Comp 363 - Design and Analysis of Computer Algorithms

Spring Semester 2020 - Week 10 - Part 2

Dr Nick Hayward

Video - Algorithms and Data Structures

The Joy of Data - Packet Switching

The Joy of Data - Packet Switching - UP TO 41:40

Source - BBC - The Joy of Data - YouTube

hash tables - prevent duplicate entries - part 1

- key consideration for working with hash tables
  - *prevention of duplicate entries for data*
- e.g. consider initial scenario for user accounts and registration
  - *new user submits preferred username*
    - username is checked against existing records for user accounts
  - *if username already exists*
    - return user to registration page & try again...
  - *otherwise*
    - allow user to continue registration
- sounds like an easy process
  - *quickly creates a large dataset of user accounts, names, &c.*
- each time a new user submits a registration request
  - *app has to scan large, growing list of users to check for existing usernames*

Algorithms and Data Structures

hash tables - prevent duplicate entries - part 2

- better option using *hash table*
  - *create new table to keep track of users and associated usernames*

```
user_accounts = dict()
```

- then check if username already exists in table

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

hash tables - prevent duplicate entries - part 3

- return data for queried username

```python
# 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 store such records in a list of users
  - *queries become very slow as number of users increases...*
  - *i.e. need to run a simple search over entire list*
- checking for duplicate entries in a hash table is very fast
  - *well-suited for this type of usage*

Algorithms and Data Structures

hash tables - caches - part 1

- another common use case for hash tables is *caching* with applications
- consider a web application
  - *regularly receives multiple requests for pages, data, and media*
  - *requests from both authenticated users and anonymous users*
- e.g. consider a standard usage pattern
  - *user submits request to web application - sent to defined host server*
  - *server processes request - returns data and updated page for web application*
  - *user views and interacts with page...*
- a standard, abstracted pattern for such usage
  - *provides data and page for user*
- may also find many users submit same requests for data and pages
  - *e.g. latest weather, news, photos...*
- requests may take a few seconds, perhaps even minutes, to process and return
- common usage scenario for website caching
  - *i.e. remembering processed data for submitted queries and requests*
  - *saves repetitive requests and recalculations of data*

hash tables - caches - part 2

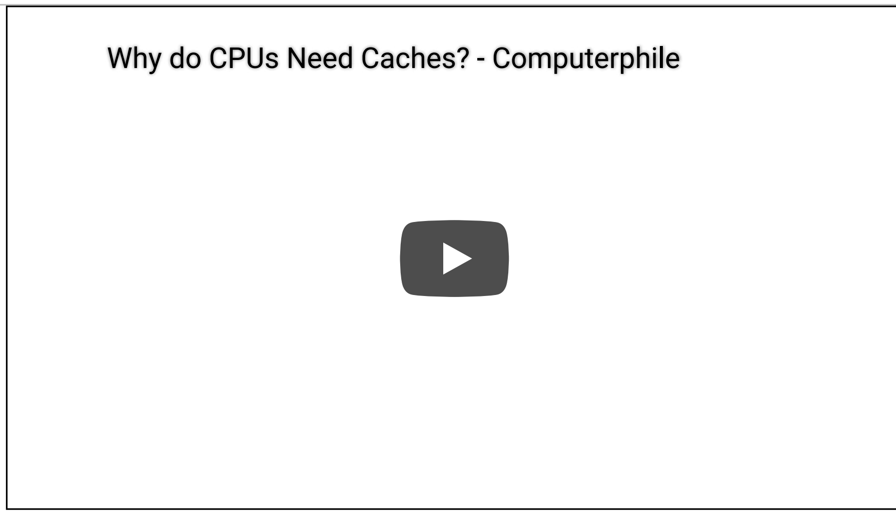
- similar pattern for authenticated users and anonymous users
  - *logged-in user may require personalised, tailored data and pages*
  - *calculated and returned by the server*
- anonymous users will see same page structure and data
  - *i.e. web application receives repetitive requests for data and pages*
  - *e.g. user's registration and login page*
- help lessen server usage
  - *server remembers such pages for anonymous users*
  - *sends same page...*
- *caching* of pages and data has two notable advantages
  - *requested web page for application returned faster*
    - removes need for repetitive requests and calculations
  - *server and web application has less work to do...*

hash tables - caches - part 3

- data may be cached in a hash table
  - *i.e. define mapping of URLs from web app's pages to associated page data*
- as user visits and requests various pages and data
  - *web app checks for cached versions of page in hash table*
  - *if page exists - server sends cached copy for request to user*
- hash tables are particularly useful for the following
  - *modeling relationships*
  - *filtering duplicate entries*
  - *caching data*

Video - Algorithms and Data Structures

cache and systems



What’s a cache for? - UP TO 4:08

Source - What’s a cache for? - YouTube

hash tables - collisions - intro

- better understand relative merits and performance of hash tables
  - *need to consider collisions*
- might strive for an ideal solution
  - *i.e. hash function always maps different keys to different slots in array*
  - *n.b. not always possible*
- for many hash functions, simply not possible to achieve

hash tables - collisions - example

- consider initial example
  - *simple hash function assigns data in array alphabetically*
- for single items of each letter
- this function will work fine
  - *i.e. assign a single title to a given letter*
  - *maintains fast performance*
- if we start adding further titles per letter
  - *encounter issue of collision*
  - *i.e. multiple keys assigned same index in array*
- if we continue with current assignment of index per letter
  - *overwrite previous titles with new title*
  - *ie. query may work but return value will be incorrect*
- need to consider a solution for such collisions

hash tables - collisions - linked list

- simplest solution for this issue of collisions
