Hash Tables

Data Structures

Anime Japan 2016 (Tokyo) – check-in line

Why are there multiple lines?

Standard convention check-in system - "bins" of lines

A-J

Anderson, S.

James, B.

Johnson, R.

Banks, I.

K-Z

Kristol, B.

Zappa, F.

Young, N.

Unbalanced convention check-in system

A-Y

Anderson, S.

James, B.

Johnson, R.

Banks, I.

Kristol, B.

Young, N.

Z

Zappa, F.

Issues with this system?

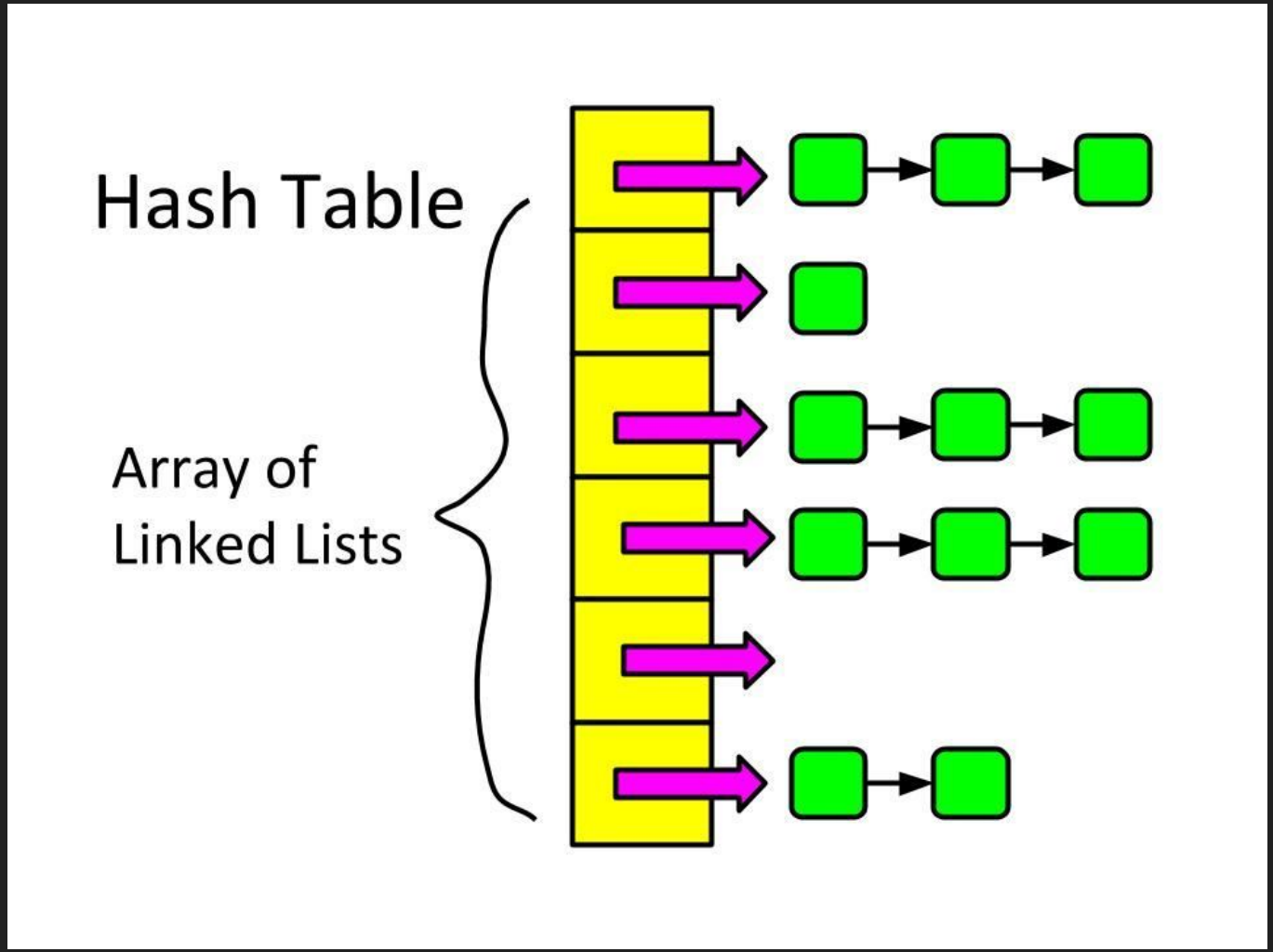
- More lines would be faster?
  - A-F, G-J, K-Z ?
  - Would 26 lines be 26 times faster?
- Unbalanced Demographics?
  - "Johnson Family Reunion"
  - the "J" line is probably going to be longer than all the rest…

