CSE 373: Data Structures and Algorithms

Lecture 7: Hash Table Collisions

Instructor: Lilian de Greef

Quarter: Summer 2017

Today

- Announcements
- Hash Table Collisions
- Collision Resolution Schemes
  - Separate Chaining
  - Open Addressing / Probing
    - Linear Probing
    - Quadratic Probing
    - Double Hashing
- Rehashing

Announcements

- Reminder: homework 2 due tomorrow
- Homework 3: Hash Tables
    - Will be out tomorrow night
    - Pair-programming opportunity! (work with a partner)
    - Ideas for finding partner: before/after class, section, Piazza
- Pair-programming: write code together
    - 2 people, 1 keyboard
    - One is the “navigator,” the other the “driver”
    - Regularly switch off to spend equal time in both roles
    - Side note: our brains tend to edit out when we make typos
    - Need to be in same physical space for entire assignment, so partner and plan accordingly!

Review: Hash Tables & Collisions

Hash Tables: Review

- A data-structure for the dictionary ADT
- *Average case* O(1) `find`, `insert`, and `delete`
  (when under some often-reasonable *assumptions*)
- An array storing (key, value) pairs
- Use *hash value* and table size to calculate array index
- Hash value calculated from key using *hash function*

`find`, `insert`, or `delete`
(key, value)

⇩

apply *hash function*
h(key) = *hash value*

⇩

index = hash value % table size

⇩

if collision, apply collision resolution

⇩

array[index] = (key, value)

Hash Table Collisions: Review

- Collision:

- We try to *avoid* them by

- Unfortunately, collisions are unavoidable in practice
  - Number of possible keys >> table size
  - No perfect hash function & table-index combo

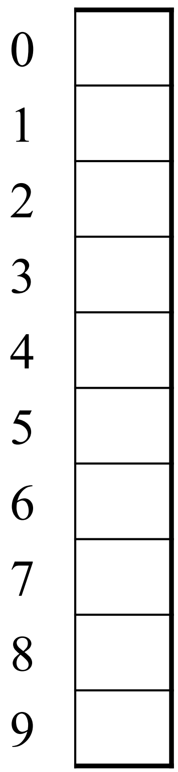
Collision Resolution Schemes: your ideas

Collision Resolution Schemes: your ideas

Separate Chaining

One of several collision resolution schemes

Separate Chaining



All keys that map to the same table location (aka “bucket”) are kept in a list (“chain”).

Example:

insert 10, 22, 107, 12, 42

and **`TableSize`** = 10

(for illustrative purposes, we’re inserting hash values)

Separate Chaining: Worst-Case

What’s the worst-case scenario for `find`?

What’s the worst-case running time for `find`?

But only with really bad luck or really bad hash function

Separate Chaining: Further Analysis

- How can `find` become slow when we have a good hash function?

- How can we reduce its likelihood?

Rigorous Analysis: Load Factor

Definition: The **load factor** (λ) of a hash table with *N* elements is

$$\lambda = \frac{N}{table\ size}$$

Under separate chaining, the average number of elements per bucket is _____

For a *random* `find`, on average

- an unsuccessful `find` compares against _______ items
- a successful `find` compares against _______ items

Rigorous Analysis: Load Factor

Definition: The **load factor** (λ) of a hash table with *N* elements is

$$\lambda = \frac{N}{table\ size}$$

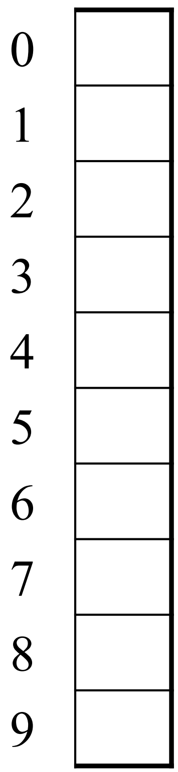
To choose a good load factor, what are our goals?

