CSE 326: Data Structures
Hash Tables

Brian Curless
Spring 2008

Announcements (5/7/08)

- Project 2B due today.
- Homework #4 due on Friday, beginning of class
- Project #3 assigned on Friday
  - Partner signups by 3pm 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:
  \[ \text{index} = h(K) \]

  hash table

  \[
  \begin{array}{|c|}
  \hline
  0 & \text{TableSize} - 1 \\
  \hline
  \end{array}
  \]

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

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
- \( h(K) = K \mod 10 \)
- **Insert**: 7, 18, 41, 94

#### 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:

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

### Simple Integer Hash Functions

- key space = integers
- TableSize = 6
- \( h(K) = K \mod 6 \)
- **Insert**: 7, 18, 41, 34

### Some String Hash Functions

- key space = strings
- \( s_i \in [0 \ldots 127] \)
  - \( K = s_0 \; s_1 \; s_2 \ldots \; s_{m-1} \) (where \( s_i \) are characters)

1. \( h(K) = s_0 \mod \text{TableSize} \)

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

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

   \[\approx s_0 + s_1 \cdot 128 + s_2 \cdot 128^2 + \ldots\]
Hash Function Desiderata
What are some desirable properties for a hash function?

- Evenly distributes values throughout the table.
- Minimizes collisions.
- Fast to compute.

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 \( P \)? \( 2^l \)

A Fancier Hash Function
Some experimental results indicate that modular hash functions with prime table 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_length; 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?

- **Open addressing**: Look at next spot in table, if empty, use if full, go to next
- **Separate chaining**: List at each table entry
Separate Chaining

Table Size = 10

Insert:
10
22
107
12
42

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: $\lambda = \text{average # of elems per bucket}$

Average cost of:
- Unsuccessful find? $\lambda$
- Successful find? $1 + \frac{\lambda}{2}$
- Insert? $1$

Alternative: Use Empty Space in the Table

$h(k) = k \mod 10$

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...

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:

$0^{th}$ probe = $h(k) \mod \text{Table Size}$
$1^{st}$ probe = $(h(k) + f(1)) \mod \text{Table Size}$
$2^{nd}$ probe = $(h(k) + f(2)) \mod \text{Table Size}$

$i^{th}$ probe = $(h(k) + f(i)) \mod \text{Table Size}$

$f(i)$ is the probing function. We'll look at a few...
**Terminology Alert!**

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

### Open Addressing Example, Revisited

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
</tr>
<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 \( f(i) \)?**

\[ f(i) = i \]

---

**Linear Probing**

- \( f(i) = i \)

**Probe sequence:**
- 0\(^{th}\) probe = \( h(K) \) \( \% \) TableSize
- 1\(^{st}\) probe = \( (h(K) + 1) \) \( \% \) TableSize
- 2\(^{nd}\) probe = \( (h(K) + 2) \) \( \% \) TableSize
- ...
- \( i^{th}\) probe = \( (h(K) + i) \) \( \% \) TableSize

---

**Linear Probing – Clustering**

- **No collision**
- **Collision in small cluster**
- **Collision in large cluster**
Analysis of Linear Probing

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

- Linear probing suffers from primary clustering.
- Performance quickly degrades for $\lambda > 1/2$.

Quadatic Probing

$$f(i) = i^2$$

- Probe sequence:
  - $0^{th}$ probe = $h(K) \mod \text{TableSize}$
  - $1^{st}$ probe = $(h(K) + 1) \mod \text{TableSize}$
  - $2^{nd}$ probe = $(h(K) + 4) \mod \text{TableSize}$
  - $3^{rd}$ probe = $(h(K) + 9) \mod \text{TableSize}$
  - $i^{th}$ probe = $(h(K) + i^2) \mod \text{TableSize}$

Quadatic Probing Example

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Insert:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>89</td>
</tr>
<tr>
<td>2</td>
<td>58</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>79</td>
<td>49</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>58</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>79</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>89</td>
<td></td>
</tr>
</tbody>
</table>

