CSE 326: Data Structures
Hash Tables

Steve Seitz
Winter 2009

Announcements (2/11/09)

- Project 2B due today.
- Homework #4 due on Friday, start of class
- Project #3 assigned on Friday
  - Partner signups NOW - midnight Friday
- Section: project warm-up, midterms returned, ...

- Reading for this lecture: Chapter 5.

Hash Tables

- Find, insert, delete: constant time on average!
- A hash table is an array of some fixed size.
- General idea:

  hash function: \( h(K) \)

  key space (e.g., integers, strings)

  \[
  \begin{array}{c}
  \text{index} = h(K) \\
  \hline
  0 & \ldots & \text{TableSize - 1} \\
  \hline
  \end{array}
  \]

  hash table

Hash Tables

Key space of size \( M \), but we only want to store subset of size \( N \), where \( N << M \).
- Keys are identifiers in programs. Compiler keeps track of them in a symbol table.
- Keys are student names. We want to look up student records quickly by name.
- Keys are chess configurations in a chess playing program.
- Keys are URLs in a database of web pages.
### Simple Integer Hash Functions

- Key space = integers
- TableSize = 10

\[ h(K) = K \mod 10 \]  
\[ (K \mod 10) \]

- Insert: 7, 18, 41, 34

### Aside: Properties of Mod

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

\[ h(K) = \text{function}(K) \mod \text{TableSize} \]

(In the previous examples, \( \text{function}(K) = K \).)

It’s worth noting a couple properties of the mod function:

- \( (a + b) \mod c = [(a \mod c) + (b \mod c)] \mod c \)
- \( (a \times b) \mod c = [(a \mod c) \times (b \mod c)] \mod c \)
- \( a \mod c = b \mod c \rightarrow (a - b) \mod c = 0 \)

### String Hash Functions?

Key space = strings

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

\[ T = h(s_i) \mod x \]

\[ \sum_{i=0}^{n-1} s_i \mod 7 \]

- spot, stop, tops, ...
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. $h(K) = s_0 \mod \text{TableSize}$

2. $h(K) = \left( \sum_{i=0}^{m-1} s_i \right) \mod \text{TableSize}$

3. $h(K) = \left( \sum_{i=0}^{m-1} s_i \cdot 128^i \right) \mod \text{TableSize}$
   $= s_0 + s_1 \cdot 128 + s_2 \cdot 128^2$

Hash Function Desiderata

What are some desirable properties for a hash function?

- Few collisions
- Even distribution = few peaks, modes
- Fast to compute
- Repeatability

Designing Hash Functions

We’ve seen a few possibilities. The simplest is **modular hashing**:

$$h(K) = K \mod P$$

where $P$ is usually just 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 $P$? $2^{11}$