- Deciding what "bins" to separate your lines into determines how many bins you have, and how long the lines get.

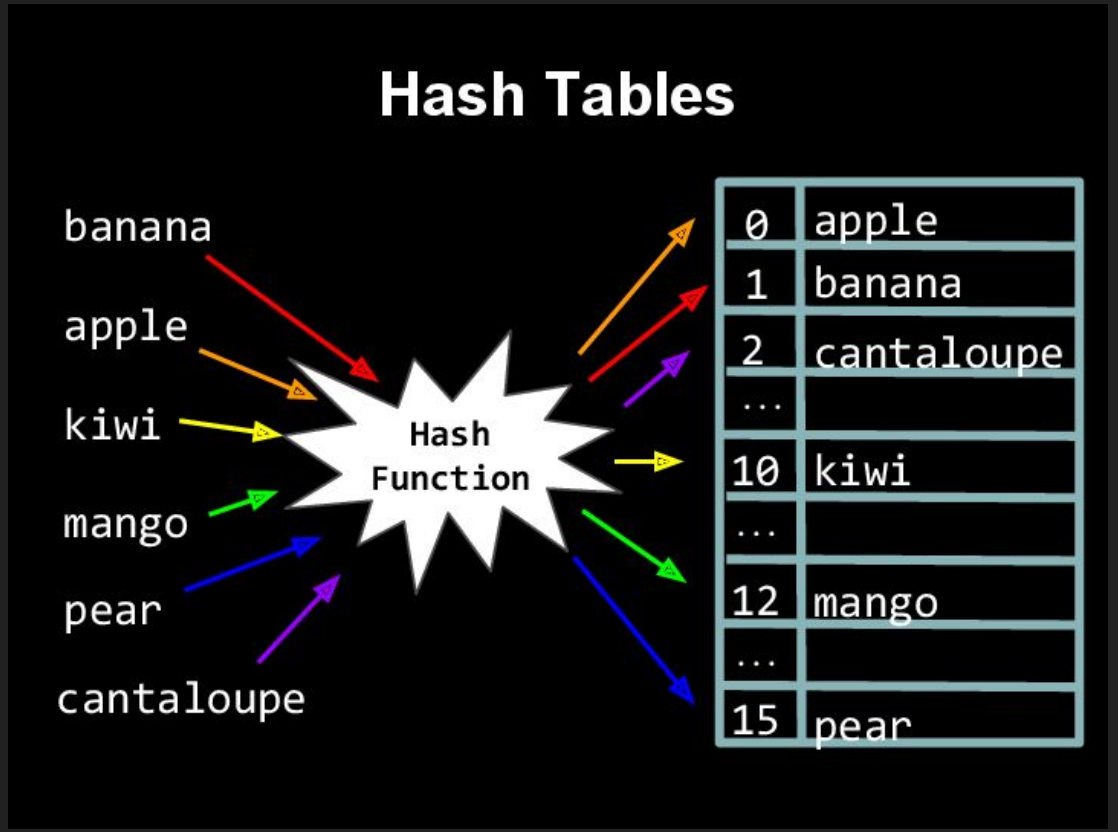
Tradeoff between:

- wasted bin allocation *(empty lines)*
- high collision bins *(long lines)*

Now think of the individual lines as linked lists, and the collection of lines as an array, and you have the **hash table** data structure.



Chaining Hash Table



Hash functions can also work with non-numeric (string) keys

O(1)

In theory…

collisions are rare enough that accessing an item by its key takes on the order of the time it would take using an array.

