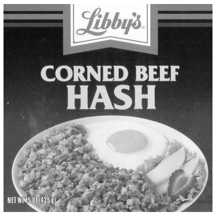


CSE 326: Data Structures Lecture #12 Hashing II

Henry Kautz
Winter 2002

Load Factor in Linear Probing

- For *any* $\lambda < 1$, 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:

$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$$

- Performance quickly degrades for $\lambda > 1/2$

Linear Probing – Expected # of Probes

Load factor	**failure**	**success**
.1	**1.11**	**1.06**
.2	**1.28**	**1.13**
.3	**1.52**	**1.21**
.4	**1.89**	**1.33**
.5	**2.5**	**1.50**
.6	**3.6**	**1.75**
.7	**6.0**	**2.17**
.8	**13.0**	**3.0**
.9	**50.5**	**5.5**

Open Addressing II: Quadratic Probing

- Main Idea: Spread out the search for an empty slot – Increment by i^2 instead of i

- $h_i(X) = (Hash(X) + i^2)$ % *TableSize*
  - h0(X) = Hash(X) % TableSize
  - h1(X) = Hash(X) + 1 % TableSize
  - h2(X) = Hash(X) + 4 % TableSize
  - h3(X) = Hash(X) + 9 % TableSize

Quadratic Probing Example

	insert(14) 14%7 = 0	insert(7) 8%7 = 1	insert(21) 21%7 =0	insert(2) 2%7 = 2
0	14	14	14	14
1		8	8	8
2				2
3				
4			21	21
5				
6				
probes:	1	1	3	1

Problem With Quadratic Probing

	insert(14) 14%7 = 0	insert(8) 8%7 = 1	insert(21) 21%7 =0	insert(2) 2%7 = 2	insert(7) 7%7 = 0
0	14	14	14	14	14
1		8	8	8	8
2				2	2
3					
4			21	21	21
5					
6					
probes:	1	1	3	1	??

Load Factor in Quadratic Probing

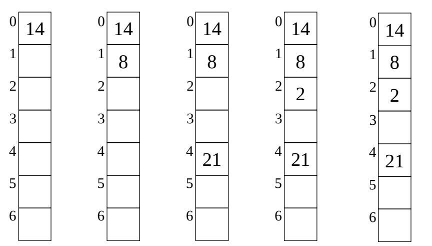
- **Theorem**: If TableSize is prime and λ ≤ ½, quadratic probing *will* find an empty slot; for greater λ, *might not*
- With load factors near ½ the expected number of probes is about 1.5
- Don't get clustering from *similar* keys (**primary** clustering), still get clustering from *identical* keys (**secondary** clustering)

Open Addressing III: Double Hashing

- **Idea**: Spread out the search for an empty slot by using a second hash function
  - *No primary or secondary clustering*
- $h_i(X) = (Hash_1(X) + i \cdot Hash_2(X)) \bmod$ *TableSize* for i = 0, 1, 2, …
- Good choice of $Hash_2(X)$ can guarantee does not get "stuck" as long as λ < 1
  - Integer keys:
    $Hash_2(X) = R – (X \bmod R)$
    where R is a prime smaller than *TableSize*

Double Hashing Example

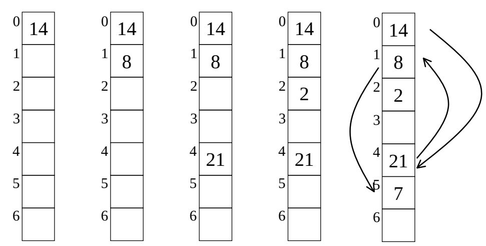
insert(14)	insert(8)	insert(21)	insert(2)	insert(7)
14%7 = 0	8%7 = 1	21%7 =0 5-(21%5)=4	2%7 = 2	7%7 = 0 5-(21%5)=4



probes:	1	1	2	1	??

Double Hashing Example

insert(14)	insert(8)	insert(21)	insert(2)	insert(7)
14%7 = 0	8%7 = 1	21%7 =0 5-(21%5)=4	2%7 = 2	7%7 = 0 5-(21%5)=4



probes:	1	1	2	1	4

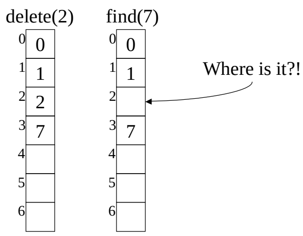
Load Factor in Double Hashing

- For *any* λ < 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
- Becomes very costly as λ nears 1. In practice, *slower* than quadratic probing if λ ≤ ½.

Deletion with Separate Chaining

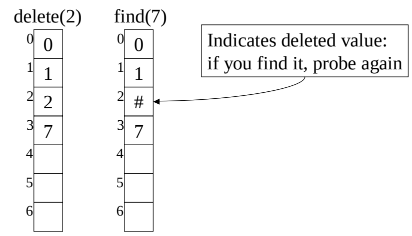
Why is this slide blank?