So for separate chaining, a good load factor is

Open Addressing / Probing

Another family of collision resolution schemes

Idea: use empty space in the table



- If `h(key)` is already full,
  - try `(h(key) + 1) % TableSize`. If full,
  - try `(h(key) + 2) % TableSize`. If full,
  - try `(h(key) + 3) % TableSize`. If full…

- Example: insert 38, 19, 8, 109, 10

Open Addressing Terminology

Trying the next spot is called (also called)

- We just did
  `i`th probe was `(h(key) + i) % TableSize`

- In general have some `f` and use
  `(h(key) + f(i)) % TableSize`

Dictionary Operations with Open Addressing

`insert` finds an open table position using a probe function

What about `find`?

What about `delete`?

- Note: `delete` with separate chaining is plain-old list-remove

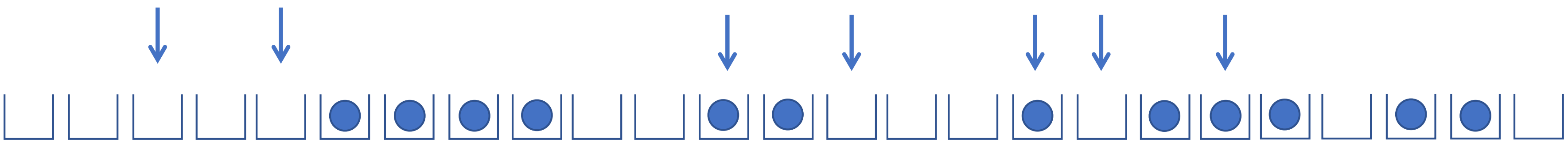
Practice:

The keys 12, 18, 13, 2, 3, 23, 5 and 15 are inserted into an initially empty hash table of length 10 using open addressing with hash function h(k) = k mod 10 and linear probing. What is the resultant hash table?

	(A)	(B)	(C)	(D)
0				
1				
2	2	12	12	12, 2
3	23	13	13	13, 3, 23
4			2	
5	15	5	3	5, 15
6			23	
7			5	
8	18	18	18	18
9			15	

Open Addressing: Linear Probing

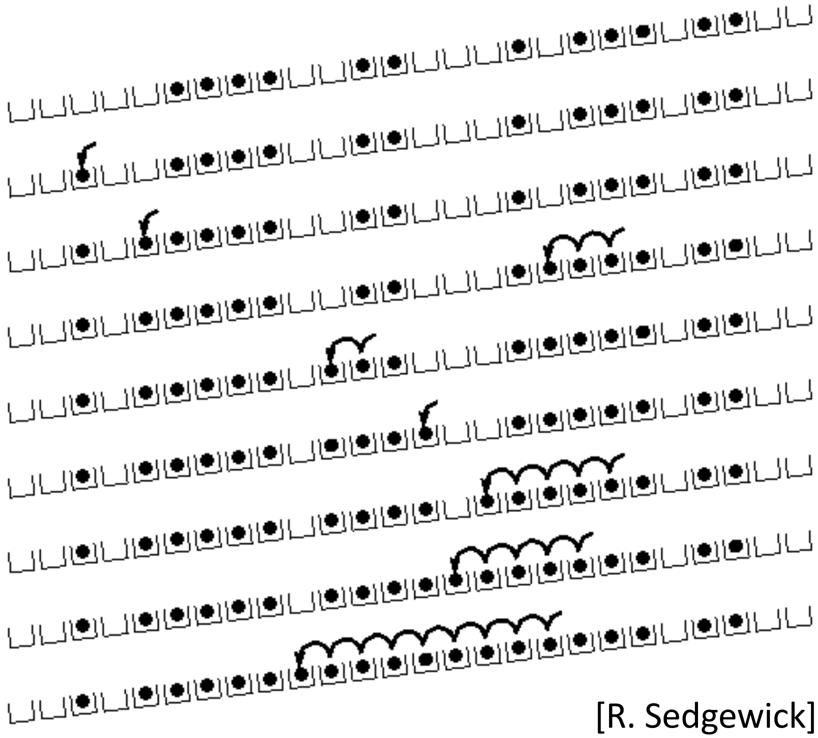
- Quick to compute! ☺
- But mostly a *bad idea*. Why?



