CSE 326: Data Structures
Lecture #13

More Hash please

Bart Niswonger
Summer Quarter 2001

Today's Outline

- Hashing
  - Hashing strings
  - Universal hash functions
  - Collisions
  - Probing
  - Rehashing
  - …

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 $s = s_1s_2s_3s_4\ldots s_5$: choose
  - $hash(s) = s_1 + s_2128 + s_3128^2 + s_4128^3 + \ldots + s_n128^n$

  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

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:

  $a = \langle a_0\ a_1 \ldots a_r \rangle$ where $0 \le a_i <$ 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

“Random” Vector Universal Hash

- 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

Alternate Universal Hash Function

- Parameterized by k, a, and b:
  - k * size should fit into an int
  - a and b must be less than size

$$h_{k,a,b}(x) = ((a \cdot x + b) \bmod k \cdot size) / k$$

Alternate Universal Hash: Example

- Context: hash integers in a table of size 16
  let k = 32, a = 100, b = 200
  $h_{k,a,b}(1000)$ = ((100*1000 + 200) % (32*16)) / 32
  = (100200 % 512) / 32
  = 360 / 32
  = 11

Alternate Universal Hash Function

- Strengths:
  - if we're building a static table, we can try many parameter values
  - random a,b has guaranteed good properties no matter what we're hashing
  - can choose any size table
  - very efficient if k and size are powers of 2
- Weaknesses
  - still need to turn non-integer keys into integers

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 complete quickly
- Hash functions
  - h(n) = n % size
  - h(n) = string as base 128 number % size
  - One Universal hash function: dot product with random vector
  - Other Universal hash functions…

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)

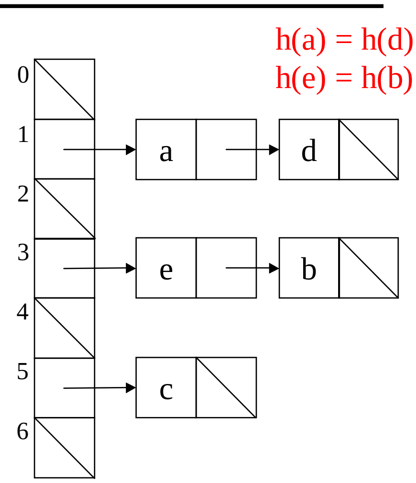
Collisions

- *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

Open Hashing or Hashing with Chaining

- Put a little dictionary at each entry
  - choose type as appropriate
  - common case is unordered linked list (chain)
- Properties
  - λ can be greater than 1
  - performance degrades with length of chains



Open Hashing Code

```
Dictionary & findBucket(const Key & k) {
  return table[hash(k)%table.size];
}

void insert(const Key & k,
            const Value & v)
{
  findBucket(k).insert(k,v);
}
```

```
void delete(const Key & k)
{
  findBucket(k).delete(k);
}

Value & find(const Key & k)
{
  return findBucket(k).find(k);
}
```

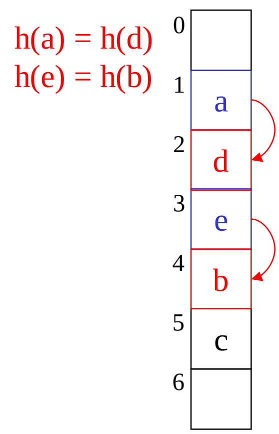
Load Factor in Open Hashing

- Search cost
  - unsuccessful search:
  - successful search:
- Desired load factor:

Closed Hashing / Open Addressing

What if we only allow one Key at each entry?
- two objects that hash to the same spot can't both go there
- first one there gets the spot
- next one must *go in another spot*

- Properties
  - $\lambda \leq 1$
  - performance degrades with difficulty of finding right spot



Probing

- Probing how to:
  - First probe - given a key k, hash to h(k)
  - Second probe - if h(k) is occupied, try h(k) + f(1)
  - Third probe - if h(k) + f(1) is occupied, try h(k) + f(2)
  - And so forth
- Probing properties
  - we force f(0) = 0
  - the i^{th} probe is to (h(k) + f(i)) mod size
  - if i reaches size - 1, the probe has failed
  - depending on f(), the probe may fail sooner
  - long sequences of probes are costly!

