Notes

REVIEW: `const char*` vs `char *const` vs `const char *const`

CS221
C and Systems Programming

Hash functions

Hash maps

2

What is a hash function?

A **hash function** is a computer function that takes some chunk of data and maps it to some smaller fixed-size value (often `unsigned`)

Data is often of arbitrary size

Hash Functions - uses

Who cares? Where do we use these?

- Encryption and security
- Data structures
- Cache memory management
- Database indexing/search
- File integrity checking

Security: passwords are stored "hashed" not plaintext
Cache mgmt: if 2 items have same hash, write older one to slow storage, cache newer one
Database search: search for hashes rather than longer data
File integrity: check for modified files by checking if hash function is modified
Video standards use hashes for data integrity checking, for the most critical data

How to make a Hash Function?

We won't talk much about it. Needs fancy mathematics.

- But we will see some example functions

What do we want from a hash function?

- Collision resistance – difficult to find two different inputs with same hash value
- "Avalanche" – small change in input causes big change in output
- Pseudo-randomness – space of possible output values is covered uniformly, for "expected" inputs
- Deterministic – same input always gives same output value
- One-way – hard to calculate input data, given only the hash value
- Easy to compute

Reference: https://ocw.mit.edu/courses/

SHOW hashdemo.c: code and usage
This is not a great hash function.

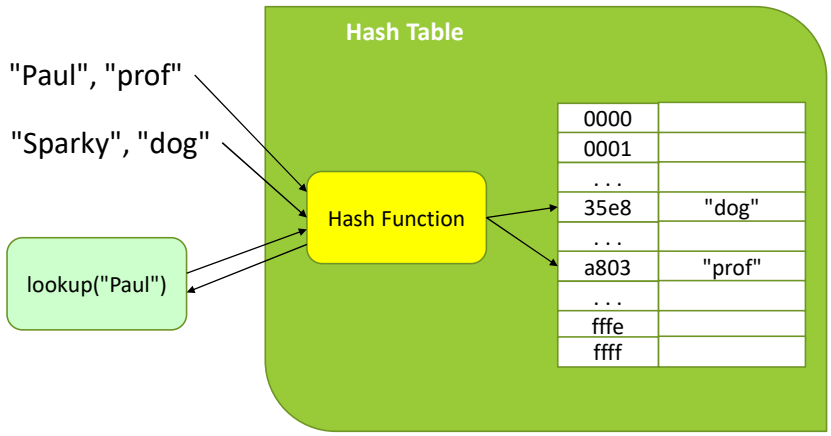
Hash Tables

```
% python
myDictionary = { }
myDictionary["Paul"] = "A Computer Science prof at USF"
myDictionary["Sparky"] = "The Wonder Dog"
print(myDictionary["Paul"])
```

```
% cat hashDemo.java
import java.util.HashMap;
HashMap<String, String> myDict = new HashMap<>();
myDict.add("Paul", "the prof");
myDict.add("Sparky", "the wonder dog");
```

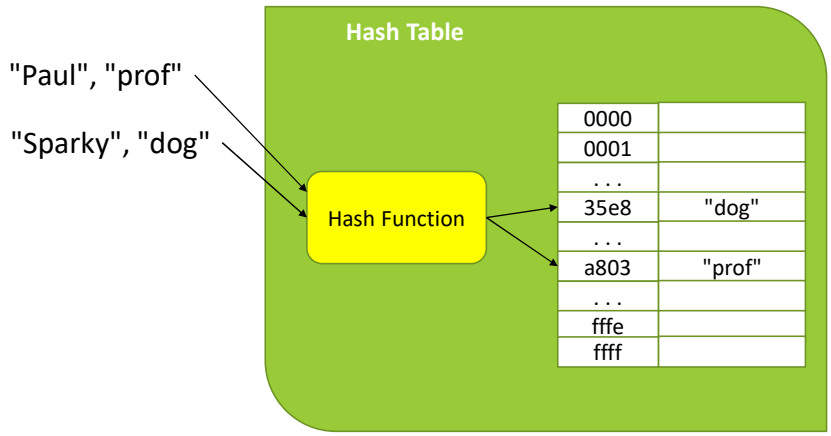
Hash **Tables**

Store arbitrary-type KEYS and VALUES, using hash value of KEY as **INDEX INTO AN ARRAY**



