We can assign a single title to a given letter of the alphabet, and maintain fast performance.

However, if we start adding further titles per letter, we encounter the issue of *collision*. In effect, multiple keys are being assigned the same slot in the array. If we continue with the current assignment of slots per letter, we will simply overwrite previous titles with the new title.

In effect, whilst the query may work, the return value will be incorrect.

So, we need to consider a work-around for such collisions.

linked list for collisions

The simplest solution for this issue of collisions is to use a linked list with the hash table.

If multiple keys are mapped to the same slot in the hash table, we may create a linked list at that position.

In effect, *d* may now store multiple records in the hash table by using a linked list as a value in the array.

However, whilst this may be a usable solution for smaller linked lists of records, it is not a fast solution for larger hash tables. We are still restricted by the slower search of the linked list for the chosen letter.

So, we might as well have chosen a linked list instead of a hash table.

considerations

As we can see with the issues of collision, your choice of *hash function* is crucial for performance and maintenance of the hash table.

Ideally, a good hash function will map keys evenly across the hash table.

So, a good hash function will create fewer collisions within the hash table.

Performance for hash tables

As we saw with the *bookshop* example, we commonly need to query data instantly, or at least as far as possible.

This is a real benefit of *hash tables*, they are fast.

Let's consider the following summary *hash table* performance

operation	average case	worst case
search	O(1)	O(n)
insert	O(1)	O(n)
delete	O(1)	O(n)

We can clearly see that for average cases, a *hash table* will have `O(1)`, constant time. Unfortunately, this does not mean *instant* time. However, it does mean that the performance time will stay the same regardless of the size of the hash table.

average case comparison

As a quick comparison, we already know that *simple search* will take `O(n)`, linear time, and binary search takes `O(log n)`, logarithmic time.

If we compared such functionality on graphs we may see a flat horizontal line for a hash table.

However, why is the graph for a *hash table* a flat line?

In effect, it's representative of the underlying nature of the query relative to a hash table. Regardless of the size of the hash table, one element or ten million, we're able to retrieve an element in the same amount of time.

This is the same as querying a known array, which also takes `O(1)`, constant time for indexed queries.

So, we can obviously see that for the *average case* hash tables are very fast.

worst case comparison

If we compare *worst case* performance, we can see that a *hash table* takes `O(n)`, linear time for everything, which is very slow for applications &c.

Again, it's useful to compare this performance against, in this example, arrays and linked lists.

operation	hash table (avg case)	hash table (worst case)	arrays	linked list
search	O(1)	O(n)	O(1)	O(n)
insert	O(1)	O(n)	O(n)	O(1)
delete	O(1)	O(n)	O(n)	O(1)

Consider the average case for hash tables. *Hash tables* are as fast as arrays at searching, i.e. getting an indexed value, and they're also as fast as linked lists for insertion and deletion.

However, the worst case is where we may encounter concerns with *hash tables*. For the worst case, a hash table is slow at each of these operations.

As you might expect, we need to ensure we do not hit the worst case performance for a hash table. A common option for reducing this possibility is to avoid *collisions*.

To help with collision avoidance, we need to check for the following

- low load factor
- good hash function

Load factor

A hash table's *load factor* is straightforward to consider and calculate.

In effect, we use the following

```
number of items in hash table / total number of slots
```

As we've seen, we may use an array for storage of a hash table, thereby allowing us to easily check the number of occupied slots in the array.

For example, consider the following sample basic hash table

```
-----------------------------------
|  3 |     |     |  7 |     |     |
-----------------------------------
```

So, this hash table has a load factor of `2/6`.

Likewise, we can see that the following hash table has a load of `1/3`.

```
-------------------
|    |  9   |     |
-------------------
```

So, *load factor* helps us quickly measure how many empty slots remain in the current hash table.

load factor usage

However, why is this inherently useful or important?

If we have 100 or 200 elements we need to store in a hash table, then we need to know whether that table can efficiently handle the data.

For example, if the table has one hundred slots, then the load factor will be `1`. If the data increases to 200, then the load factor will double to `2`. In effect, we can see how each element will *not* get its own slot in the table.

A load factor greater than `1` is, of course, a bad thing for most cases. We have more elements than space in the table.

As the load factor continues to grow, we need to add more slots to the hash table.

Commonly, as a hash table is reaching capacity load, we need to consider a resize.

Depending on the programming language used for the hash table, we may need to create a larger array for the table. A good heuristic for this increase is to double the array size. In the current example, we can double the size to 200.

Then, we may re-insert all of the existing elements into the new hash table using the hash function.

So, the new hash table will now have a load factor of 100/200 or `0.5`. With this lower load factor, we also reduce the number of collisions in the table. The table should also perform better.

A good heuristic for resizing a hash table is when the load factor is above `0.7`.

However, such resizing may also incur a cost in time and performance. Resizing is expensive, and we need to ensure we do not resize a hash table on a regular basis.

Overall, even with resizes, hash tables still average `O(1)`, constant time.