CSE 332: Hash Tables

Richard Anderson, Steve Seitz
Winter 2014

Announcements (1/29/14)

- HW #3 due now
- HW #4 out today
- Project 2A due Thursday night.
- Reading for this lecture: Chapter 5.

AVL find, insert, delete: O(log n)

Suppose (unique) keys between 0 and 1000.

– Can we do better than O(log n)?

Arrays for Dictionaries

Now suppose keys are first, last names

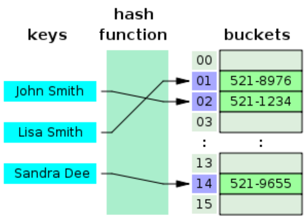
– how big is the key space?

But keyspace is sparsely populated

– $<10^5$ active students

Hash Tables

- Map keys to a smaller array called a hash table
  - via a hash function h(K)
  - Find, insert, delete: O(1) on average!



hash table

Simple Integer Hash Functions

- key space K = integers
- TableSize = 10
- h(K) =
- **Insert**: 7, 18, 41, 34

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Simple Integer Hash Functions

- key space K = integers
- TableSize = 7
- h(K) = K % 7
- **Insert**: 7, 18, 41, 34

0	
1	
2	
3	
4	
5	
6	

Aside: Properties of Mod

To keep hashed values within the size of the table, we will generally do:

h(K) = function(K) % TableSize

(In the previous examples, function(K) = K.)

Useful properties of mod:

– (a + b) % c = [(a % c) + (b % c)] % c
– (a b) % c = [(a % c) (b % c)] % c
– a % c = b % c $\rightarrow$ (a – b) % c = 0

String Hash Functions?

What's a good hash function for a string?

Some String Hash Functions

key space = strings

K = $s_0\ s_1\ s_2 \ldots s_{m-1}$ (where s_i are chars: $s_i \in [0, 128]$)

1. h(K) = s_0 % TableSize
2. h(K) = $\left(\sum_{i=0}^{m-1} s_i\right)$ % TableSize
3. h(K) = $\left(\sum_{i=0}^{m-1} s_i \cdot 128^i\right)$ % TableSize

Hash Function Desiderata

What are good properties for a hash function?

Designing Hash Functions

Often based on **modular hashing**:

h(K) = f(K) % P

P is typically the TableSize

P is often chosen to be prime:

– Reduces likelihood of collisions due to patterns in data
– Is useful for guarantees on certain hashing strategies (as we'll see)

But what would be a more convenient value of P?

A Fancier Hash Function

Some experimental results indicate that modular hash functions with prime tables sizes are not ideal.

Lots of better solutions, e.g.,

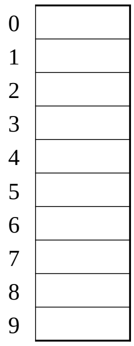
```
jenkinsOneAtATimeHash(String key, int keyLength) {
   hash = 0;
   for (i = 0; i < key_len; i++) {
        hash += key[i];
        hash += (hash << 10);
        hash ^= (hash >> 6);
   }
   hash += (hash << 3);
   hash ^= (hash >> 11);
   hash += (hash << 15);

   return hash % TableSize;
}
```


Collision Resolution

Collision: when two keys map to the same location in the hash table.

How handle this?

Separate Chaining



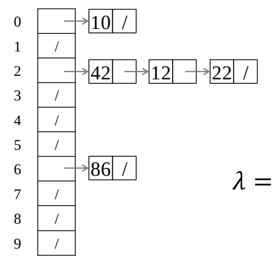
Insert:
10
22
107
12
42

All keys that map to the same hash value are kept in a list (or "bucket").

Analysis of Separate Chaining

The load factor, λ, of a hash table is $\lambda = \frac{N}{\text{TableSize}}$

λ = average # of elems per bucket



$\lambda =$

Analysis of Separate Chaining

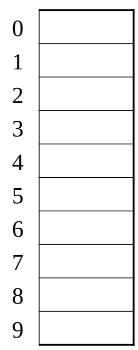
The load factor, λ, of a hash table is $\lambda = \frac{N}{\text{TableSize}}$

λ = average # of elems per bucket

Average cost of:

- Unsuccessful find?
- Successful find?
- Insert?

