Hash Tables

CS211
Fall 2000

Goal: Design a Dictionary

<table>
<thead>
<tr>
<th>Operations</th>
<th>Array implementation:</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert (key, value)</td>
<td>Uses an array of (key, value) pairs</td>
</tr>
<tr>
<td>remove (key)</td>
<td></td>
</tr>
<tr>
<td>get (key)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Unsorted</th>
<th>Sorted</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>remove</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>get</td>
<td>O(n)</td>
<td>O(log n)</td>
</tr>
</tbody>
</table>

n is the number of items currently held in the array

Direct Address Table

- An easy version of a Hash Table
- Assumes the key set is from a small Universe
- Example: Addresses on my street
  - Start at 1, go to 40
  - A few lots don't have houses
- For a Direct Address Table, we make an array as large as the Universe
- To find an entry, we just index to that entry of the array
- Dictionary operations all take O(1) time

What if the Universe is large?

- Idea is to re-use table entries via a hash function
  - h: U → [0, ..., m-1] where m = table size
- h must
  - Be easy to compute
  - Cause few collisions
  - Have equal probability for each table position

Typical situation:
U = all legal identifiers
Typical hash function:
h converts each letter to a number and we compute a function of these numbers

A Hashing Example

- Suppose each word below has the following hash code:
<table>
<thead>
<tr>
<th>word</th>
<th>hash code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jan</td>
<td>7</td>
</tr>
<tr>
<td>Feb</td>
<td>0</td>
</tr>
<tr>
<td>Mar</td>
<td>5</td>
</tr>
<tr>
<td>Apr</td>
<td>2</td>
</tr>
<tr>
<td>May</td>
<td>4</td>
</tr>
<tr>
<td>Jun</td>
<td>7</td>
</tr>
<tr>
<td>Jul</td>
<td>3</td>
</tr>
<tr>
<td>Aug</td>
<td>7</td>
</tr>
<tr>
<td>Sep</td>
<td>2</td>
</tr>
<tr>
<td>Oct</td>
<td>5</td>
</tr>
</tbody>
</table>

How do we resolve collisions?
- We'll use chaining: each table position is the head of a list
- For any particular problem, this might work terribly
- In practice, using a good hash function, we can assume each position is equally likely

Analysis for Hashing with Chaining

- Analyzed in terms of load factor λ = n/m = (items in table)/(table size)
- Claim U is the same as the average number of items per table position = n/m = λ
- We count the expected number of probes (key comparisons)
- Goal: Determine U = number of probes for an unsuccessful search
- Now we want to determine S = number of probes for a successful search (shown on blackboard)
Table Doubling

We know each operation takes time $O(\lambda)$ where $\lambda = n/m$.

But isn't $\lambda = \Theta(n)$?

What's the deal here? It's still linear time!

Table Doubling:
- Set a bound for $\lambda$ (call it $\lambda_0$).
- Whenever $\lambda$ reaches this bound:
  - Create a new table, twice as big and
  - Re-insert all the data.
- Easy to see operations usually take time $O(1)$.

Analysis of Table Doubling

- Suppose we reach a state with $n$ items in a table of size $m$ and that we have just completed a table doubling.

<table>
<thead>
<tr>
<th>Copying Work</th>
<th>Everything has just been copied</th>
<th>n inserts</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Half were copied previously</td>
<td>n/2 inserts</td>
</tr>
<tr>
<td></td>
<td>Half of those were copied previously</td>
<td>n/4 inserts</td>
</tr>
<tr>
<td>Total work</td>
<td>$n + n/2 + n/4 + \ldots = 2n$</td>
<td></td>
</tr>
</tbody>
</table>

Analysis of Table Doubling, Cont'd

- Total number of insert operations needed to reach current table = copying work + initial insertions of items = $2n + n = 3n$ inserts.
- Each insert takes expected time $O(\lambda_0)$ or $O(1)$, so total expected time to build entire table is $O(n)$.
- Thus, expected time per operation is $O(1)$.

Disadvantages of table doubling:
- Worst-case insertion time of $O(n)$ is definitely achieved (but rarely).
- Thus, not appropriate for time critical operations.

Java Hash Functions

- Most Java classes implement the `hashCode()` method.
- `hashCode()` returns an `int`.
- Java’s `HashMap` class uses $h(X) = X.hashCode() \mod m$.
- $h(X)$ in detail:
  ```java
  int hash = X.hashCode();
  int index = (hash & 0x7FFFFFFF) % m;
  ```

What `hashCode()` returns:
- `Integer`: uses the `int` value.
- `Float`: converts to a bit representation and treats it as an `int`.
- Short `String`s: 37*previous + next value.
- Long `String`s: sample of 8 characters; 39*previous + next value.

Hash Tables in Java

- Use chaining
- Initial (default) size = 101
- Load factor = $\lambda_0 = 0.75$
- Uses table doubling ($2^\ast$previous + 1)

Hashing Application: Spell Checking

- We want to create a “spelling dictionary” containing 10,000 words.
- A spelling query should be fast.
- Should return true iff word is contained in dictionary.
- Basic idea:
  - Use a `Hashtable` consisting only of bits (say 100K bytes or about 800,000 bits).
  - Compute a hash value for each word and turn on the corresponding bit in the table.
  - What’s the probability of a false positive? (It’s too high!)
  - Fix: Use more hash functions.