CSE 332: Data Abstractions

Lecture 10: Hashing

Ruth Anderson
Spring 2014
Announcements

• **Project 2** – Phase A due *THIS* Thursday
• **Homework 3** – due Wednesday at the BEGINNING of lecture
Today

- Dictionaries
  - Hashing
Motivating Hash Tables

For dictionary with $n$ key/value pairs

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>find</th>
<th>delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted linked-list</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Unsorted array</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Sorted linked list</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Sorted array</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Balanced tree</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>
Hash Tables

- Aim for constant-time (i.e., $O(1)$) find, insert, and delete
  - “On average” under some reasonable assumptions
- A hash table is an array of some fixed size
- Basic idea:

  hash function:
  \[
  \text{index} = h(\text{key})
  \]

  key space (e.g., integers, strings)
Aside: Hash Tables vs. Balanced Trees

- In terms of a Dictionary ADT for just `insert`, `find`, `delete`, hash tables and balanced trees are just different data structures
  - Hash tables $O(1)$ on average (assuming few collisions)
  - Balanced trees $O(\log n)$ worst-case

- Constant-time is better, right?
  - Yes, but you need “hashing to behave” (must avoid collisions)
  - Yes, but `findMin`, `findMax`, `predecessor`, and `successor` go from $O(\log n)$ to $O(n)$, `printSorted` from $O(n)$ to $O(n \log n)$
    - Why your textbook considers this to be a different ADT
    - Not so important to argue over the definitions
Hash Tables

• There are $m$ possible keys ($m$ typically large, even infinite)
• We expect our table to have only $n$ items
• $n$ is much less than $m$ (often written $n << m$)

Many dictionaries have this property

  – Compiler: All possible identifiers allowed by the language vs. those used in some file of one program

  – Database: All possible student names vs. students enrolled

  – AI: All possible chess-board configurations vs. those considered by the current player

  – …
Hash functions

An ideal hash function:
• Is fast to compute
• “Rarely” hashes two “used” keys to the same index
  – Often impossible in theory; easy in practice
  – Will handle collisions a bit later

hash function: \[ \text{index} = h(\text{key}) \]

key space (e.g., integers, strings)
Who hashes what?

• Hash tables can be generic
  – To store elements of type E, we just need E to be:
    1. Comparable: order any two E (like with all dictionaries)
    2. Hashable: convert any E to an int

• When hash tables are a reusable library, the division of responsibility generally breaks down into two roles:

  ![Diagram showing the division of responsibility between client and hash table library]

• We will learn both roles, but most programmers “in the real world” spend more time as clients while understanding the library

4/21/2014
More on roles

Some ambiguity in terminology on which parts are “hashing”

Two roles must both contribute to minimizing collisions (heuristically)

• Client should aim for different ints for expected items
  – Avoid “wasting” any part of $E$ or the 32 bits of the int
• Library should aim for putting “similar” ints in different indices
  – conversion to index is almost always “mod table-size”
  – using prime numbers for table-size is common
What to hash?

• We will focus on two most common things to hash: ints and strings

• If you have objects with several fields, it is usually best to have most of the “identifying fields” contribute to the hash to avoid collisions

• Example:

```java
class Person {
    String first; String middle; String last;
    Date birthdate;
}
```

• An inherent trade-off: hashing-time vs. collision-avoidance
  – Bad idea(?): Only use first name
  – Good idea(?): Only use middle initial
  – Admittedly, what-to-hash is often an unprincipled guess 😞
**Hashing integers**

key space = integers

Simple hash function:

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

- Client: \( f(x) = x \)
- Library \( g(x) = f(x) \mod \text{TableSize} \)
- Fairly fast and natural

Example:

- TableSize = 10
- Insert 7, 18, 41, 34, 10
- (As usual, ignoring corresponding data)
Hashing integers (Soln)

key space = integers

Simple hash function:

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

- Client: \( f(x) = x \)
- Library \( g(x) = f(x) \mod \text{TableSize} \)
- Fairly fast and natural

Example:

- TableSize = 10
- Insert 7, 18, 41, 34, 10
- (As usual, ignoring corresponding data)
Collision-avoidance

- With \( x \% \text{ TableSize} \) the number of collisions depends on
  - the ints inserted (obviously)
  - \text{TableSize}

- Larger table-size tends to help, but not always
  - Example: 70, 24, 56, 43, 10
    with \text{TableSize} = 10 and \text{TableSize} = 60

- Technique: Pick table size to be prime. Why?
  - Real-life data tends to have a pattern
  - “Multiples of 61” are probably less likely than “multiples of 60”
  - We’ll see some collision strategies do better with prime size
More arguments for a prime table size

If TableSize is 60 and...
- Lots of data items are multiples of 5, wasting 80% of table
- Lots of data items are multiples of 10, wasting 90% of table
- Lots of data items are multiples of 2, wasting 50% of table

If TableSize is 61...
- Collisions can still happen, but 5, 10, 15, 20, ... will fill table
- Collisions can still happen but 10, 20, 30, 40, ... will fill table
- Collisions can still happen but 2, 4, 6, 8, ... will fill table

In general, if $x$ and $y$ are “co-prime” (means $\gcd(x,y)==1$), then

$$(a * x) \mod y == (b * x) \mod y \text{ if and only if } a \mod y == b \mod y$$

- Given table size $y$ and keys as multiples of $x$, we’ll get a decent distribution if $x$ & $y$ are co-prime
- So good to have a TableSize that has no common factors with any “likely pattern” $x$
What if the key is not an int?

