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 \( \lambda > 1/2 \)

Open Addressing II: Quadratic Probing
- Main Idea: Spread out the search for an empty slot – Increment by \( i^2 \) instead of \( i \)
- \( h_i(X) = (\text{Hash}(X) + i^2) \mod \text{TableSize} \)
  - \( h_0(X) = \text{Hash}(X) \mod \text{TableSize} \)
  - \( h_1(X) = \text{Hash}(X) + 1 \mod \text{TableSize} \)
  - \( h_2(X) = \text{Hash}(X) + 4 \mod \text{TableSize} \)
  - \( h_3(X) = \text{Hash}(X) + 9 \mod \text{TableSize} \)

Linear Probing – Expected # of Probes

<table>
<thead>
<tr>
<th>Load factor</th>
<th>failure</th>
<th>success</th>
</tr>
</thead>
<tbody>
<tr>
<td>.1</td>
<td>1.11</td>
<td>1.06</td>
</tr>
<tr>
<td>.2</td>
<td>1.28</td>
<td>1.13</td>
</tr>
<tr>
<td>.3</td>
<td>1.52</td>
<td>1.21</td>
</tr>
<tr>
<td>.4</td>
<td>1.89</td>
<td>1.33</td>
</tr>
<tr>
<td>.5</td>
<td>2.5</td>
<td>1.50</td>
</tr>
<tr>
<td>.6</td>
<td>3.6</td>
<td>1.75</td>
</tr>
<tr>
<td>.7</td>
<td>6.0</td>
<td>2.17</td>
</tr>
<tr>
<td>.8</td>
<td>13.0</td>
<td>3.0</td>
</tr>
<tr>
<td>.9</td>
<td>50.5</td>
<td>5.5</td>
</tr>
</tbody>
</table>

Open Addressing II: Quadratic Probing

<table>
<thead>
<tr>
<th>Insertions</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 ( \mod 7 = 0 )</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Problems With Quadratic Probing

<table>
<thead>
<tr>
<th>Insertions</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 ( \mod 7 = 0 )</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Quadratic Probing Example

Probes:
1 1 3 1
Load Factor in Quadratic Probing

- **Theorem**: If TableSize is prime and $\lambda \leq \frac{1}{2}$, quadratic probing will find an empty slot; for greater $\lambda$, might not
- With load factors near $\frac{1}{2}$ 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
  - $h_i(X) = (Hash_i(X) + i \cdot Hash_2(X)) \mod \text{TableSize}$
    - for $i = 0, 1, 2, \ldots$
  - Good choice of $Hash_2(X)$ can guarantee does not get “stuck” as long as $\lambda < 1$
    - Integer keys:
      - $Hash_{prim}(X) = R - (X \mod R)$
      - where R is a prime smaller than TableSize

Double Hashing Example

<table>
<thead>
<tr>
<th>insert(14)</th>
<th>insert(8)</th>
<th>insert(21)</th>
<th>insert(2)</th>
<th>insert(7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>8</td>
<td>21</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

probes: 1 1 2 1 ??

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
- Becomes very costly as $\lambda$ nears 1. In practice, slower than quadratic probing if $\lambda \leq \frac{1}{2}$.

Deletion with Separate Chaining

Why is this slide blank?
Deletion in Open Addressing

```
delete(2) find(7)
0 1 2 3 4 5 6
0 1 2 3 4 5 6
```

Where is it?!

What should we do instead?

Lazy Deletion

```
delete(2) find(7)
0 1 2 3 4 5 6
0 1 2 3 4 5 6
```

Indicates deleted value: if you find it, probe again

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 \( \lambda > \frac{1}{2} \)
  - Linear probing and double hashing slow if \( \lambda > \frac{1}{2} \)
  - 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
\( h_1(x) = x \mod 5 \) rehashes to \( h_2(x) = x \mod 11 \)

```
λ=1
0 1 2 3 4
25 37 85 52 98
λ=5/11
0 1 2 3 4 5 6 7 8 9 10
25 37 83 52 98
```

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 \( bk \).

\[
\begin{align*}
an + b(1 + 2 + 4 + 8 + \ldots + n) &= an + b \sum_{i=0}^{\log n} 2^i \\
&= an + b(2n - 1)
\end{align*}
\]

- 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 \( bk \).

\[
\begin{align*}
an + b(1 + 2 + 4 + 8 + \ldots + n) &= an + b \sum_{i=0}^{\log n} 2^i \\
&= an + b(2n - 1)
\end{align*}
\]

- 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 \( \lambda = 1 \) (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

Solutions

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

Storage

- Assume words are strings and entries are pointers to strings

  - Array + binary search
  - Separate chaining
  - Open addressing

  \[
  \text{table size} + 2n \text{ pointers} = \frac{n}{\lambda} + 2n
  \]

Analysis

- Binary search
  - storage: \( n \) pointers + words \( = 360 \text{KB} \)
  - time: \( \log_2 n \leq 15 \) probes per access, worst case

- Separate chaining
  - storage: \( 2n + n\lambda \) pointers + words \( (\lambda = 1 \Rightarrow 600 \text{KB}) \)
  - time: \( 1 + \lambda/2 \) probes per access on average \( (\lambda = 1 \Rightarrow 1.5) \)

- Open addressing
  - storage: \( n\lambda \) pointers + words \( (\lambda = 0.5 \Rightarrow 480 \text{KB}) \)
  - time: \( \frac{1}{\lambda} \left(1 - \frac{1}{\lambda-1}\right) \) probes per access on average \( (\lambda = 0.5 \Rightarrow 1.5) \)

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

Which one should we use?
Coming Up

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