Another Quadatic Probing Example

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>TableSize = 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>48</td>
<td>$h(K) = K \mod 7$</td>
</tr>
<tr>
<td>1</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>76</td>
<td>insert(76) 76 % 7 = 6</td>
</tr>
<tr>
<td>3</td>
<td>49</td>
<td>insert(49) 49 % 7 = 5</td>
</tr>
<tr>
<td>4</td>
<td>48</td>
<td>insert(48) 48 % 7 = 6</td>
</tr>
<tr>
<td>5</td>
<td>55</td>
<td>insert(55) 55 % 7 = 6</td>
</tr>
<tr>
<td>6</td>
<td>47</td>
<td>insert(47) 47 % 7 = 6</td>
</tr>
</tbody>
</table>

Quadatic Probing Example 2

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>TableSize = 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td></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: If the following holds

$$ (h(K) + i^2) \not\equiv (h(K) + j^2) \pmod{T} $$

for prime $T$ and all $0 \leq i, j \leq T/2$ and $i \neq j$,
then assertion #1 is true.

Assertion #3: If assertion #2 is true, then so is assertion #1.

Quadratic Probing: Properties

- For any $\lambda < \frac{1}{2}$, 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:

$$ f(i) = i \ast g(K) $$

- Probe sequence:
  
  - $0^{th}$ probe = $h(K) \% \text{TableSize}$
  - $1^{st}$ probe = $(h(K) + g(K)) \% \text{TableSize}$
  - $2^{nd}$ probe = $(h(K) + 2\ast g(K)) \% \text{TableSize}$
  - $3^{rd}$ probe = $(h(K) + 3\ast g(K)) \% \text{TableSize}$
  - $i^{th}$ probe = $(h(K) + i\ast g(K)) \% \text{TableSize}$
Double Hashing Example

TableSize = 7
h(K) = K % 7
g(K) = 5 - (K % 5)

Insert(76) 76 % 7 = 6 and 5 - 76 % 5 =
Insert(93) 93 % 7 = 2 and 5 - 93 % 5 =
Insert(40) 40 % 7 = 5 and 5 - 40 % 5 =
Insert(47) 47 % 7 = 5 and 5 - 47 % 5 = 3
Insert(10) 10 % 7 = 3 and 5 - 10 % 5 =
Insert(55) 55 % 7 = 6 and 5 - 55 % 5 = 5

Another Example of Double Hashing

Hash Functions:
T = TableSize = 10
h(K) = K % T
g(K) = 1 + (K/T) % (T-1)

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

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td>83</td>
<td>28</td>
<td>147</td>
</tr>
</tbody>
</table>

Analysis of Double Hashing

- Double hashing is safe for $\lambda < 1$ for at least one case:
  - $h(k) = k \mod p$
  - $g(k) = q - (k \mod q)$
  - $2 < q < p$, 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?

- find
- remove from list
Deletion in Open Addressing

Can we do something similar for open addressing?

- Delete → mark "occupied"
- Find
- Insert

h(k) = k % 7
Linear probing

Delete(23)
Find(59)
Insert(30)

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 (λ = 1)
  - Open addressing: half full (λ = 0.5)
  - When an insertion fails
  - Some other threshold
- Cost of a single rehashing?

Rehashing Example

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

\[ \lambda = 1 \]

\[ \begin{array}{cccc}
0 & 1 & 2 & 3 \\
25 & 37 & 83 & 52 \\
\end{array} \]

\[ \lambda = 5/11 \]

\[ \begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
25 & 37 & 83 & 52 & 98 \\
\end{array} \]

Rehashing Picture

- Starting with table of size 2, double when load factor > 1.
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 = $n + 2^{k+1} - 2 < 3n$
- Example
  - $n = 33$, Total = $33 + 64 - 2 = 95 < 99$

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?
- 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.)