Lecture 28

Hashing & Hash Tables

- Hashing
  - Fingerprints
  - Applications
  - Hash Function
- Hash Tables
  - Open Addressing
  - Chaining
  - `structure` Package
  - Applications

Hashing

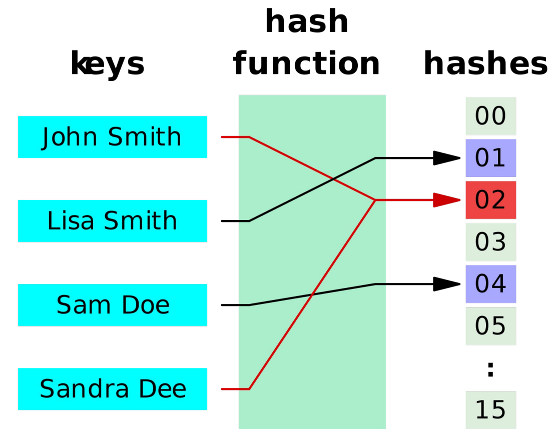
Hashing and Hash Tables

Previously, we saw how ordered keys allow dictionaries to outperform arbitrary maps.

Now we'll consider another improvement upon an arbitrary map.

This improvement involves computing a *hash* of each key.

A *hash* is a somewhat random and efficient meal.



A hash function.

A *hash table* orders keys absolutely (rather than relatively) by their hash value.

A nice analogy for hash functions is *fingerprinting*.

Fingerprints

Fingerprint History

The history of fingerprinting dates back thousands of years, with modern advances beginning in the late 19th century.

~1000-2000 B.C. - Fingerprints were used on clay tablets for business transactions in ancient Babylon.

3rd Century B.C. - Thumbprints begin to be used on clay seals in China to “sign” documents.

610-907 A.D. - During the T’ang Dynasty, a time when imperial China was one of the most powerful and wealthy regions of the world, fingerprints are reportedly used on official documents.

1st Century A.D. - A petroglyph located on a cliff face in Nova Scotia depicts a hand with exaggerated ridges and finger whorls, presumably left by the Mi'kmaq people.

1882 - Gilbert Thompson, employed by the U.S. Geological Survey in New Mexico, uses his own fingerprints on a document to guard against forgery. This event is the first known use of fingerprints for identification in America.

1883 - “Life on the Mississippi,” a novel by Mark Twain, tells the story of a murderer who is identified by the use of fingerprints. His later book "Pudd'n Head Wilson” includes a courtroom drama involving fingerprint identification.

1888 - Sir Francis Galton’s began his study of fingerprints during the 1880s, primarily to develop a tool for determining genetic history and hereditary traits. Through careful study of the work of Faulds, which he learned of through his cousin Sir Charles Darwin, as well as his examination of fingerprints collected by Sir William Herschel, Galton became the first to provide scientific evidence that no two fingerprints are exactly the same, and that prints remain the same throughout a person’s lifetime. He calculated that the odds of finding two identical fingerprints were 1 in 64 billion.

1892 - Galton’s book “Fingerprints” is published, the first of its kind. In the book, Galton detailed the first classification system for fingerprints; he identified three types (loop, whorl, and arch) of characteristics for fingerprints (also known as minutia). These characteristics are to an extent still in use today, often referred to as Galton’s Details.

http://www.fingerprintamerica.com/fingerprinthistory.asp

Fingerprint Types

Fingerprints can be divided into different basic types, and then further divided into subtypes.

Arch
5% of prints

Loop (right)
60-65% of prints

Whorl
30-35% of prints

Tented Arch
a special type of arc

Numerical Classification of Fingerprints

Mapping fingerprints to numbers.

Classifying

Before computerisation, manual filing systems were used in large fingerprint repositories. Manual classification systems were based on the general ridge patterns of several or all fingers (such as the presence or absence of circular patterns). This allowed the filing and retrieval of paper records in large collections based on friction ridge patterns alone. The most popular systems used the pattern class of each finger to form a key (a number) to assist lookup in a filing system. Classification systems include the Roscher system, the Juan Vucetich system, and the Henry Classification System. The Roscher system was developed in Germany and implemented in both Germany and Japan, the Vucetich system (developed by a Croatian-born Buenos Aires Police Officer) was developed in Argentina and implemented throughout South America, and the Henry system was developed in India and implemented in most English-speaking countries.[10]

In the Henry system of classification, there are three basic fingerprint patterns: loop, whorl and arch,[11] which constitute 60–65%, 30–35% and 5% of all fingerprints respectively.[citation needed] There are also more complex classification systems that break down patterns even further, into plain arches or tented arches,[10] and into loops that may be radial or ulnar, depending on the side of the hand toward which the tail points. Ulnar loops start on the pinky-side of the finger, the side closer to the ulna, the lower arm bone. Radial loops start on the thumb-side of the finger, the side closer to the radius. Whorls may also have sub-group classifications including plain whorls, accidental whorls, double loop whorls, peacock's eye, composite, and central pocket loop whorls.[10]

Other common fingerprint patterns include the tented arch, the plain arch, and the central pocket loop.