Alternative: Use Empty Space in the Table



Insert:
38
19
8
109
10

Try h(K).
If full, try h(K)+1.
If full, try h(K)+2.
If full, try h(K)+3.
Etc…

Open Addressing

The approach on the previous slide is an example of **open addressing**:

After a collision, try "next" spot. If there's another collision, try another, etc.

Finding the next available spot is called **probing**:

0^{th} probe = h(k) % TableSize
1^{th} probe = (h(k) + f(1)) % TableSize
2^{th} probe = (h(k) + f(2)) % TableSize
. . .
i^{th} probe = (h(k) + f(i)) % TableSize

f(i) is the probing function. We'll look at a few…

Linear Probing

f(i) = i

- Probe sequence:
  - 0^{th} probe = h(K) % TableSize
  - 1^{th} probe = (h(K) + 1) % TableSize
  - 2^{th} probe = (h(K) + 2) % TableSize
  - . . .
  - i^{th} probe = (h(K) + i) % TableSize

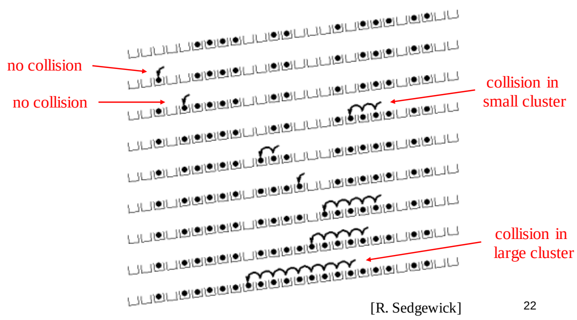
Linear Probing

0	8
1	109
2	10
3	
4	
5	
6	
7	
8	38
9	19

Insert:
38
19
8
109
10

Try h(K)
If full, try h(K)+1.
If full, try h(K)+2.
If full, try h(K)+3.
Etc…

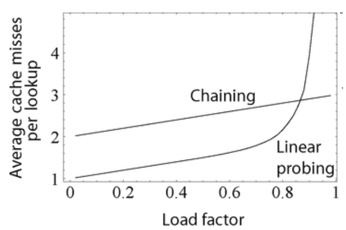
Linear Probing – Clustering



[R. Sedgewick]

Analysis of Linear Probing

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



Quadratic Probing

Less likely to encounter Primary Clustering

$f(i) = i^2$

- Probe sequence:
  - 0^{th} probe = h(K) % TableSize
  - 1^{th} probe = (h(K) + 1) % TableSize
  - 2^{th} probe = (h(K) + 4) % TableSize
  - 3^{th} probe = (h(K) + 9) % TableSize
  - . . .
  - i^{th} probe = (h(K) + i^2) % TableSize

25

Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Insert:
89
18
49
58
79

26

Another Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	

TableSize = 7
h(K) = K % 7

insert(76) 76 % 7 =6
insert(40) 40 % 7 =5
insert(48) 48 % 7 =6
insert(5) 5 % 7 =5
insert(55) 55 % 7 =6
insert(47) 47 % 7 =5

27

Quadratic Probing: Success guarantee for $\lambda < ½$

Assertion #1: If T = TableSize is **prime** and $\lambda < ½$, then quadratic probing will find an empty slot in $\leq$ T/2 probes

Assertion #2: For prime T and all $0 \leq i,j \leq T/2$ and $i \neq j$,

$$(h(K) + i^2) \% T \neq (h(K) + j^2) \% T$$

Assertion #3: Assertion #2 proves assertion #1.

28

Quadratic Probing: Success guarantee for $\lambda < ½$

We can prove assertion #2 by contradiction.
Suppose that for some $i \neq j$, $0 \leq i,j \leq T/2$, prime T:

$$(h(K) + i^2) \% T = (h(K) + j^2) \% T$$

29

Quadratic Probing: Properties

- For *any* $\lambda < ½$, quadratic probing will find an empty slot; for bigger λ, quadratic probing *may* find a slot.
- Quadratic probing does not suffer from *primary* clustering: keys hashing to the same *area* is ok
- But what about keys that hash to the same *slot*?
  - ***Secondary Clustering!***

30

Double Hashing

Idea: given two different (good) hash functions h(K) and g(K), it is unlikely for two keys to collide with both of them.