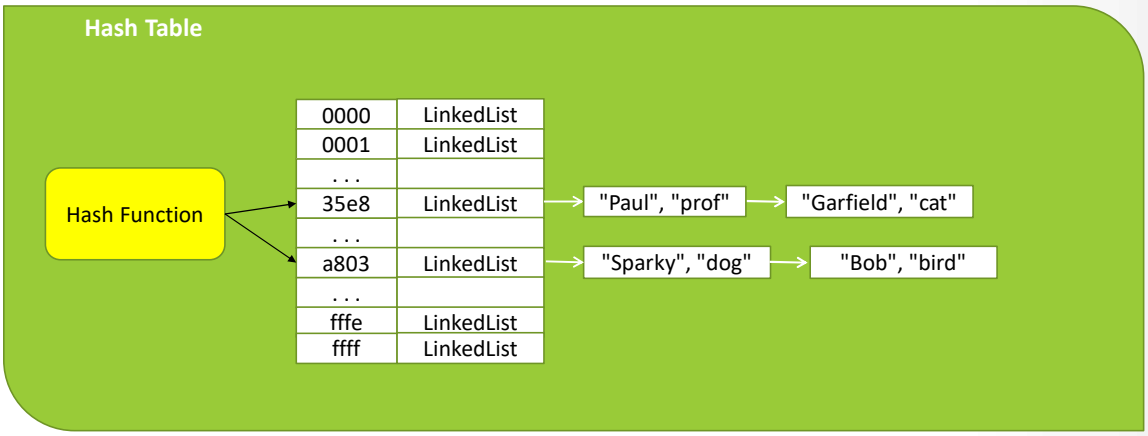
We don't have to search through a long list to find the string "Paul". We don't even need to search through a sorted tree to find the string. We can find the value in O(1) time…at the expense of memory.

Hash Tables – speed vs memory



We have 16-bit hash values. 64k entries in our hash table. As long as we have << 64k inputs, and if the probability of COLLISION is low, this is fine. But suppose we have millions of entries. Do we need hundreds of billions of hash entries? Pretty expensive

Hash Tables – speed vs memory TRADE-OFF

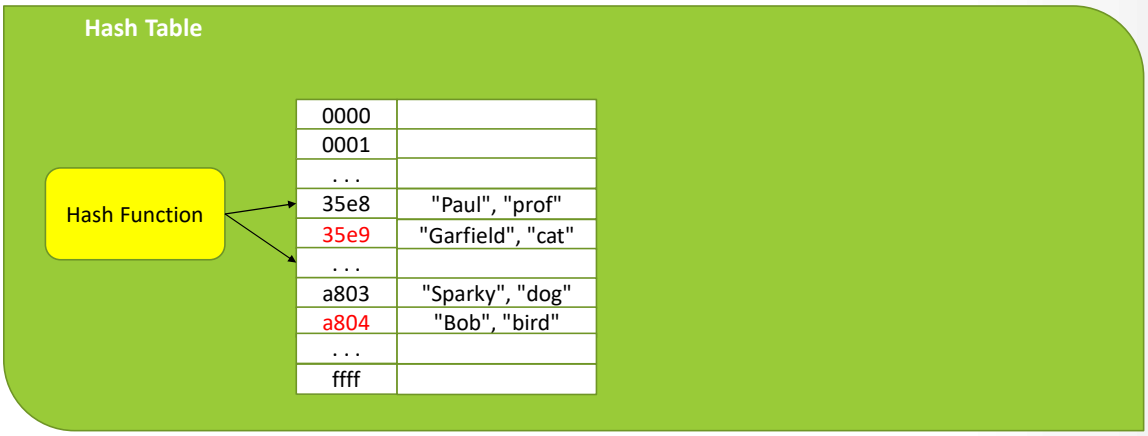


We can TRADE between memory and speed: more memory, FEWER duplicate keys, shorter linked lists

If Hash Function is Uniform, most LinkedLists should have about the same number of entries.

Remember: hash lookup speed is O(1). LinkedList speed is O(list length)

Hash Tables – speed vs memory TRADE-OFF



If your table is big enough to store all your data, instead of LinkedLists, if you have a COLLISION, just look for the NEXT EMPTY slot
(Garfield's hash is 35e8. Bob's hash is a803.)

Fancy Mathematics, you say?

Some widely used Hash Functions shown not as good as the designers thought

- **MD4**: Pr{collision} ≈ 2^{-6}
- **MD5**: Pr{collision} ≈ 2^{-37}

Commonly used hash functions: **SHA-224, SHA-256, SHA-384, SHA-512**

These are used in programs such as **ssh**, BitCoin, etc.

Goal for MD5 was exp(2, -64)