The system used by most experts, although complex, is similar to the Henry System of Classification. It consists of five fractions, in which *R* stands for right, *L* for left, *i* for index finger, *m* for middle finger, *t* for thumb, *r* for ring finger and *p*(pinky) for little finger. The fractions are as follows: *Ri/Rt + Rr/Rm + Lt/Rp + Lm/Li + Lp/Lr.* The numbers assigned to each print are based on whether or not they are whorls. A whorl in the first fraction is given a 16, the second an 8, the third a 4, the fourth a 2, and 0 to the last fraction. Arches and loops are assigned values of 0. Lastly, the numbers in the numerator and denominator are added up, using the scheme:

(Ri + Rr + Lt + Lm + Lp)/(Rt + Rm + Rp + Li + Lr)

and a 1 is added to both top and bottom, to exclude any possibility of division by zero. For example, if the right ring finger and the left index finger have whorls, the fractions would look like this:

0/0 + 8/0 + 0/0 + 0/2 + 0/0 + 1/1, and the calculation: (0 + 8 + 0 + 0 + 0 + 1)/(0 + 0 + 0 + 2 + 0 + 1) = 9/3 = 3.

Using this system reduces the number of prints that the print in question needs to be compared to. For example, the above set of prints would only need to be compared to other sets of fingerprints with a value of 3.[12]

https://en.wikipedia.org/wiki/Fingerprint

To aid in the look-up process there are systems for mapping each print to a number.

Many prints map to the same number, and these prints are considered individually.

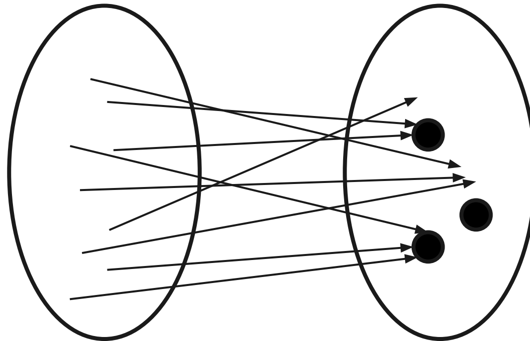
Hash Functions

Hash Function

A *hash function* f maps keys to integers.

Hash functions is *good* (i.e., useful in practice) when the following properties hold:

1. The function can be evaluated quickly.
2. The range of the function is used uniformly or close to uniformly.



Approximately the same number of arrows enter each point.
In other words, this function f spreads out evenly across its range.

The values of a hash function f are often large integers and the result is taken modulo M.

The result can be viewed as a new hash function f' (with the same domain and a smaller range).

- Choosing M to be a prime is often helpful.
- M is often the size of array used to store the values in a hash table.

Applications

Fingerprinting as a Metaphor

Let's focus on the following aspects of fingerprints:

- They reduce a lot of information down to a little information.
  - This information can be further reduced down to an individual number.
- This fingerprint can be obtained quickly.
  - Faster than a full DNA test.
- Two individuals with the same fingerprint, or the same fingerprint number, are probably the same individual, but not necessarily.
  - Two fingerprints that have the same number can be checked manually.

In computer science we often have similar goals.

Application: Duplicate Testing

Suppose that we have a large collection of documents and we'd like to determine if any are duplicates.

Document 1

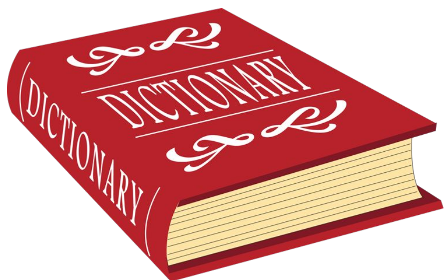
Document 2

Idea: Compute a hash value for each document.

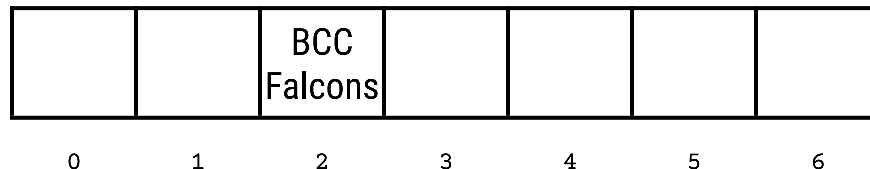
- The hash function should be fast to evaluate.
- False positives should be rare and can be checked thoroughly.
- Same idea works for images, files, etc.
- What are good/bad choices for the hash function?