Hash Functions

must fulfill the following conditions:

- **Deterministic** (always maps to the same bin)
- **Predefined range for a given size** (always maps a given key into the range [0, b-1]

Simple Hash Functions (inefficient)

`map(name) -> Bin: (A-J), (K-Z)`

- 2 bins, so two linked lists
- O(n/2) is still O(n)...
- but conventions have finite # of people attending, so twice as fast is better than nothing.
- 3-4 lines seems to be the sweet spot

`map(id) -> Bin: id % 10`

- works with any size student id
- 10 bins, so 10 linked lists
- depending on how ID numbers are assigned, it's possible this won't help at all (what if they all end in zero?)

Handling Collisions

Chaining Hash Tables:

Collisions are stored in the same bin, in a linked list.

This is the only type of hash table we will focus on in this class.

May not be as efficient as probe-based HTs, but the easiest to understand.

Probe-based hash tables:

If a collision would occur, store the value in a different, related bin (found by **"probing"**).

- Linear probing -> store in (bin + 1, +2,...)
- Quadratic -> based on equation, store in bin X
- etc.

The implementation differs, but the interface is the same.

Common uses of Hash Tables

- Python **`dictionary`** and C++ **`map`** data structures
  - store a value under a string based key
  - an old-school name for this structure is "associative array"
  - example:
    - `definition_of["doe"] = "a deer, a female deer"`
    - `definition_of["far"] = "a long long way to go"`
- Data storage where retrieval in near-constant time is preferred
  - If you have one bin for every possible key, you have an array (but if you're storing 6 digit numbers, you need enough memory for a million bins, even if most are empty…)
  - If you only have one bin, you have a linked list.
  - Efficient implementation of a hash table leads to performance close to that of an array, with much less memory used, if most possible keys are unused.

Hashing in Cryptography

Common problem:

- No system is 100% secure
- People reuse passwords (no matter how many times you tell them not to)

How to prevent passwords stolen from one site from being used to break into accounts elsewhere?

Ashley Madison

in 2015, didn't turn out to be as "anonymous" as users were hoping.

Retrospective by KrebsOnSecurity:

https://krebsonsecurity.com/2022/07/a-retrospective-on-the-2015-ashley-madison-breach/

Leaks revealed customer billing information (names, address, phone numbers), and some users were blackmailed using this information.

Could it have been worse?

A number of compromised systems have had their user databases leaked.

If those databases store passwords in "cleartext", then malicious actors now have:

- usernames (typically email addresses)
- a password they used on one site

A typical vector of attack would be to attempt to log in with that address and password on other sites, such as online banking.

Password Hashing

So now we know: never store passwords in "cleartext" (unencrypted).

One other feature of hash functions is that they are **one-way**.

- Hashing a key always returns the same hash value.
- But this value is not enough information to "recover" the original key.

Therefore,

- storing a **hash** of the password allows you to find out "is the password they typed in == the password they registered with?"
- without storing the password itself.

Password Hashing

The algorithm (python-ish):

```
username = input("enter username:")

login_pw = input("enter password:")

login_hash = hash(login_pw)

stored_hash = user_database.retrieve_hash(username)

if login_hash == stored_hash:

    correct_login = True
```

Password Hashing

The algorithm (c++-ish):

```
string username, login_pw;
cin << username << login_pw;
login_hash = hash(login_pw);
stored_hash = user_database.retrieve_hash(username);
if (login_hash == stored_hash) {
    correct_login = true;
}
```

Hashing and Salting

(but not smothered and covered)

(pictured: Waffle House menu)

Hashing and Salting

https://hashtoolkit.com/common-passwords/

Even though hash functions are **one way**, you can hash a lot of different potential passwords and then see if any of the stored hashes match those.

