CSE 326: Data Structures
Lecture #12
Whoa... Good Hash, Man

Bart Niswonger

Summer Quarter 2001

Today's Outline

- Unix Tutorial
  - What do you want covered?
- Midterm
  - Amortized time
  - ADT vs Data Structure
- Hashing

Intermediate Unix Tutorial

- 2 minutes

- 3 things you love about unix
- 3 things you hate
- 5 things you wish you knew how to do

- 1 gift idea

Asymptotic Time

- Bounds *worst-case* running time
  - Over *m* operations

- Worst-case for *single* operation may be really bad, but worst-case for *m* operations is bounded

ADT vs Data Structure

Abstract Data Type	*Data structures*
– Abstract	– Concrete implementation
– Operations & semantics	– Set of algorithms
– Data-less	– Holds data
– One	– Many
– No notion of running time or complexity	– Very particular running times and complexities

Review

Dictionary ADT

- Dictionary operations
  - create
  - destroy
  - insert
  - find
  - delete

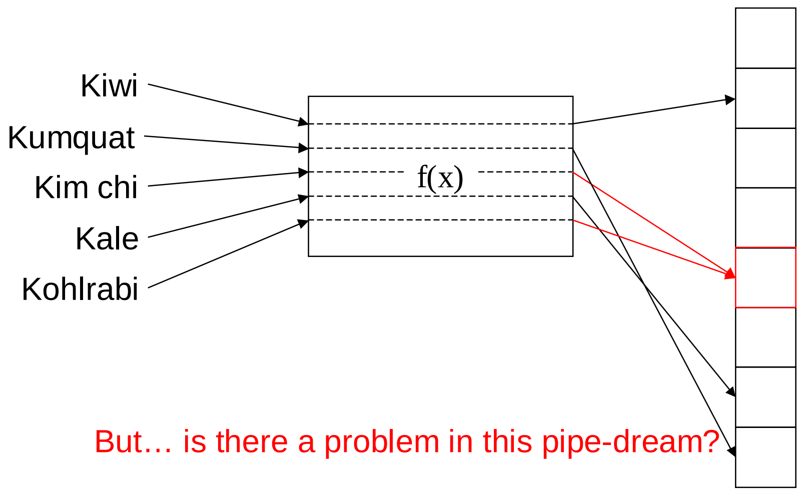
insert →
•kohlrabi
- upscale tuber

← find(kiwi)
• kiwi
- Australian fruit

- kim chi
  - spicy cabbage
- Krispy Kreme
  - tasty doughnut
- kiwi
  - Australian fruit
- kale
  - leafy green
- Krispix
  - breakfast cereal

- Stores *values* associated with user-specified *keys*
  - values may be any (homogenous) type
  - keys may be any (homogenous) comparable type

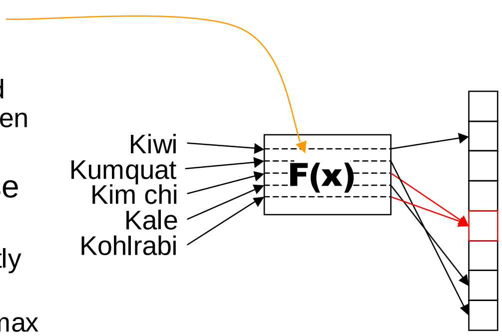
Hash Table Approach



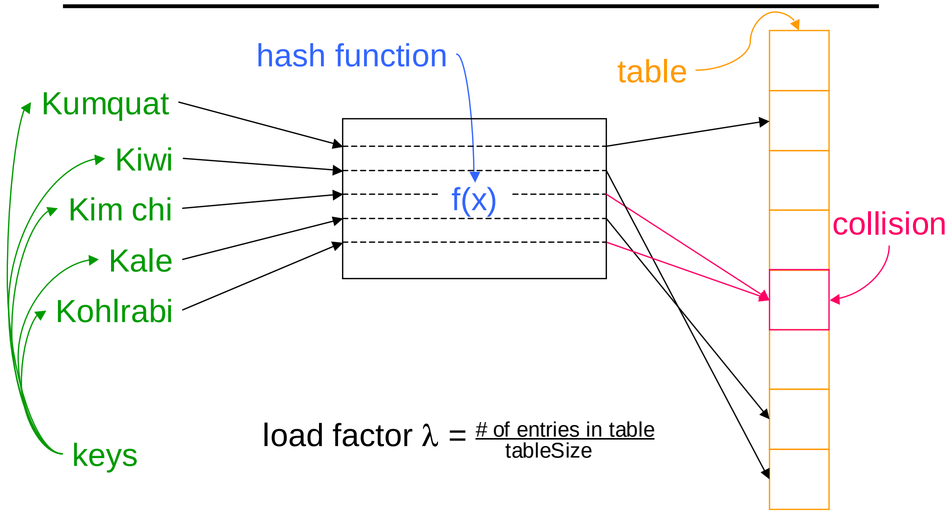
But… is there a problem in this pipe-dream?