Application: Efficient Maps using Hashing

Suppose that we'd like to implement a map or dictionary with <u>expected</u> O(1)-time for the most common operations of `get`, `put`, and `remove`? This will be our primary application for hashing.



Think of a dictionary with dynamic data
(i.e., (key, value) pairs are added and removed).



We want to store the (key, value) pairs in an array.
We'll use the hash of each key to determine its index.
In this example, f(BCC) = 2, so we store (key, value) for BCC in index 2.

Idea: Compute the hash function for the key, and use this as the index into the array.

- How big should the array be?
- What happens when keys map to the same integer?

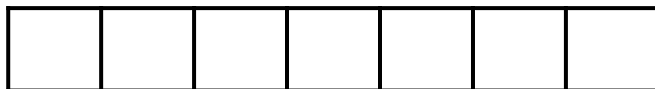
Hash Tables

Hash Table

A *hash table* is a data structure that includes two parts:

a) An array of size M that stores values.
b) A hash function that maps values to indices {0, 1, …, M-1}. Remember that the function may be taken modulo M.

When a hash table is used to implement a map, the values are actually (key, value) pairs, and the hash function maps keys to indices.



A hash table with an array of length M = 7.
The hash function will map all values (or keys in (key, value) pairs) to indices in {0, 1, …, 6}.
(Or the results of the hash function will be taken modulo 7 to make the above true.)

A *collision* occurs when two different values (or keys in (key, value) pairs) map to the same index.

- This will happen infrequently if the hash function's range is significantly smaller than M, and if the hash function is good.

Example: Phone Numbers

Suppose that we wish to maintain a collection of ~100 phone numbers.

What would be a good hash function for this type of data?

IMPORTANT PHONE NUMBERS (Area code 619)

Adult Enrichment Center	667.1322	Fire (Business)	667.1355	Recycling/Trash	287.7555
Animal Control	667.7536	Graffiti Hotline	667.7560	Rides4Neighbors	667.1321
Building Inspection	667.1176	Human Resources	667.1175	Senior Taxi Scrip	667.1321
Business Licenses	667.1118	Library	469.2151	Sewer Charges	667.1126
City Attorney	667.1128	Mayor	667.1100	Sidewalk Maint.	667.1450
City Clerk	667.1120	Municipal Pool	667.1494	Storm Water Hotline	667.1134
City Council	667.1106	Park Permits	667.1300	Streetlight Maint.	667.1450
City Hall	463.6611	Parking Citations	667.1117	Traffic Eng. Hotline	667.1144
City Manager	667.1105	Planning	667.1177	Traffic Signal Maint.	667.1166
Crime Prevention	667.7845	Police (Business)	667.1400	Zoning	667.1177
Dog Licenses	667.1114	Public Works	667.1450		
Engineering	667.1166	Recreation	667.1300		

A list of phone numbers.

A good function would be the last two digits, or the sum of all of the digits.

A bad function would be the first two digits.

Regardless of the choice there could still be collisions.

Example: Strings

English words can be mapped to base-26 numbers in the normal way:

A	L	P	H	A	B	E	T
0	11	14	7	0	1	4	19

This number can then be mapped to a base-10 number in the normal way:

$$0 \cdot 26^7 + 11 \cdot 26^6 + 14 \cdot 26^5 + 7 \cdot 26^4 + 0 \cdot 26^3 + 1 \cdot 26^2 + 4 \cdot 26^1 + 19 \cdot 26^0$$

This is a good hash function in many situations and for many different values of M.

- It can be applied to different string data.
- What about binary string data?

Resolving Collisions

There are two basic approaches to resolving collisions.

1. Chaining.
2. Open Addressing.

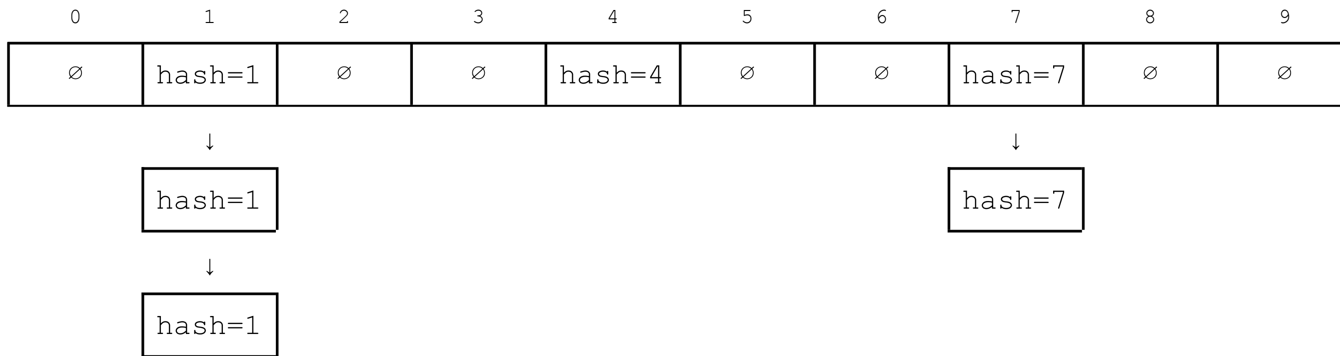
They have different pros and cons depending on the situation.

