Hashing

December 6th
#CS136#Datastructures#AdvancedProgramming
#Bills
Administrative Details

• No lab today

• Practice exam, study guide will be posted online
  • Don’t panic: much longer than our exam will be!

• TAs available this weekend (see calendar)

• Bill² Review: Tuesday @ 1:30-2:30pm in Physics 205
Today’s Outline

• More applications of Hashing!
  • Cuckoo hashing
  • Sets/Membership Queries
  • Checksums/Integrity
  • Duplicate Detection

• (new material not on CS136 exam)
Quick Hash Table Review

• A hash function maps a key to an index

• The index specifies a hash table bin where the key-value pair should be stored.

• Assuming:
  • Computing the hash function is O(1)
  • Our hash function evenly distributes objects
  • We have a reasonable load factor
  • Bins have O(1) random access (e.g., an array)

• We can get/put key-value pairs in O(1) time!!!
Problems?

• Typically, the domain (set of possible keys) is larger than the range (possible of hash function outputs)

   All Possible Strings (Domain)

   32-bit Integers (Range)

• Multiple keys will map to the same bin
Managing Collisions

• **Collision**: two keys map to the same bin

We can minimize cost of collisions in a few ways:
- Use a hash function that uniformly distributes keys across the range
- Keep the **load factor** low
- Use an array with a (relatively) prime-number-length
  - Why?
    - Consider this String hash function:
      \[ h(s) = s[0] + k^1 \cdot s[1] + k^2 \cdot s[2] + \ldots k^{n-1} \cdot s[n] \]
    - Strings with the same \( s[0] \) hash the same modulo \( k \).
Techniques to Resolve Collisions

• **Linear Probing**
  - When something else is in our bin, scan and insert into the first bin without an element
  - When we delete a key-value pair, drop a placeholder to note that other elements may have been shifted past the newly “emptied” bin

• **External Chaining**
  - Instead of key-value pairs, each bin holds a list
  - To insert: place a key-value pair at end of its bin’s list
  - Downside: extra space required to store lists
New Technique: Cuckoo Hashing
Techniques to Resolve Collisions

- **Cuckoo Hashing**
  - Select 2 independent hash functions
    - A key can now land in 1 of 2 places
  - Resolve collisions by “pushing” others out of our bin and placing them in the bin associated with their other hash
  - The process may need to repeat

- What happens when we:
  - put(X) where hash\(_1\)(X) = 0?
  - put(Y) where hash\(_1\)(Y) = 7?

We must avoid cycles!
Cuckoo Hashing

• For independent hash functions and low load factor, $O(1)$

• No runs like we have with linear probing
  • No shifting “down the line” on inserts
  • At most 2 checks per lookup
Membership Queries
Memory Hierarchy

• **Problem 1:** Sometimes (almost always?) we have more data than fits in memory

• **Solution:** Store a subset of our data in a cache

  • When we need something that isn’t in cache, we kick out the least valuable things to make room for the thing we need
Memory Hierarchy

- **Problem 2:** Not all levels in our cache have the same cost
Memory Hierarchy

- **Problem 2:** Not all levels in our cache have the same cost
Memory Hierarchy

- **Problem 3:** Not all levels in our cache have the same speed
Memory Hierarchy

- Result: we have a lot of slow, cheap storage, less RAM, and very little CPU cache.
- We will focus on the interaction between RAM and disk
Scenario: Photo Storage

Suppose:

• We have a small RAM cache that holds 2 photos

• Our cache is initially empty

• We read from disk into cache, and evict the least recently used photo when we need space
Memory Hierarchy

Small, fast

Big, slow
Memory Hierarchy

get(cat)

Small, fast

Big, slow

?
Memory Hierarchy

get(cat)
Memory Hierarchy

get(cat)
get(cow)
Memory Hierarchy

get(cat)
get(cow)

Small, fast
RAM

Big, slow
Memory Hierarchy

get(cat)
get(cow)
get(dog)

Small, fast

Big, slow
Memory Hierarchy

get(cat)
get(cow)
get(dog)
Memory Hierarchy

- get(cat)
- get(cow)
- get(dog)
- get(goat)

Small, fast

Big, slow
Memory Hierarchy

get(cat)
get(cow)
get(dog)
get(goat)

Small, fast

Big, slow
Memory Hierarchy

get(cat)
get(cow)
get(dog)
get(goat)
get(cat)
Memory Hierarchy

get(cat)
get(cow)
get(dog)
get(goat)
get(cat)

Small, fast

Big, slow
Memory Hierarchy

get(cat)  get(cow)  get(dog)  get(goat)  get(cat)  get(liger)

Small, fast

Big, slow
Memory Hierarchy

cat
get(cow)
get(dog)
get(goat)
get(cat)
get(liger)

Small, fast

Big, slow

???
Memory Hierarchy

• **Problem:** We paid an expensive cost just to find out the thing we were looking for didn’t exist!!

