Hash Tables

http://fscked.org/writings/225notes/week13/week13.html

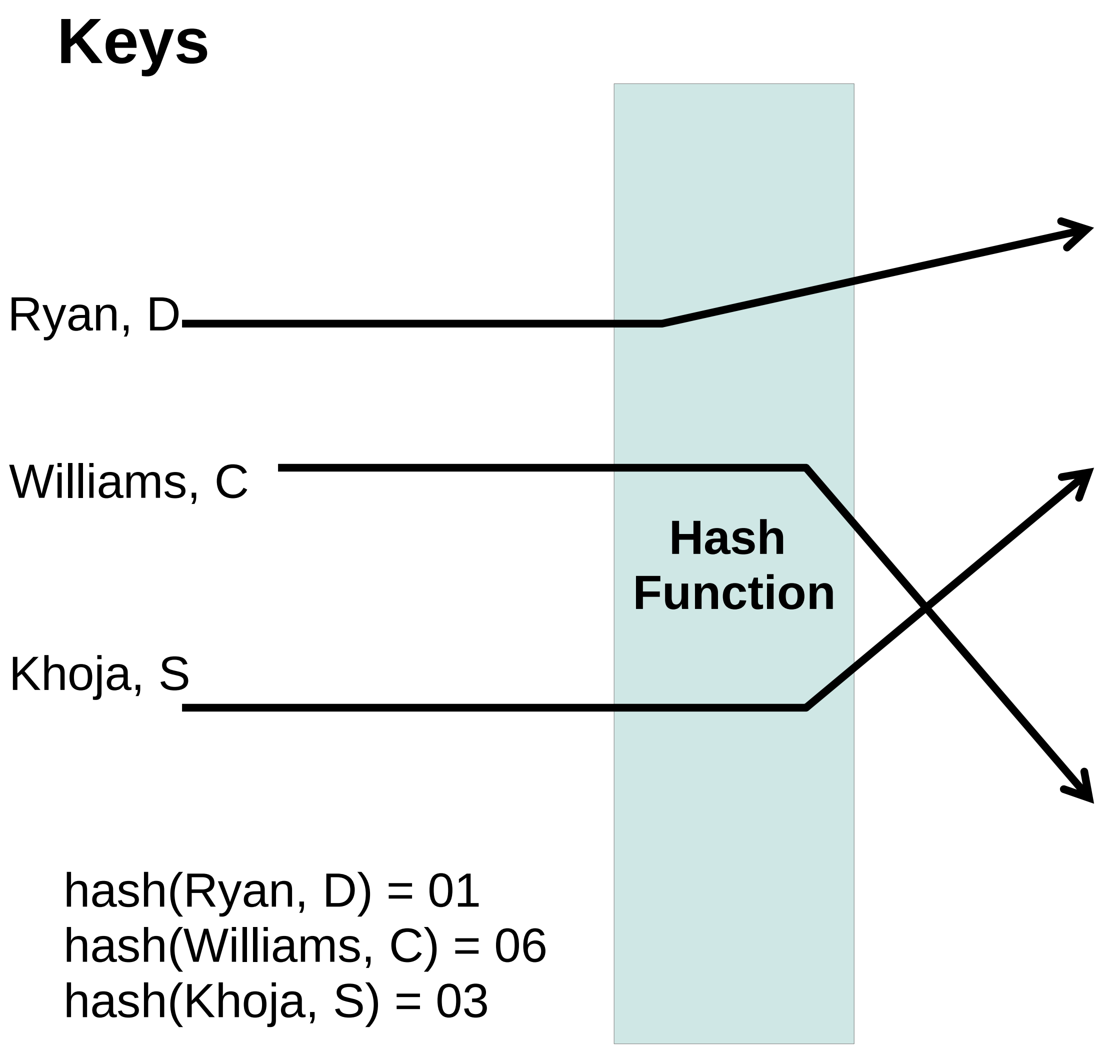
http://en.wikipedia.org/wiki/Hash_table

Hash Table

- A hash table (or hash map) is a data structure that maps keys (identifiers) into a certain location (bucket)

- A hash function changes the key into an index value (or hash value)

The Hash Table has a fixed length. We'll see how to add space dynamically later.



hash(Ryan, D) = 01
hash(Williams, C) = 06
hash(Khoja, S) = 03

Bucket	Data
01	Ryan, D
02	
03	Khoja, S
04	
05	
06	Williams, C
07	

Collisions

- Perfect Hash - each key maps to an empty bucket
  - Rare!
- Collisions occur where two different keys map to the same bucket

hash(Ryan, D) = 01
hash(Knuth, D) = 01

- Solution?