(Primary) Clustering

Linear probing tends to produce ***clusters***, which lead to long probing sequences

- Called
- Saw this starting in our example



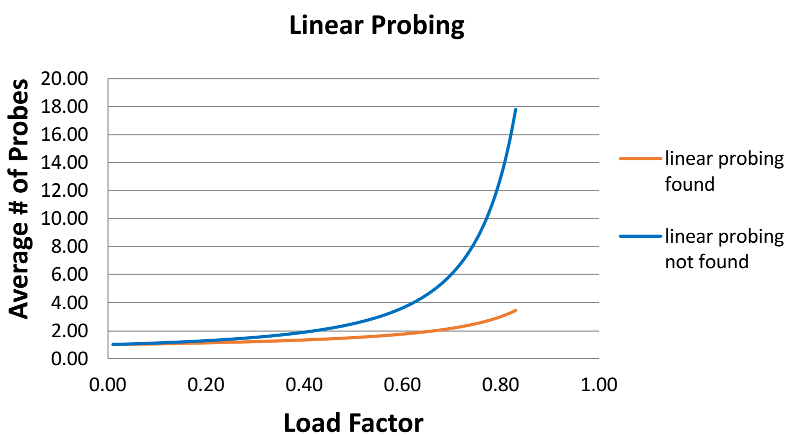
[R. Sedgewick]

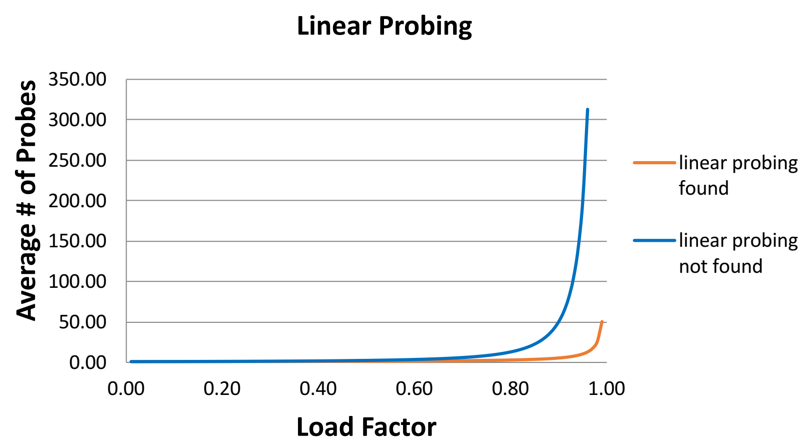
Analysis of Linear Probing

- For any $\lambda < 1$, linear probing will find an empty slot
  - It is "safe" in this sense: no infinite loop unless table is full
- Non-trivial facts we won't prove:
  Average # of probes given λ (in the limit as **`TableSize`** $\rightarrow \infty$)
  - Unsuccessful search: $\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$
  - Successful search: $\frac{1}{2}\left(1+\frac{1}{(1-\lambda)}\right)$
- This is pretty bad: need to leave sufficient empty space in the table to get decent performance (see chart)

Analysis: Linear Probing

- Linear-probing performance degrades rapidly as table gets full
  (Formula assumes "large table" but point remains)





- By comparison, chaining performance is linear in λ and has no trouble with $\boldsymbol{\lambda > 1}$

Any ideas for alternatives?

Open Addressing: Quadratic Probing

- We can avoid primary clustering by changing the probe function

  `(h(key) + f(i)) % TableSize`

- A common technique is quadratic probing: **`f(i) = i`^{2}**
  - So probe sequence is:
    - 0^{th} probe: `h(key) % TableSize`
    - 1^{st} probe:
    - 2^{nd} probe:
    - 3^{rd} probe:
    - …
    - i^{th} probe: `(h(key) + i`^{2}`) % TableSize`

- Intuition: Probes quickly "leave the neighborhood"

Quadratic Probing Example #1

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

TableSize = 10

Insert:

89

18

49

58

79

i^{th} probe: `(h(key) + i`^{2}`) % TableSize`

Quadratic Probing Example #2

0	
1	
2	
3	
4	
5	
6	

TableSize = 7

Insert:

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

i^{th} probe: `(h(key) + i²) % TableSize`

Quadratic Probing: Bad News, Good News

- Bad news:
  - Quadratic probing can cycle through the same full indices, never terminating despite table not being full
- Good news:
  - If `TableSize` is *prime* and $\lambda < ½$, then quadratic probing will find an empty slot in at most `TableSize/2` probes
  - So: If you keep $\lambda < ½$ and `TableSize` is *prime*, no need to detect cycles
  - Proof is posted online next to lecture slides
    - Also, slightly less detailed proof in textbook
    - Key fact: For prime `T` and `0 < i,j < T/2` where $i \neq j$, $(k + i^2) \% T \neq (k + j^2) \% T$ (i.e., no index repeat)

Clustering Part 2

- Quadratic probing does not suffer from primary clustering:
  no problem with keys initially hashing to the same neighborhood

- But it's no help if keys initially hash to the same index:

  This is called

- Can avoid secondary clustering

Open Addressing: Double Hashing

Idea:

- Given two good hash functions *h* and *g*, it is very unlikely that for some *key*, `h(key) == g(key)`

- So make the probe function **`f(i) = i*g(key)`**

Probe sequence:

- 0^{th} probe: `h(key) % TableSize`
- 1^{st} probe: `(h(key) + g(key))   % TableSize`
- 2^{nd} probe:
- 3^{rd} probe:
- …
- i^{th} probe: `(h(key) + i*g(key)) % TableSize`

Double Hashing Analysis

- Intuition: Because each probe is "jumping" by `g(key)` each time, we "leave the neighborhood" *and* "go different places from other initial collisions"

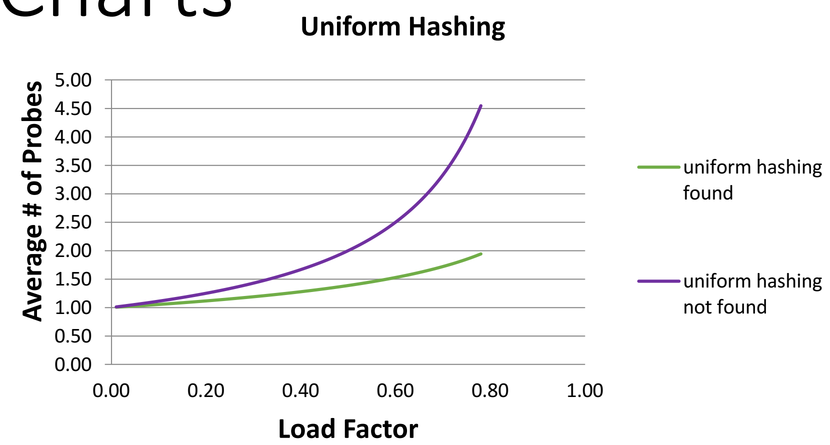
- Requirements for second hash function:

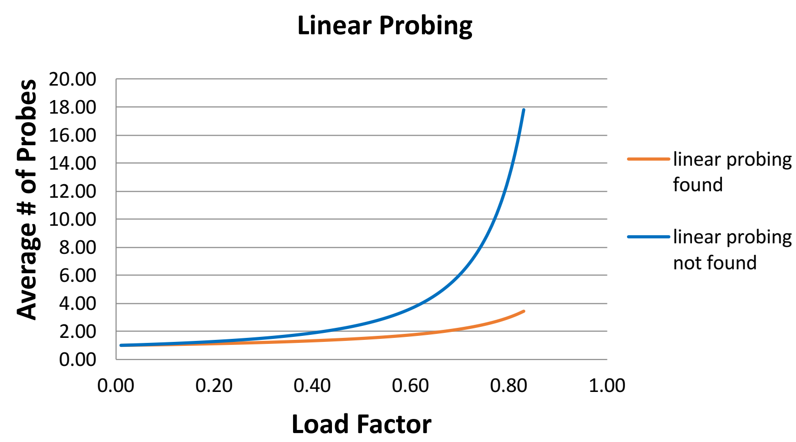
- Example of double hash function pair that works:
  - `h(key) = key % p`
  - `g(key) = q - (key % q)`
  - `2 < q < p`
  - `p` and `q` are prime