• **Idea:** Cache a set of all the keys (names of all photos on disk)

  1. Check the names set first *before* checking disk

  2. Don’t go to disk if we know the thing isn’t there
Membership Queries

• How to implement our name set?
  • If we want to look things up quickly, use a hash set

• If we want to avoid collisions:
  • Make it big
  • Use a large hash so to uniquely fingerprint each file \( P(\text{collision}) == \text{small} \)

• **New problem**: keys can be long, fingerprints are large. Now our set takes up a large portion of our cache
Membership Queries

• **Insight**: we don’t need to be perfect.

• If we go to disk an extra time, no worse off
  • False positives are not ideal, but they are OK

• If we don’t go to disk when something exists, BAD (or sick)
  • False negatives are correctness bugs, not OK

• We will build a structure that does **approximate membership queries** and is more efficient than a set.
Bloom Filter

• Answers with “possibly in set” or “definitely not in set”
• We save space by not explicitly storing hashes or keys

• How it works:
  • Create a bit array of $m$ bits
  • Select $k$ hash functions
  • Hash each element $k$ times and set all $k$ bits
  • An element is missing if any of its $k$ bits is unset
  • An element may be present if all of its $k$ bits are set
Bloom Filters

Insert(key):

    for hashFunction_i in hashFuncions_{i...k}:
        bitmap[hashFunction_i(key) % m] = 1

Query(key):

    for hashFunction_i in hashFuncions_{i...k}:
        if (bitmap[hashFunction_i(key) % m] != 1):
            return "not in set"
    return "maybe in set"
Bloom Filters

- Deleting keys?
  - A key maps to $k$ bits, and although setting any one of those $k$ bits to zero would remove that key from the set, it will also remove every key that maps to one of those bits.
  - Deleting would introduce false negatives!

- Resizing Bitmap?
  - No way to grow array using just the bit values
  - Although keys are not stored, they are often available
  - When the false positive rate gets too high (overloaded, too many “deletes” still in bitmap), read keys from slower media and resize+rehash
Related DS: Quotient Filters

• A nifty idea with an even nifty-er paper name (Don’t Thrash: How to Cache your Hash in Flash)
• Uses linear probing to support efficient deletes and merges
• “Write-optimized” data structure (my research area)
• Based on an end-of-chapter problem in an undergraduate data structures textbook
  • You can publish a paper with the skills you already have!
  • (and if you were like Bloom, you could name it after yourself)
Integrity/Tamper Evidence
Detecting Changes

- Sometimes we can’t trust the integrity of our stuff
  - Our laptop is from 2006, and our HDD is dying…
  - We store our data in “the cloud” and we don’t trust “the man”
- We live in a place with government censorship and we want to ensure no one has modified a document
- We download something from the internet and we are afraid a “man-in-the-middle” has given us a decoy or a virus
- We are a multi-national company that wants to verify that people pay for official software/media (DRM)
Detecting Changes

- **Observation:** cryptographic hash functions have the following properties
  - Deterministic
  - Non-invertible (given $\text{hash}(x)$ impractical to find $x$)
  - Large Range (many bits in hash)
  - Evenly distributed

- **Insight:** If we pick a good enough hash function, we can trust it to uniquely identify the contents

- (related ideas: checksumming/fingerprinting)
Detecting Changes

- Calculate a fingerprint (cryptographic hash) of objects that we store, and we keep the fingerprint safe
- If we later retrieve the thing we stored, recompute the fingerprint
  - If they match, we are (almost) guaranteed to be safe
  - If they differ by even one bit, there is a problem
Detecting Changes

- Download verification (ubuntu .iso example)
- Scanning files for errors
- Git
- ...
Detecting Duplicates
Deduplication

- Imagine you are a cloud storage provider, and someone uploads the hit song Shoot_Pass_Slam.mp3.
- Millions of other people will as well (Shaq Diesel went platinum after all).
- Do we really need to store millions of copies of the same file?
  - NO! Hash tables/sets can map duplicate keys to the same value.
  - Map every file called “Shoot_Pass_Slam.mp3” to the same file contents.
- What if the file names different?
Deduplication

Instead of mapping:

```
file_name  ->  file_contents
```

map:

```
file_name  ->  hash_of_contents
```

Then have a separate key-value store mapping:

```
hash_of_contents  ->  file_contents
```

**Insight:** many problems in computer science can be solved with a layer of indirection!
Deduplication

• What if we aren’t storing music, but file that are actively modified?
  • We may not want to deduplicate at the coarse granularity of whole files

• Instead, break a file into chunks, and deduplicate chunks
  • Now we map:
    \[\text{file}_\text{name} \rightarrow \text{recipe}\]

* A recipe contains (file offset, chunk length, fingerprint) triples

• We only store one copy of unchanged chunks!
Summary

• Hashing is a powerful technique with many uses
• We can build interesting new data structures
• We can add new twists to existing data structures
• We must be careful to use the right hash function for the task