Linear Probing

- Probe sequence is
  - h(k) mod size
  - h(k) + 1 mod size
  - h(k) + 2 mod size
  - …

f(i) = i

- findEntry using linear probing:

```
bool findEntry(const Key & k, Entry *& entry) {
  int probePoint = hash1(k);
  do {
    entry = &table[probePoint];
    probePoint = (probePoint + 1) % size;
  } while (!entry->isEmpty() && entry->key != k);
  return !entry->isEmpty();
}
```

Linear Probing Example

	insert(76)	insert(93)	insert(40)	insert(47)	insert(10)	insert(55)
	76%7 = 6	93%7 = 2	40%7 = 5	47%7 = 5	10%7 = 3	55%7 = 6
0				47	47	47
1						55
2		93	93	93	93	93
3					10	10
4						
5			40	40	40	40
6	76	76	76	76	76	76
probes:	1	1	1	3	1	3

Load Factor in Linear Probing

- For *any* λ < 1, linear probing will find an empty slot
- Search cost (for large table sizes)
  - successful search: $\frac{1}{2}\left(1+\frac{1}{(1-\lambda)}\right)$
  - unsuccessful search: $\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$
- Linear probing suffers from *primary clustering*
- Performance quickly degrades for λ > 1/2

Quadratic Probing

$f(i) = i^2$

- Probe sequence is
  - h(k) mod size
  - (h(k) + 1) mod size
  - (h(k) + 4) mod size
  - (h(k) + 9) mod size
  - …
- findEntry using quadratic probing:

```
bool findEntry(const Key & k, Entry *& entry) {
  int probePoint = hash1(k), numProbes = 0;
  do {
    entry = &table[probePoint];
    numProbes++;
    probePoint = (probePoint + 2*numProbes - 1) % size;
  } while (!entry->isEmpty() && entry->key != key);
  return !entry->isEmpty();
}
```

Quadratic Probing Example J

	insert(76)	insert(40)	insert(48)	insert(5)	insert(55)
	76%7 = 6	40%7 = 5	48%7 = 6	5%7 = 5	55%7 = 6
0			48	47	47
1					
2				5	5
3					55
4					
5		40	40	40	40
6	76	76	76	76	76
probes:	1	1	2	3	3

Quadratic Probing Example L

	insert(76)	insert(93)	insert(40)	insert(35)	insert(47)
	76%7 = 6	93%7 = 2	40%7 = 5	35%7 = 0	47%7 = 5
0				35	35
1					
2		93	93	93	93
3					
4					
5			40	40	40
6	76	76	76	76	76
probes:	1	1	1	1	∞

Quadratic Probing Succeeds (for $\lambda \le ½$)

- If size is prime and $\lambda \le ½$, then quadratic probing will find an empty slot in size/2 probes or fewer.
  - show for all **`0 ≤ i, j ≤ size/2`** and **`i ≠ j`**

    `(h(x) + i²) mod size ≠ (h(x) + j²) mod size`
  - by contradiction: suppose that for some i, j:

    `(h(x) + i²) mod size = (h(x) + j²) mod size`

    `i² mod size = j² mod size`

    `(i² - j²) mod size = 0`

    `[(i + j)(i - j)] mod size = 0`
  - but how can **`i + j = 0`** or **`i + j = size`** when **`i ≠ j`** and **`i,j ≤ size/2`**?
  - same for **`i - j mod size = 0`**

Quadratic Probing May Fail (for $\lambda > ½$)

- For any i larger than size/2, there is some j smaller than i that adds with i to equal size (or a multiple of size). D'oh!