So…let's try probing with a second hash function:

f(i) = i * g(K)

- Probe sequence:
  - 0^{th} probe = h(K) % TableSize
  - 1^{th} probe = (h(K) + g(K)) % TableSize
  - 2^{th} probe = (h(K) + 2*g(K)) % TableSize
  - 3^{th} probe = (h(K) + 3*g(K)) % TableSize
  - . . .
  - i^{th} probe = (h(K) + i*g(K)) % TableSize

Double Hashing Example

0	
1	
2	
3	
4	
5	
6	

TableSize = 7
h(K) = K % 7
g(K) = 5 – (K % 5)

Insert(76) 76 % 7 = 6 and 5 - 76 % 5 =
Insert(93) 93 % 7 = 2 and 5 - 93 % 5 =
Insert(40) 40 % 7 = 5 and 5 - 40 % 5 =
Insert(47) 47 % 7 = 5 and 5 - 47 % 5 =
Insert(10) 10 % 7 = 3 and 5 - 10 % 5 =
Insert(55) 55 % 7 = 6 and 5 - 55 % 5 =

Another Example of Double Hashing

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hash Functions:
T = TableSize = 10
h(K) = K % T
g(K) = 1 + (K/T) % (T-1)

Insert these values into the hash table in this order. Resolve any collisions with double hashing:
13
28
33
147
43

Analysis of Double Hashing

- Double hashing is safe for $\lambda < 1$ for this case:
  - h(k) = k % p
  - g(k) = q – (k % q)
  - 2 < q < p, and p, q are primes
- Expected # of probes (for large table sizes)
  - unsuccessful search:

$$\frac{1}{1-\lambda}$$

  - successful search:

$$\frac{1}{\lambda}\log_e\left(\frac{1}{1-\lambda}\right)$$

Deletion in Separate Chaining

How do we delete an element with separate chaining?

Deletion in Open Addressing

0	
1	
2	16
3	23
4	59
5	
6	76

h(k) = k % 7
Linear probing

Delete(23)
Find(59)
Insert(30)

Need to keep track of deleted items... leave a "marker"

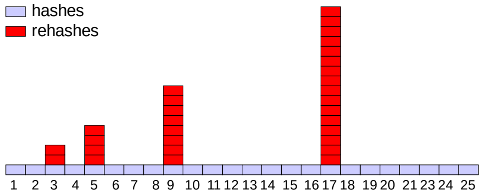
Rehashing

When the table gets too full, create a bigger table (usually 2x as large) and hash all the items from the original table into the new table.

- When to rehash?
  - Separate chaining: full ($\lambda = 1$)
  - Open addressing: half full ($\lambda = 0.5$)
  - When an insertion fails
  - Some other threshold
- Cost of a single rehashing?

Rehashing Picture

- Starting with table of size 2, double when load factor > 1.



Amortized Analysis of Rehashing

- Cost of inserting n keys is < 3n
- suppose $2^k + 1 \le n \le 2^{k+1}$
  - Hashes = n
  - Rehashes = $2 + 2^2 + \ldots + 2^k = 2^{k+1} - 2$
  - Total = $n + 2^{k+1} - 2 < 3n$
- Example
  - n = 33, Total = 33 + 64 –2 = 95 < 99

Equal objects must hash the same

- The Java library (and your project hash table) make a very important assumption that clients must satisfy…

  If `c.compare(a,b) == 0`, then we require
  `h.hash(a) == h.hash(b)`

- If you ever override equals
  - You need to override hashCode also in a consistent way
  - See CoreJava book, Chapter 5 for other "gotchas" with equals

Hashing Summary

- Hashing is one of the most important data structures.
- Hashing has many applications where operations are limited to find, insert, and delete.
  - But what is the cost of doing, e.g., findMin?
- Can use:
  - Separate chaining (easiest)
  - Open hashing (memory conservation, no linked list management)
  - Java uses separate chaining
- Rehashing has good amortized complexity.
- Also has a big data version to minimize disk accesses: extendible hashing. (See book.)

Terminology Alert!

- We (and the book) use the terms
  - “chaining” or “separate chaining”
  - “open addressing”
- Very confusingly
  - “open hashing” is a synonym for “chaining”
  - “closed hashing” is a synonym for “open addressing”