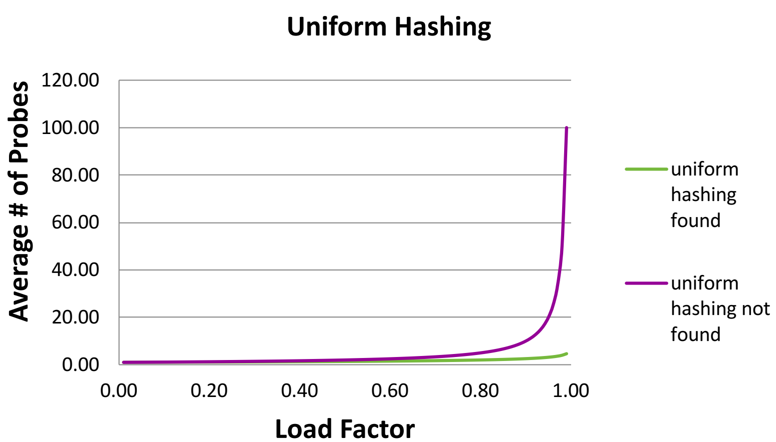
More Double Hashing Facts

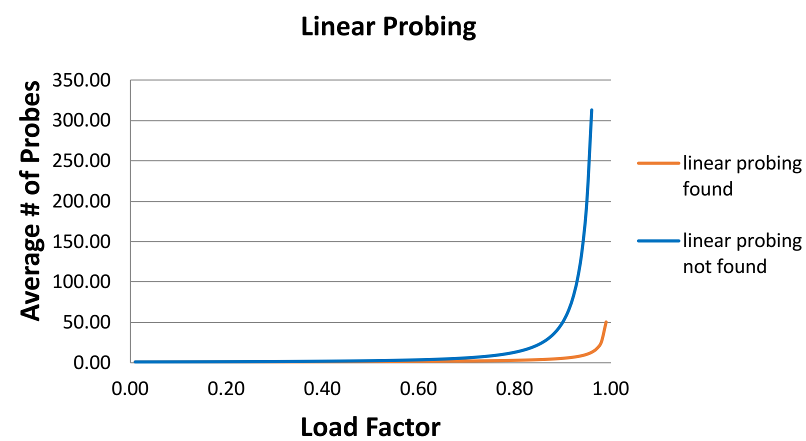
- Assume "uniform hashing"
  - Means probability of `g(key1) % p == g(key2) % p` is `1/p`
- Non-trivial facts we won't prove:
  Average # of probes given λ (in the limit as `TableSize` $\rightarrow \infty$)
  - Unsuccessful search (intuitive): $\dfrac{1}{1-\lambda}$
  - Successful search (less intuitive): $\dfrac{1}{\lambda}\log_e\left(\dfrac{1}{1-\lambda}\right)$
- Bottom line: unsuccessful bad (but not as bad as linear probing), but successful is not nearly as bad

Charts


Uniform Hashing
Average # of Probes
5.00
4.50
4.00
3.50
3.00
2.50
2.00
1.50
1.00
0.50
0.00
0.00
0.20
0.40
0.60
0.80
1.00
Load Factor
uniform hashing found
uniform hashing not found


Linear Probing
Average # of Probes
20.00
18.00
16.00
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
0.00
0.20
0.40
0.60
0.80
1.00
Load Factor
linear probing found
linear probing not found


Uniform Hashing
Average # of Probes
120.00
100.00
80.00
60.00
40.00
20.00
0.00
0.00
0.20
0.40
0.60
0.80
1.00
Load Factor
uniform hashing found
uniform hashing not found


Linear Probing
Average # of Probes
350.00
300.00
250.00
200.00
150.00
100.00
50.00
0.00
0.00
0.20
0.40
0.60
0.80
1.00
Load Factor
linear probing found
linear probing not found

Rehashing

Rehashing

- What do we do if the table gets too full?
- How do we copy over elements?

Rehashing

- What's "too full" in Separate Chaining?

- "Too full" for Open Addressing / Probing

Rehashing

- How big do we want to make the new table?

- Can keep a list of prime numbers in your code, since you likely won't grow more than 20-30 times ($2^{30} = 1,073,741,824$)