Load Factor in Quadratic Probing

- For *any* $\lambda \leq ½$, quadratic probing will find an empty slot; for greater λ, quadratic probing *may* find a slot
- Quadratic probing does not suffer from primary clustering
- Quadratic probing *does* suffer from *secondary* clustering
  - How could we possibly solve this?

Double Hashing

- Probe sequence is
  - $h_1(k)$ mod size
  - $(h_1(k) + 1 \cdot h_2(x))$ mod size
  - $(h_1(k) + 2 \cdot h_2(x))$ mod size
  - …

$f(i) = i \cdot hash_2(x)$

- Code for finding the next linear probe:

```
bool findEntry(const Key & k, Entry *& entry) {
  int probePoint = hash1(k), hashIncr = hash2(k);
  do {
    entry = &table[probePoint];
    probePoint = (probePoint + hashIncr) % size;
  } while (!entry->isEmpty() && entry->key != k);
  return !entry->isEmpty();
}
```

A Good Double Hash Function…

…is quick to evaluate.

…differs from the original hash function.

…never evaluates to 0 (mod size).

One good choice is to choose
prime R < size
and:
$hash_2(x) = R - (x \bmod R)$

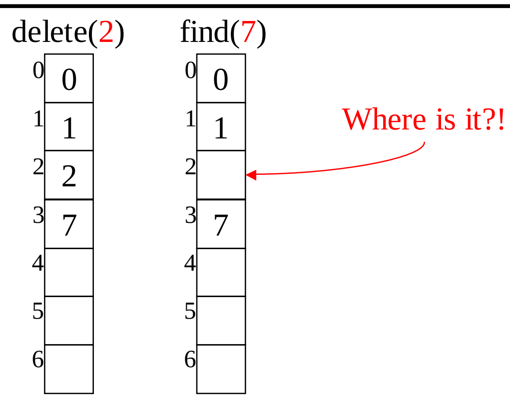
Double Hashing Example

	insert(76)	insert(93)	insert(40)	insert(47)	insert(10)	insert(55)
	76%7 = 6	93%7 = 2	40%7 = 5	47%7 = 5 5 - (47%5) = 3	10%7 = 3	55%7 = 6 5 - (55%5) = 5
0						
1				47	47	47
2		93	93	93	93	93
3					10	10
4						55
5			40	40	40	40
6	76	76	76	76	76	76
probes:	1	1	1	2	1	2

Load Factor in Double Hashing

- For *any* $\lambda < 1$, double hashing will find an empty slot (given appropriate table size and $hash_2$)
- Search cost appears to approach optimal (random hash):
  - successful search: $\frac{1}{\lambda} \ln \frac{1}{1-\lambda}$
  - unsuccessful search: $\frac{1}{1-\lambda}$
- No primary clustering and no secondary clustering
- One extra hash calculation

Deletion in Closed Hashing



- Must use lazy deletion!
- On insertion, treat a deleted item as an empty slot

The Squished Pigeon Principle

- An insert using closed hashing *cannot* work with a load factor of 1 or more.
- An insert using closed hashing with quadratic probing may not work with a load factor of ½ or more.
- Whether you use open or closed hashing, large load factors lead to poor performance!
- How can we relieve the pressure on the pigeons?

Hint: remember what happened when we overran a d-Heap's array!

Rehashing

- When the load factor gets "too large" (over a constant threshold on λ), rehash all the elements into a new, larger table:
  - takes O(n), but amortized O(1) as long as we (just about) double table size on the resize
  - spreads keys back out, may drastically improve performance
  - gives us a chance to retune parameterized hash functions
  - avoids failure for closed hashing techniques
  - allows arbitrarily large tables starting from a small table
  - clears out lazily deleted items

To Do

- Finish Project II
- Read chapter 5 in the book

Coming Up

- Extendible hashing (hashing for **HUGE** data sets)
- Disjoint-set union-find ADT
- Project II due (Wednesday)
- Project III Handout (Wednesday)
- Quiz (Thursday)