Notes - Algorithms & Data Structures - Hash Tables - Part 1

- 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.

abstract data type

Storage options and patterns may be described conceptually as an abstract data type.

So, we need to define a specification for this particular abstract data type.

After defining the specification, we may then choose a data structure as the foundation for this implementation.

Table abstract data type

An initial specification for this abstract data type may be outlined as follows

- table may be used to store objects
    - e.g.

id	place	country
5	philae	egypt
21	athens	greece
37	rome	italy
24	sparta	greece

- objects may be arbitrarily complicated, e.g.
    - each object has unique *key*
    - keys may be compared for equality
    - keys used to identify objects
- assume there are methods &c. for the following
    - check for empty or full table
    - insert object into table - assuming table is not already full
    - for a given key - retrieve object for that key
    - for a given key - update object for that key
        - commonly replace current object at key with new object
    - for a given key - delete object for that key
        - assumes key is already stored in table
    - traverse or list each item in current table
        - if order defined - should follow increasing order...

This outline is predicated on a simple initial assumption that each object is uniquely identified by its key.

Implementations of table data structure

We may consider three common approaches for the implementation of a *table data structure*.

For example, we might use one of the following options

- sorted arrays
- binary search trees
- hash tables

sorted array implementation

If we choose a sorted *array* for the *table* data structure, we may determine if it is full or empty in constant time `O(1)` assuming we maintain a variable for its size.

To insert an element, we need to find its correct position. This will take, on average, the same time as finding an element.

To find an element, crucial for all operations except traversal itself, we may use *binary search*. This will take, for example, `O(log n)`, logarithmic time.

Likewise, we may consider the complexity for *retrieval* and *update*. These will also produce `O(log n)`, logarithmic, times.

However, if we need to delete or insert an item, we'll need to shift the following element (to the right of the current node for deletion &c.) either to the left for deletion, or to the right for insertion.

e.g.

```
[3, 6, 2, 33. 17. 97]
```

- delete node `33`
    - element `17` will need to shift to its left
- insert node at position of node `2`
    - element `33` &c. will need to shift to the right

This will take an average `n/2` steps, so such operations will have a complexity of `O(n)`, or linear time.

Ordered traversal is simple for this type of data structure, so we may also see complexity of `O(n)`.

binary search tree implementation

An alternative to sorted arrays might use *binary search trees*.

However, whilst this is certainly possible, its worst case may also produce a tree that is very deep and narrow. Such unbalanced trees will have *linear* complexity for lookups.

There is, of course, a *self-balancing binary search tree*. This is able to produce a worst case that is the same as the average case. For such trees, we commonly see time complexity of `O(log n)`, logarithmic, for insertion, deletion, search, retrieval, and update. We may also see complexity of `O(n)`, linear, for traversal.

However, the downside of such self-balancing trees is the sheer complexity of implementation, management, and initial comprehension.

hash table implementation

Hash tables may provide a benefit for such table data structure usage.

Whilst expending more space than may actually be required or necessary, we may speed up the inherent operations of the table.

We may consider further such usage.

Hash tables

There are many concepts to consider as we review *hash tables*, including

- initial implementation

- collisions
- hash functions
- performance
- ...

However, it is often useful to begin with a conceptual example to help review hash tables, and their underlying functionality.

manage a bookshop

For example, we currently manage a bookshop. It has many valuable first editions, but also some common paperback publications and latest releases.

When someone wants to purchase a book, we need to check the price in a register, which contains the variant prices for editions, publications, and each book in the shop.

However, if the register is not organised in alphabetical order, it may, of course, take a long time to check every entry for the required book. In effect, this would involve a *simple search* where the seller checks every line of the register. A time of `O(n)`, linear time, will not be profitable for the shop.

If the register is ordered alphabetically, however, we may then run a *binary search* to find the required price. Complexity will now fall to a more manageable and useful time of `O(log n)`, logarithmic time.

A quick comparison for searching required items in the register,

items in register	O(n)	O(log n)
100	10 seconds	1 second (7 lines - check $\log_2 100$)
1000	1.66 minutes	1 second (10 lines - check $\log_2 1000$)
10000	16.6 minutes	2 seconds (14 lines = check ($\log_2 10000$)

Whilst *binary search* is a faster option for this type of register, in particular compared to *simple search*, it's still a tad annoying to search through the register for each requested book purchase.

To help manage this register, we might initially consider an array. In effect, each item will need to store the book's title and its price. If we then sort this array by title, we can use binary search to find the associated price. This would, of course, give us a time of `O(log n)`, or logarithmic time.

Ideally, however, we really need a way to query the register and return the price of a book in `O(1)`, constant time. Instead of a default array, we'll try implementing *hash functions*.

Hash functions

We may consider a *hash function* as a simple concept where we input a string, and return an output number.