Password	MD5	SHA1
123456	e10adc3949ba59abbe56e057f20f883e	7c4a8d09ca3762af61e59520943dc26494f8941b
password÷NO PASSWORDS!	320157b0a9d971845c5b0a0796058c79	d8be2ef007bd17d46b3cd126861cd58655a40c33
12345678	25d55ad283aa400af464c76d713c07ad	7c222fb2927d828af22f592134e8932480637c0d
qwerty	d8578edf8458ce06fbc5bb76a58c5ca4	b1b3773a05c0ed0176787a4f1574ff0075f7521e
123456789	25f9e794323b453885f5181f1b624d0b	f7c3bc1d808e04732adf679965ccc34ca7ae3441
12345	827ccb0eea8a706c4c34a16891f84e7b	8cb2237d0679ca88db6464eac60da96345513964
1234	81dc9bdb52d04dc20036dbd8313ed055	7110eda4d09e062aa5e4a390b0a572ac0d2c0220
111111	96e79218965eb72c92a549dd5a330112	3d4f2bf07dc1be38b20cd6e46949a1071f9d0e3d
1234567	fcea920f7412b5da7be0cf42b8c93759	20eabe5d64b0e216796e834f52d61fd0b70332fc
dragon	8621ffdbc5698829397d97767ac13db3	af8978b1797b72acfff9595a5a2a373ec3d9106d

Hashing and Salting

The solution is to add a **salt** to your hash.

- User creates account: enters username, password
- Generate a random "salt" (a string), add that to the password.
- Hash the combined string and store that as the hash.
- Store the salt as well.

On login:

- user enters their password
- load the salt, add that to the password.
- Hash the combined string.
- Compare to the stored hash.

Hashing and Salting - Creating account

Example:

I register as user "bobdobbs" with password "123456".

The system generates the random salt "$.Qwe4k" (this is different every time)

```
hashed_pw = hash ("123456$.Qwe4k")
```

(We'll pretend that results in "HASH_HERE".)

We store these values in the database:

```
{   "username": "bobdobbs",
    "hashed_pw: "HASH_HERE",
    "salt": "$.Qwe4k"
}
```

Hashing and Salting - Login to account

Example:

I attempt to log in as:

- user "bobdobbs"
- password "123456".

The system loads user "bobdobbs" from the database, and gets the stored hash and the salt.

```
{    "username": "bobdobbs",
     "hashed_pw: "HASH_HERE",
     "salt": "$.Qwe4k"
}
```

We hash "123456" + "$.Qwe4k"

If this results in the same value as what is stored in hashed_pw,

then the user has entered the correct password.

Benefits of Salting

Now, in order to "reverse engineer" someone's password, it is no longer sufficient to run the hash function on every possible password they might use.

You would now have to run the hash function on EVERY COMBINATION of:

- any password they might use
- + any potential salts

With a salt of "00-99", this makes the password 100 times harder to crack.

With 6 characters * (26 + 26 + 10) (uppercase, lowercase, and digits):

6 * 26 * 26 * 10 = 40,560 additional hashes must be run **per password**.

Deprecation of MD5 and SHA-1

Modern production systems use hash functions such as SHA-256, which are intentionally time consuming to run (which doesn't impact efficiency – what's one more second per user registration?)

As a result, running hash functions like SHA-256 for every possible combination of password + salt is not going to happen

(unless maybe you work for "No Such Agency.")

Implementation: Horner's Method

```
# just adding each letter's ASCII value isn't efficient
# so we multiply the sum by a constant prime number
StringHash (String: key, Integer: size):
    Integer: total = 0
    for each character in key:
        total = CONST * total + CharToNum(character)
    return total % size
```

Python Reference Implementation

Video walkthrough: https://youtu.be/ctpCl2c9U04

Source code:

https://github.com/norrisaftcc/csc249/tree/main/m5-hashing/hashing_and_salting_example

No hashing: https://replit.com/join/bhevddiwkn-ftccprogramming

Hash, no salt: https://replit.com/join/fhbrlpxuvu-ftccprogramming

Hash and salt: https://replit.com/join/fhbmfxddds-ftccprogramming

Hash Functions in standard libraries

The slides provide pseudocode for a simple string to number hash , but for this program we can use hash functions provided in our language's standard library.

For C++, this is **std::hash**. Information here: https://en.cppreference.com/w/cpp/utility/hash

For python, it's **hashlib**. https://docs.python.org/3/library/hashlib.html