Hash Table
Dictionary Data Structure

- Hash function: maps keys to integers
  - result: can quickly find the right spot for a given entry
- Unordered and sparse table
  - result: cannot efficiently list all entries,
  - Cannot find min and max efficiently,
  - Cannot find all items within a specified range efficiently.



Hash Table Terminology



Hash Table Code (First Pass)

```
Value & find(Key & key) {
  int index = hash(key) % tableSize;
  return Table[index];
}
```

What should the hash function be? (for integers)

How should we resolve collisions?

What should the table size be?

A Good Hash Function…

…is easy (fast) to compute (O(1) *and* practically fast).

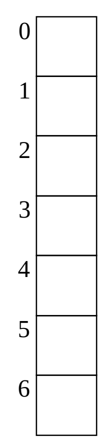
…distributes the data evenly (hash(a) $\neq$ hash(b))

…uses the whole hash table (for all $0 \leq k <$ size, there's an i such that hash(i) % size = k).

A Good Hash Function for Integers

- Choose
  - tableSize is prime
  - hash(n) = n % tableSize
- Example:
  - tableSize = 7

insert(4)
insert(17)
find(12)
insert(9)
delete(17)



Good Hash Function for Strings?

- I want to be able to:

  insert("kale")
  insert("Krispy Kreme")
  insert("kim chi")

Good Hash Function for Strings?

- Sum the ASCII values of the characters.
- Consider only the first 3 characters.
  - Uses only 2871 out of 17,576 entries in the table on English words.
- Let $s = s_1 s_2 s_3 s_4 \ldots s_5$: choose
  - $hash(s) = s_1 + s_2 128 + s_3 128^2 + s_4 128^3 + \ldots + s_n 128^n$

Think of the string as a base 128 number.

- Problems:
  - hash("really, really big") = well… something really, really big
  - hash("one thing") % 128 = hash("other thing") % 128

Easy to Compute String Hash

- Use Horner's Rule

```
int hash(String s) {
  h = 0;
  for (i = s.length() - 1; i >= 0; i--) {
    h = (s_i + 128*h) % tableSize;
  }
  return h;
}
```

Universal Hashing

- For any fixed hash function, there will be some pathological sets of inputs
  - everything hashes to the same cell!
- Solution: Universal Hashing
  - Start with a large (parameterized) class of hash functions
    - No sequence of inputs is bad for all of them!
  - When your program starts up, pick one of the hash functions to use at random (for the entire time)
  - Now: no bad inputs, only unlucky choices!
    - If universal class large, odds of making a bad choice very low
    - If you do find you are in trouble, just pick a different hash function and re-hash the previous inputs

“Random” Vector Universal Hash

- Parameterized by prime size and vector:

  $a = <a_0\ a_1\ \ldots\ a_r>$ where $0 <= a_i <$ size
- Represent each key as r + 1 integers where $k_i <$ size
  - size = 11, key = 39752 ==> <3,9,7,5,2>
  - size = 29, key = “hello world” ==> <8,5,12,12,15,23,15,18,12,4>

$$h_a(k) = \left(\sum_{i=0}^{r} a_i k_i\right) \bmod size$$

dot product with a “random” vector!

Universal Hash Function

- Strengths:
  - works on any type as long as you can form k_i’s
  - if we’re building a static table, we can try many *a*’s
  - a random *a* has guaranteed good properties no matter what we’re hashing
- Weaknesses
  - must choose prime table size larger than any k_i

Hash Function Summary

- Goals of a hash function
  - reproducible mapping from key to table entry
  - evenly distribute keys across the table
  - separate commonly occurring keys (neighboring keys?)
  - complete quickly
- Example Hash functions
  - h(n) = n % size
  - h(n) = string as base 128 number % size
  - One Universal hash function: dot product with random vector

How to Design a Hash Function

- Know what your keys are
- Study how your keys are distributed
- Try to include all important information in a key in the construction of its hash
- Try to make “neighboring” keys hash to very different places
- Prune the features used to create the hash until it runs “fast enough” (very application dependent)

Collisions

- *Pigeonhole principle* says we can't avoid all collisions
  - try to hash without collision m keys into n slots with $m > n$
  - try to put 6 pigeons into 5 holes
- What do we do when two keys hash to the same entry?
  - open hashing: put little dictionaries in each entry

    shove extra pigeons in one hole!
  - closed hashing: pick a next entry to try

To Do

- Project II
- Homework 4
- Read Chapter 5 (fast!)

Coming Up

- More hashing
- Cool stuff!
- Project III