In this type of conceptual usage, we may define a string as input data for the query as a sequence of bytes. In effect, we may define this usage of a hash function as *mapping strings to numbers*.

To help with this mapping, there are some requirements for a hash function. For example,

- consistency - needs to ensure input string always returns the same number
    - i.e. without this predictable mapping, the hash table will not work...
- mapping - hash function should map different words to different numbers.
    - i.e. the function will be no use if it simply returns `7` for each input string
    - best case will allow every string to map to a different number

basic implementation - input

This example usage will allow us to implement the desired query for the register in our bookshop. In effect, try to achieve querying for a book's price in `O(1)`, constant time.

So, we may begin with an empty array

```
-------------------------
|   |   |   |   |   |   |
-------------------------
| 0 | 1 | 2 | 3 | 4 | 5 |
-------------------------
```

We may use this array to store the bookshop's prices.

We'll start by adding a price for the title **The Glass Bead Game**. We pass this title to the *hash function*, which returns the number `3`. We may then use this number to store the title's price at index 3 in the array.

```
---------------------------
|   |   |   | 7.95 |   |   |
---------------------------
| 0 | 1 | 2 |  3   | 4 | 5 |
---------------------------
```

Then, perhaps, if we input the title **Hannibal's Footsteps** in the hash function, it will return a numerical value of `1`. We may then store the price of this title at index `1` in our current array.

```
-------------------------------
|   | 18.95 |   | 7.95 |   |   |
-------------------------------
| 0 |   1   | 2 |  3   | 4 | 5 |
-------------------------------
```

If we continue this pattern of input, we'll be able to input each title in the bookshop's register in the hash function, and then store the associated price in the array. So, we'll have an array full of prices.

basic implementaton - query

If we then want to retrieve a price for a title in the bookshop's register, we will not need to search through the array.

Instead, we simply pass the title to the hash function, which will return a number. This number will follow the earlier rules, and provide a consistent value for the input title. For example, `3` for the input **The Glass Bead Game**, which we may use to get the price from the array.

In effect, the hash function will tell us exactly where the price is stored in the data structure without the need to search.

This pattern works because the *hash function* adheres to the following requirements,

- it consistently maps the input string to the same number - i.e. the index in the array for the stored value, e.g. price
    - input string once to get the initial number for the index position in the array
    - input string whenever the price is needed for a title from the array
- it maps different strings to different numbers
    - every variant input string will map to a different index position in the array - each price may now be stored in the array...
- it knows the size of the array, i.e. its maximum size, and only returns valid numbers for the index

So, we now have a hash function and an array, which will combine to produce the required data structure, the **hash table**.

In effect, we have a data structure with added logic to consider for its default implementation and usage. This is a noticeable difference with default arrays and lists, which customarily map straight to memory.

The hash table uses its hash function to calculate and record where elements may be stored.

Programming usage

Hash tables are a useful and powerful option for organising data with fast retrieval.

They may also be referenced as *hash maps*, *maps*, *dictionaries*, and *associative arrays*.

Each input query, such as the title of a book from the bookshop's register, will be returned instantly from the underlying array for the hash table (all things being equal for memory usage, system access &c.)

general usage

For most applications, you will not need to implement your own hash tables.

Python, for example, implements hash tables, which are referenced as *dictionaries* in the language.

An example usage will create a new hash table with the function `dict`,

```
bookshop = dict()
```

So, bookshop is a new hash table, which may store book titles and associated prices.

```
bookshop["The Glass Bead Game"] = 7.95
bookshop["Hannibal's Footsteps"] = 18.95
```

This will populate the hash table with the titles and the prices

```
{'The Glass Bead Game': 7.95, "Hannibal's Footsteps": 18.95}
```

To retrieve the price for a stored title in the hash table,

```
print(bookshop["The Glass Bead Game"])
7.95
```

So, we can see the clear mapping of keys to values in the current *hash table*.

usage cases

A common use of hash tables is lookup of associative data sets. For example, a username and ID or name and address &c.

If we consider briefly, for example, an address book, we need to map people's names to addresses, phone numbers, email addresses &c.

In effect, we need a convenient and reliable way to perform the following functionality

- add a name - associate address &c. with that specific name
- enter a name - find and return address details associated with that name

Obviously, we can quickly see how a *hash table* is an ideal option for this type of usage.

We may

- create a map from the name to the address information
- query and return associated data

So, we may create a simple address book using a *hash table* with Python

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

# add some entries and addresses
```

```
address_book["daisy"] = "dawlish"
address_book["emma"] = "cannes"

# check return of address for daisy...
print(address_book["daisy"])
```

This type of lookup is used at a larger scale for various real-world uses. For example, each time we perform a query to a domain name, we need to query the IP address for the host server. In effect, the URL is translated to an IP address from a lookup.

This lookup process is known as *DNS resolution*. Hash tables are one way to provide this type of functionality.