Chaining

Chaining

In *chaining* the entries of the array are actually references to unsorted linked lists.

In other words, the array entries are like the buckets in bucket sort.



A hash table that uses chaining to handle collisions.

`put`: Add the new entry to the front of the appropriate linked list.

`get`: Travel through the appropriate linked list to find the entry.

`remove`: Travel through the appropriate linked list to find and delete the entry.

Open Addressing

Open Addressing

In *open addressing* the next open entry of the array is used during insertions.

Depending on the order of insertions the next open entry could be after an entry with a different hash value.

0	1	2	3	4	5	6	7	8	9
		hash=2		hash=4			hash=7		hash=9
		hash=2	hash=2	hash=4			hash=7		hash=9
		hash=2	hash=2	hash=4	hash=2		hash=7		hash=9
hash=9		hash=2	hash=2	hash=4	hash=2		hash=7		hash=9

A sequence of insertions into a hash table that uses open addressing for collisions.
The top row shows four values in the hash table. Then successive rows show the result of put on values that hash to 2, 2, 2, and 9, respectively.

`put`: Add the new entry to the next open entry to the right (wrapping-around).

`get`: Start at the correct entry and proceed right until finding it or an empty entry.

`remove`: Start at the correct entry and proceed right to find the entry if it exists. Delete the value and rehash the values to the right until there is an empty entry.

`structure` Package

```
 */
public class Hashtable<K,V> implements Map<K,V>, Iterable<V>
{
    /**
     * A single key-value pair to be used as a token
     * indicating a reserved location in the hashtable.
     * Reserved locations are available for insertion,
     * but cause collisions on lookup.
     */
    protected static final String RESERVED = "RESERVED";
    /**
     * The data associated with the hashtable.
     */
    protected Vector<HashAssociation<K,V>> data;
    /**
     * The number of key-value pairs in table.
     */
    protected int count;

    /**
     * The maximum load factor that causes rehashing of the table.
     */
    protected final double maximumLoadFactor = 0.6;

    /**
     * Construct a hash table that is capable of holding at least
     * initialCapacity values.  If that value is approached, it will
     * be expanded appropriately.  It is probably best if the capacity
     * is prime.  Table is initially empty.
     *
     * @pre initialCapacity > 0
     * @post constructs a new Hashtable
```

Take a look at `Hashtable.java` and related files.

Applications

Applications

The three most important algorithms at Yahoo (according to Udi Manber) are:

1. *Hashing*.
2. *Hashing*.
3. *Hashing*.

There are several reasons for this:

- In practice the expected run-time is often the most important measurement, whereas in theory we usually concentrate on worst-case run-time.
- Big companies often don't mind paying for extra storage space.
- There are many different types of applications of hashing.

In some cases the application only works because the underlying hash function has a specific property. We will see an example of this on the next slide.

Application: File Integrity

Suppose that we want to ensure that a particular file was downloaded correctly from the internet.

Idea: The website provides a hash for the file, and the downloader verifies that the download was correct by making sure that their hash of the file is the same.

For this application we need the following properties:

- The hash function should be sensitive to small changes in the file.
- Both parties use the same hash function.

Application: Cryptographic Hash

Suppose that we want to detect if a particular file is modified, without keep a second copy of it.

Idea: Store a hash of the file. We can detect modifications to the file by checking its hash value.

For this application we need the following properties:

- It should be very difficult to create a second file with the same hash value.
  In other words, finding inverses should be very difficult.

Application: Detecting Similarity

Suppose that we want to check a large number of images for similarity.

Idea: Compute a hash value for each file. If similar images map to similar hash values, then we can sort the hash values and check for close values.

For this application we need the following properties:

- Similar items have similar hash values.

Application: Substring Matching

Suppose that we want to find a substring of length k in a larger string of length n.
How long will this take using brute force?

Idea: Compute a hash value for each substring of length k.

For this application we need the following properties:

- The hash value of $s_1\ s_2\ \dots\ s_k$ should be computable quickly from the hash value of $s_0\ s_1\ \dots\ s_{k-1}$, so that it can be updated. (The hash functions for strings that we discussed does have this property.