Hash Function

- Hash function – compute the key's bucket address from the key
  - some function h(K) maps the domain of keys K into a range of addresses 0, 1, 2, … M-1

The Problem

- Finding a suitable function h
- Determining a suitable M
- Handling collisions

Hash Function

- Mid Square
  - (turn the key into an integer)
  - square the key
  - take some number of bits from the center to form the bucket address

Example

- Problem: Let's assume that the key value is simply the sum of the ASCII values squared. If the key value is 16-bits and we take the middle 8-bits:

a) How big is the hash table?

b) What is the range of bucket addresses?

c) Where does the key AB map to in the hash table?

Implementation section 2.9

- How do we access the middle 8 in an integer?

One Hex-digit is 4 bits

-
```
// assume 4 byte integers
unsigned int key = 0x1231a456;
unsigned int middle;

middle = (key & 0x000ff000) >> 12;

printf("%08x %08x\n", key, middle);
```

pad with zero (0) — 8 wide (8) — hexadecimal output (x)

http://www.eskimo.com/~scs/cclass/int/sx4ab.html

Hash Function

- Division Hashing
  - bucket = key % N
  - N is the length of the hash table AND a prime number

a) How big is the hash table?

b) What is the range of bucket addresses?

c) Where does the key AB map to in the hash table?

Advantages?

Disadvantages?

Collision Handling

- Open Addressing
  - If both K and C map to the same bucket we have a collision
    - K and C are distinct
    - K is inserted first
  - To resolve using OA, find another unoccupied space for C

    BUT: We must do this systematically so we can find C again easily!
- Analysis: (summation of the # of probes to locate each key in the table) / # of keys in the table

Open Addressing

- Find another open bucket
- bucket = (h(K) + f(i)) % N
  - N is the length of the table
  - h(K) : original hash of key K
  - f(i) : i is the number of times you have hashed and failed to find an empty slot
  - First hash is:
    - bucket = (h(K) + f(0)) % N
    - f(0) = 0

Linear Probing

- f(i) = i

- Example:
  h(Kn) = n % 11
  Insert
  M13
  G7
  Q17
  Y25
  R18
  Z26
  F6

Bucket	Data
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Primary Clustering!

Primary Clustering

- Primary Clustering - this implies that all keys that collide at address b will extend the cluster that contains b

Quadratic Probing

- f(i) = i^2
- Example:
  h(Kn) = n % 11
  Insert
  M13
  G7
  Q17
  Y25
  R18
  Z26
  F6

Bucket	Data
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Secondary Clustering!

Secondary Clustering

- Secondary Clustering - is when adjacent clusters join to form a composite cluster

Double Hash

- f(i) = h2(k) * i
  - h2(k) is some second hash function
  - unique probe sequence for every key

  - bucket = (h(K) + h2(K) * i) % N

  - h2(k) should be relatively prime to N for all k
    - don't produce zero

  - Example
    - h(k) = k % N
      h2(k) = 1 + (k % (N - 1))

Rehash

- Reallocate the table larger and reinsert every element

Chaining (Open Hashing)

- Each bucket is the head of a linked list
  - if you hash a key to a bucket, insert the data into the list
  - insert at front, back, or in sorted order.
    - why would this decision matter?

Problem

- Hash the keys M13, G7, Q17, Y25, R18, Z26, and F6 using the hash formula h(Kn) = n mod 9 with the following collision handling technique: (a) linear probing, (b) chaining

- Compute the average number of probes to find an arbitrary key K for both methods.

- avg = (summation of the # of probes to locate each key in the table) / # of keys in the table