- If keys aren’t *ints*, the **client** must convert to an *int*
  - Trade-off: speed and distinct keys hashing to distinct *ints*

- Common and important example: Strings
  - Key space $K = s_0s_1s_2...s_{m-1}$
    - where $s_i$ are chars: $s_i \in [0,256]$
  - Some choices: Which avoid collisions best?

1. $h(K) = s_0$

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

3. $h(K) = \left( \sum_{i=0}^{m-1} s_i \cdot 37^i \right)$

Then on the **library** side we typically mod by Tablesize to find index into the table
Specializing hash functions

How might you hash differently if all your strings were web addresses (URLs)?
Aside: Combining hash functions

A few rules of thumb / tricks:

1. Use all 32 bits (careful, that includes negative numbers)

2. Use different overlapping bits for different parts of the hash
   – This is why a factor of $37^i$ works better than $256^i$
   – Example: “abcde” and “ebcda”

3. When smashing two hashes into one hash, use bitwise-xor
   – bitwise-and produces too many 0 bits
   – bitwise-or produces too many 1 bits

4. Rely on expertise of others; consult books and other resources

5. If keys are known ahead of time, choose a perfect hash
Collision resolution

Collision:

When two keys map to the same location in the hash table

We try to avoid it, but number-of-keys exceeds table size

So hash tables should support collision resolution

– Ideas?
Flavors of Collision Resolution

Separate Chaining

Open Addressing
- Linear Probing
- Quadratic Probing
- Double Hashing
**Separate Chaining**

Chaining: All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example: insert 10, 22, 107, 12, 42 with mod hashing and \texttt{TableSize} = 10
Separate Chaining

Chaining: All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example: insert 10, 22, 107, 12, 42 with mod hashing and TableSize = 10
Separate Chaining

Chaining: All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example: insert 10, 22, 107, 12, 42 with mod hashing and TableSize = 10
Separate Chaining

Chaining: All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example: insert 10, 22, 107, 12, 42 with mod hashing and TableSize = 10
Separate Chaining

| 0 | 10 /
|---|---|
| 1 | /
| 2 | 12 → 22 /
| 3 | /
| 4 | /
| 5 | /
| 6 | /
| 7 | 107 /
| 8 | /
| 9 | /

Chaining: All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example: insert 10, 22, 107, 12, 42 with mod hashing and **TableSize** = 10
Chaining: All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example: insert 10, 22, 107, 12, 42 with mod hashing and TableSize = 10

Worst case time for find?
Thoughts on separate chaining

• Worst-case time for find?
  – Linear
  – But only with really bad luck or bad hash function
  – So not worth avoiding (e.g., with balanced trees at each bucket)
    • Keep # of items in each bucket small
    • Overhead of AVL tree, etc. not worth it for small n

• Beyond asymptotic complexity, some “data-structure engineering”
  can improve constant factors
  – Linked list vs. array or a hybrid of the two
  – Move-to-front (part of Project 2)
  – Leave room for 1 element (or 2?) in the table itself, to optimize
    constant factors for the common case
    • A time-space trade-off…
Time vs. space (constant factors only here)
More rigorous separate chaining analysis

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

$$\lambda = \frac{N}{\text{TableSize}} \leftarrow \text{number of elements}$$

Under chaining, the average number of elements per bucket is ___

So if some inserts are followed by random finds, then on average:

• Each unsuccessful find compares against ____ items
• Each successful find compares against _____ items

• How big should TableSize be??
More rigorous separate chaining analysis

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

$$\lambda = \frac{N}{\text{TableSize}} \leftarrow \text{number of elements}$$

Under chaining, the average number of elements per bucket is $\lambda$

So if some inserts are followed by random finds, then on average:
- Each unsuccessful find compares against $\lambda$ items
- Each successful find compares against $\lambda/2$ items
- If $\lambda$ is low, find & insert likely to be O(1)
- We like to keep $\lambda$ around 1 for separate chaining
Load Factor?

\[ \lambda = \frac{n}{TableSize} = ? \]
Load Factor?

\[ \lambda = \frac{n}{TableSize} = \frac{5}{10} = 0.5 \]
Load Factor?

\[ \lambda = \frac{n}{\text{Table Size}} = ? \]
Load Factor?

\[ \lambda = \frac{n}{\text{TableSize}} = \frac{21}{10} = 2.1 \]
Separate Chaining Deletion?
Separate Chaining Deletion

- Not too bad
  - Find in table
  - Delete from bucket
- Say, delete 12
- Similar run-time as insert