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*
Hash Table Collisions: Review

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

• We try to *avoid* them by having a good hash function (unique index)

• 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:

\[
\text{insert } 10, 22, 107, 12, 42 \\
\text{and } \textbf{TableSize} = 10
\]

(for illustrative purposes, we’re inserting hash values)
Separate Chaining: Worst-Case

What’s the worst-case scenario for `find`?
all keys indexed to the same bucket

What’s the worst-case running time for `find`?
$O(n)$ linear

But only with really bad luck or really bad hash function
$O(n)$ not worth avoiding worst-case
Separate Chaining: Further Analysis

• How can find become slow when we have a good hash function?
  
  \# elements \gg table size

  mean long chains

• How can we reduce its likelihood?

  Maintain a good ratio of \# elements to the table size
  (resize the table as needed)
Rigorous Analysis: Load Factor

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

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

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 $\lambda/2$ items
Rigorous Analysis: Load Factor

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

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

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

- short chains (not too high)
- efficient use of table space (not too low)

So for separate chaining, a good load factor is $1, 1.5, \text{ or } 2$
Open Addressing / Probing

Another family of collision resolution schemes
Idea: use empty space in the table

- If \( h(key) \) is already full,
  - try \((h(key) + 1) \mod \text{TableSize}\). If full,
  - try \((h(key) + 2) \mod \text{TableSize}\). If full,
  - try \((h(key) + 3) \mod \text{TableSize}\). If full...

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

\[ 38 \mod 10 = 8 \]
Open Addressing Terminology

Trying the next spot is called \textit{probing} \hfill (also called \textit{open addressing}) \hfill

\begin{itemize}
  \item We just did \textit{linear probing} \hfill \textit{open addressing} \hfill
    \[ i^{th \ probe \ was} \ (h(key) + i) \ % \ TableSize \]
  \item In general have some \textit{probe function} \( f \) and use
    \[(h(key) + f(i)) \ % \ TableSize \]
\end{itemize}
Dictionary Operations with Open Addressing

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

**What about find?**
- must use same probe function to "retrace the trail"
- unsuccessful search when reach empty bucket

**What about delete?**
- use "lazy" deletion
  to replace element with marker/flag
to say "no data here, but keep probing"

• 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 $h(k) = k \text{ mod } 10$ and linear probing. What is the resultant hash table?

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>13</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>18</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(A) | (B) | (C) | (D)
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 primary clustering
• 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 $\lambda$ (in the limit as $\text{TableSize} \to \infty$)
    • 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 $\lambda$ and has no trouble with $\lambda > 1$
Any ideas for alternatives?

Different probe function!
Open Addressing: Quadratic Probing

• We can avoid primary clustering by changing the probe function
  \[ (h(key) + f(i)) \mod \text{TableSize} \]

• A common technique is quadratic probing: \( f(i) = i^2 \)
  • So probe sequence is:
    • 0th probe: \( h(key) \mod \text{TableSize} \)
    • 1st probe: \( (h(key) + 1) \mod \text{TableSize} \)
    • 2nd probe: \( (h(key) + 4) \mod \text{TableSize} \)
    • 3rd probe: \( (h(key) + 9) \mod \text{TableSize} \)
    • ...
    • \( j^{\text{th}} \) probe: \( (h(key) + i^2) \mod \text{TableSize} \)

• Intuition: Probes quickly “leave the neighborhood”
Quadratic Probing Example #1

Table Size $= 10$

Insert:
- 89
- 18
- 49
- 58
- 79

$i^{th}$ probe: $(h(key) + i^2) \mod \text{TableSize}$
Quadratic Probing Example #2

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)

Yikes! For all $n$, $(n^2 + 5) \mod 7$ is 0, 2, 5, 6, 6!

$i^{th}$ probe: $(h(key) + i^2) \mod \text{TableSize}$