  - *use a linked list with hash table*
- e.g. if multiple keys are mapped to same slot in hash table
  - *create a linked list at that position*
- i.e. *d* may store multiple records in hash table
  - *using a linked list as a value in the array*
- a working solution for smaller linked lists of records
  - *not a fast solution for larger hash tables*
  - *still restricted by slower search of linked list for chosen letter*

hash tables - collisions - considerations

- *collision* demonstrates importance of chosen *hash function*
    - *crucial for performance and maintenance of hash table*
- good hash function will map keys evenly across hash table
- good hash function will create fewer collisions within hash table

hash tables - performance

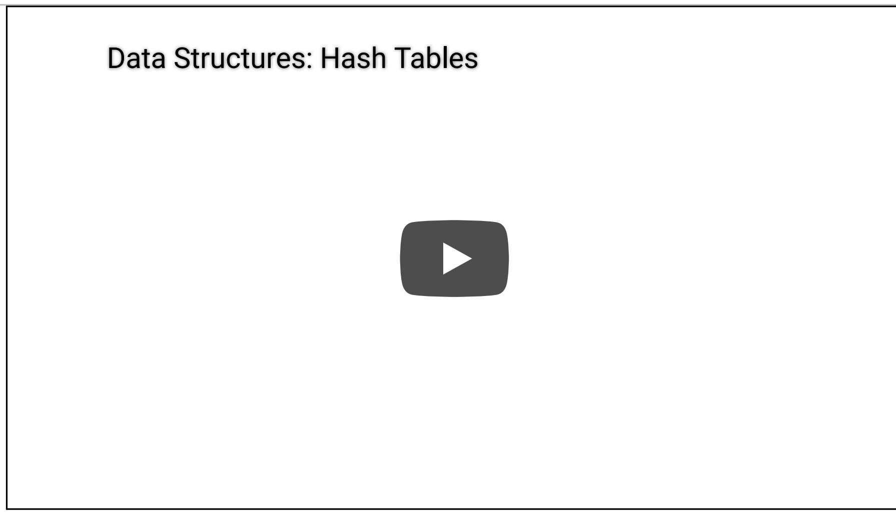
- *bookshop* example demonstrated
  - *need to query data instantly*
  - *i.e. at least as far as possible...*
- a real benefit of *hash tables* is their performance
- e.g. summary of *hash table* performance

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

- for average cases
  - *hash table is `O(1)`, constant time*
  - *n.b. does not mean instant time*
  - *means performance time will stay same regardless of hash table size*

Video - Algorithms and Data Structures

hash tables - part 4



Hash tables - performance - Java - UP TO 6:08

Source - Hash tables - performance - YouTube *hash tables - performance - average case comparison*

- quick comparison
  - *simple search will take* `O(n)`*, linear time*
  - *binary search takes* `O(log n)`*, log time*
- compared such functionality on graphs
  - *may see a flat horizontal line for a hash table*
- why is graph for a *hash table* a flat line?
  - *representative of underlying nature of query relative to a hash table*
- i.e. regardless of size of hash table
  - *e.g. one element or ten million*
  - *able to retrieve element in same amount of time*
- same as querying a known array
  - *also takes constant time for indexed queries*
- for *average case*
  - *hash tables are very fast...*

Algorithms and Data Structures

hash tables - performance - worst case comparison

- compare *worst case* performance
  - *hash table takes `O(n)`, linear time for everything*
  - *very slow for applications &c.*
- useful to compare this performance
  - *e.g. against 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)

- average case for hash tables
  - *Hash tables are as fast as arrays at searching*
  - *i.e. getting an indexed value*
  - *also as fast as linked lists for insertion and deletion*
- worst case may raise concerns with *hash tables*
- for worst case
  - *hash table is slow at each of these operations*
- need to ensure we do not hit worst case performance for hash table
  - *common option for reducing this possibility is to avoid collisions*
- help with collision avoidance
  - *low load factor*
  - *good hash function*

hash tables - load factor - part 1

- hash table's *load factor* is straightforward to consider and calculate
  - *i.e consider the following*

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

- may use array for storage of a hash table
- allows us to easily check number array usage...

hash tables - load factor - part 2

- e.g. consider basic hash table

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

- this hash table has a load factor of `2/6`
- following hash table has a load of `1/3`

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

- *load factor* measures usage and capacity of current hash table *hash tables - load factor usage - part 1*

- why is this inherently useful or important?
- e.g. if we have 100 or 200 elements
  - *need to store in a hash table*
  - *need to know if that table can efficiently handle data*
- e.g. if table has one hundred slots,
  - *load factor will be 1*
- if data increases to 200
  - *load factor will double to 2*
  - *i.e. each element will not get unique slot in table*
- load factor greater than 1
  - *poor usage for most cases*
  - *i.e. more elements than space in table*
- as load factor continues to grow
  - *need to add more slots to hash table*

hash tables - load factor usage - part 2

- as hash table is reaching capacity load
  - *need to consider a resize*
- depending on programming language used for hash table
  - *may need to create a larger array for table*
  - *good heuristic for increase is to double array size*
  - *e.g. double size to* `200`
- then re-insert existing elements into new hash table using hash function
- new hash table has an improved load factor
  - *i.e.* `100/200` *or* `0.5`
  - *lower load factor reduces number of collisions in table*
  - *table should also perform better*
- good heuristic for resizing a hash table
  - *when load factor is above* `0.7`
- resizing may incur a cost in time and performance
  - *resizing is expensive*
  - *need to ensure we do not resize a hash table on a regular basis*
- even with resizes - hash tables still average `O(1)`

Resources

videos

- BBC - The Joy of Data - YouTube
- Hash tables - Java - performance - YouTube
- What’s a cache for? - YouTube