A Fancier Hash Function

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

Instead, we can work on designing a really good hash function:

```java
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 can we cope with collisions?

Separate Chaining:

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

Analysis of Separate Chaining

The load factor, $\lambda$, of a hash table is

$$\lambda = \frac{N}{\text{Table Size}}$$

Separate chaining: $\lambda =$ average # of elems per bucket

Average cost of:
- Unsuccessful find? $\lambda$
- Successful find? $\lambda/2$
- Insert? $\lambda$

Alternative: Use Empty Space in the Table

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:
- 0th probe = h(k) % TableSize
- 1st probe = (h(k) + f(1)) % TableSize
- 2nd probe = (h(k) + f(2)) % TableSize
- ...
- ith probe = (h(k) + f(i)) % TableSize

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

Open Addressing Example, Revisited

<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>109</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>38</td>
</tr>
<tr>
<td>9</td>
<td>19</td>
</tr>
</tbody>
</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...

What is f(i)? ≥ 0

Terminology Alert!

- Separate chaining is sometimes called open hashing.
- Open addressing is sometimes called closed hashing.

Linear Probing

f(i) = i

- Probe sequence:
  - 0th probe = h(K) % TableSize
  - 1st probe = (h(K) + 1) % TableSize
  - 2nd probe = (h(K) + 2) % TableSize
  - ...
  - ith probe = (h(K) + i) % TableSize
Linear Probing – Clustering

Analysis of Linear Probing

- For any $\lambda < 1$, linear probing will find an empty slot
- Expected # of probes (for large table sizes)
  - unsuccessful search: $\frac{1}{2} \left( 1 + \frac{1}{(1-\lambda)^2} \right) = 2.5$ if $\lambda = 0.5$
  - $50.5$ if $\lambda = 0.2$
  - 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

$f(i) = i^2$

- Probe sequence:
  - 0th probe = $h(K) \%$ TableSize
  - 1st probe = $(h(K) + 1) \%$ TableSize
  - 2nd probe = $(h(K) + 4) \%$ TableSize
  - 3rd probe = $(h(K) + 9) \%$ TableSize
  - $i$th probe = $(h(K) + i^2) \%$ TableSize
Quadratic Probing Example

<table>
<thead>
<tr>
<th>0</th>
<th>49 (+1)</th>
<th>Insert:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>89</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>58 (+4)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>79 (+4)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>87</td>
<td></td>
</tr>
</tbody>
</table>

Another Quadratic Probing Example

TableSize = 7
h(K) = K % 7

<table>
<thead>
<tr>
<th>0</th>
<th>48</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>55</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>40</td>
</tr>
<tr>
<td>6</td>
<td>76</td>
</tr>
</tbody>
</table>

Quadratic Probing: Success guarantee for $\lambda < \frac{1}{2}$

Assertion #1: If $T = \text{TableSize}$ is prime and $\lambda < \frac{1}{2}$, then quadratic probing will find an empty slot in $T/2$ probes or fewer.

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

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

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

Quadratic Probing: Success guarantee for $\lambda < \frac{1}{2}$

We can prove assertion #2 by contradiction.

Suppose that for some $i \neq j$, $0 \leq i, j \leq T/2$, prime $T$:

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

Then $T$ is not prime.


determinant $= (i^2 - j^2)^T = 0$
Quadratic Probing: Properties

- For any $\lambda < 1/2$, quadratic probing will find an empty slot; for bigger $\lambda$, quadratic probing may find a slot.

- Quadratic probing does not suffer from primary clustering: keys hashing to the same area are not bad.

- But what about keys that hash to the same spot? — Secondary Clustering!

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:
  0th probe = $h(K) \mod \text{TableSize}$
  1st probe = $(h(K) + g(K)) \mod \text{TableSize}$
  2nd probe = $(h(K) + 2g(K)) \mod \text{TableSize}$
  3rd probe = $(h(K) + 3g(K)) \mod \text{TableSize}$
  \[\vdots\]
  ith probe = $(h(K) + ig(K)) \mod \text{TableSize}$

Another Example of Double Hashing

<table>
<thead>
<tr>
<th>Hash Functions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T = \text{TableSize} = 10$</td>
</tr>
<tr>
<td>$h(K) = K \mod T$</td>
</tr>
<tr>
<td>$g(K) = 1 + \left\lfloor \left(K/T \mod (T-1) \right) \right\rfloor$</td>
</tr>
</tbody>
</table>

Insert these values into the hash table in this order. Resolve any collisions with double hashing:

- $13$
- $28$
- $33 \quad h = 3, g = 4$
- $147 \quad h = 7, g = 6$
- $43 \quad h = 3, g = 5$
Analysis of Double Hashing

- Double hashing is safe for $\lambda < 1$ for at least one 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

Can we do something similar for open addressing?

- Delete
- Find
- Insert

$h(k) = k \% 7$
Linear probing

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>30</td>
<td></td>
<td>76</td>
<td>76</td>
</tr>
<tr>
<td>4</td>
<td>59</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Rehashing

**Idea:** 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?** $O(\lambda)$
Rehashing Example

- Separate chaining example:
  \[ h_1(x) = x \mod 5 \text{ rehashes to } h_2(x) = x \mod 11. \]

- Hash table sizes:
  \( \lambda = 1 \)
  \[
  \begin{array}{cccccc}
  h_1 &=& 0 & 1 & 2 & 3 & 4 \\
  25 & 37 & 83 & 52 & 98 \\
  \end{array}
  \]

- Hash table sizes:
  \( \lambda = 5/11 \)
  \[
  \begin{array}{cccccccccccc}
  h_1 &=& 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
  25 & 37 & 83 & 52 & 98 \\
  \end{array}
  \]

Rehashing Picture

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

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |
| hashes | rehashes |

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? \( O(1) \)
- 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 textbook.)

Amortized Analysis of Rehashing

- Cost of inserting \( n \) keys is \(<3n\)
- \( 2^k + 1 \leq n \leq 2^{k+1} \)
  - Hashes = \( n \)
  - Rehashes = \( 2 + 2^2 + \ldots + 2^k = 2^{k+1} - 2 \)
  - Total = \( n + 2^{k+1} - 2 \leq 3n \)
- Example
  - \( n = 33 \), Total = \( 33 + 64 - 2 = 95 < 99 \)
Java hashCode() Method

- Class Object defines a hashCode method
  - Intent: returns a suitable hashcode for the object
  - Result is arbitrary int; must scale to fit a hash table (e.g. obj.hashCode() % nBuckets)
  - Used by collection classes like HashMap
- Classes should override with calculation appropriate for instances of the class
  - Calculation should involve semantically “significant” fields of objects

hashCode() and equals()

- To work right, particularly with collection classes like HashMap, hashCode() and equals() must obey this rule:
  - if a.equals(b) then it must be true that a.hashCode() == b.hashCode()
  - Why?
- Reverse is not required