Deletion in Open Addressing



What should we do instead?

Lazy Deletion



But *now* what is the problem?

The Squished Pigeon Principle

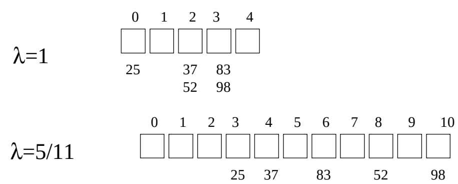
- An insert using open addressing *cannot* work with a load factor of 1 or more.
  - Quadratic probing can *fail* if $\lambda > ½$
  - Linear probing and double hashing *slow* if $\lambda > ½$
  - Lazy deletion never frees space
- Separate chaining becomes slow once $\lambda > 1$
  - Eventually becomes a linear search of long chains
- How can we relieve the pressure on the pigeons?

REHASH!

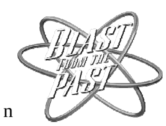
Rehashing Example

Separate chaining

$h_1(x) = x \bmod 5$ rehashes to $h_2(x) = x \bmod 11$



Stretchy Stack Amortized Analysis

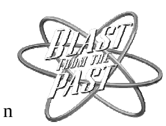


- Consider sequence of n operations
  push(3); push(19); push(2); …
- What is the max number of stretches? log n
- What is the total time?
  - let's say a regular push takes time *a*, and stretching an array contain *k* elements takes time *bk*.

$$an + b(1+2+4+8+...+n) = an + b\sum_{i=o}^{\log n} 2^i$$
$$= an + b(2n-1)$$

- Amortized time = $(an+b(2n-1))/n = O(1)$

Rehashing Amortized Analysis



- Consider sequence of n operations
  insert(3); insert(19); insert(2); …
- What is the max number of rehashes? log n
- What is the total time?
  - let's say a regular hash takes time *a*, and rehashing an array contain *k* elements takes time *bk*.

$$an + b(1+2+4+8+...+n) = an + b\sum_{i=o}^{\log n} 2^i$$
$$= an + b(2n-1)$$

- Amortized time = $(an+b(2n-1))/n = O(1)$

Rehashing without Stretching

- Suppose input is a mix of inserts and deletes
  - Never more than TableSize/2 active keys
  - Rehash when λ=1 (half the table must be deletions)
- Worst-case sequence:
  - T/2 inserts, T/2 deletes, T/2 inserts, Rehash,
    T/2 deletes, T/2 inserts, Rehash, …
- Rehashing at most doubles the amount of work – still O(1)

Case Study

- Spelling dictionary
  - 30,000 words
  - static
  - arbitrary(ish) preprocessing time
- Goals
  - fast spell checking
  - minimal storage
- Practical notes
  - almost all searches are successful
  - words average about 8 characters in length
  - 30,000 words at 8 bytes/word is 1/4 MB
  - pointers are 4 bytes
  - there are *many* regularities in the structure of English words

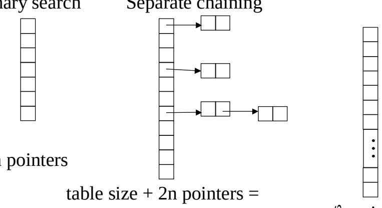
Why?

Solutions

- Solutions
  - sorted array + binary search
  - separate chaining
  - open addressing + linear probing

Storage

- Assume words are strings and entries are pointers to strings



Analysis

- Binary search
  - storage: n pointers + words = 360KB
  - time: $\log_2 n \le 15$ probes per access, worst case
- Separate chaining
  - storage: 2n + n/λ pointers + words (λ = 1 $\Rightarrow$ 600KB)
  - time: 1 + λ/2 probes per access on average (λ = 1 $\Rightarrow$ 1.5)
- Open addressing
  - storage: n/λ pointers + words (λ = 0.5 $\Rightarrow$ 480KB)
  - time: $\frac{1}{2}\left(1+\frac{1}{(1-\lambda)}\right)$ probes per access on average (λ = 0.5 $\Rightarrow$ 1.5)

Which one should we use?

A Random Hash…

- Universal hashing
  - Given a particular input, pick a hash function parameterized by some random number
  - Useful in proving average case results – instead of randomizing over inputs, randomize over choice of hash function
- Minimal perfect hash function: one that hashes a given set of n keys into a table of size n with no collisions
  - Always exist
  - Might have to search large space of parameterized hash functions to find
  - Application: compilers
- One way hash functions
  - Used in cryptography
  - Hard (intractable) to invert: given just the hash value, recover the key

Coming Up

- Wednesday: Nick leads the class
- Try all the homework problems BEFORE Thursday, so you can ask questions in section!
- Friday: Midterm