CSE 326: Data Structures
It’s an open-and-closed hash!

Hannah Tang and Brian Tjaden
Summer Quarter 2002

Reminder: Dictionary ADT

• Dictionary operations
  – insert
  – find
  – delete

• Stores values associated with user-specified keys
  – values may be any (homogenous) type
  – keys may be any (homogenous) comparable type

Implementations So Far

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>find</th>
<th>delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted list</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Sorted list</td>
<td>O(n)</td>
<td>O(log n)?</td>
<td>O(n)</td>
</tr>
<tr>
<td>Trees</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
</tbody>
</table>

How about O(1) insert/find/delete?

Hash Table Goal

We can do:

```
[1] = "Albert"
```

We want to do:

```
[1] = "what’s..."
```

Hash Table Approach

But... is there a problem? is this a pipe-dream?
Hash Table
Dictionary Data Structure

- Hash function: maps keys to integers
  - result: can quickly find the right spot for a given entry

What if we have a sparse unordered table?
Can we efficiently list all entries?

Hash Table Terminology

- hash function
- collision
- keys
- load factor $\lambda = \# \text{ of entries} / \text{table size}$

Hash Table Code
First Pass

```c
int find(Key k) {
    int index = hash(k) % tableSize;
    return Table[index];
}
```

What should the hash function be?
How should we resolve collisions?
What should the table size be?

A Good Hash Function…

- is easy (fast) to compute ($O(1)$ and practically fast).
- distributes the data evenly ($hash(a) \% size \neq hash(b) \% size$).
- uses the whole hash table (for all $0 \leq k < size$, there’s an $i$ such that $hash(i) \% size = k$).

Good Hash Function for Integers

- Choose
  - tableSize is prime
  - hash(n) = n
- Example:
  - tableSize = 7
  - insert(4)
  - insert(17)
  - find(12)
  - insert(9)
  - delete(17)

Good Hash Function for Strings?

- Let $s = s_1 s_2 s_3 \ldots s_n$:
  - $hash(s) = \text{ASCII}(s_1) + \text{ASCII}(s_2) + \ldots + \text{ASCII}(s_n)$

Problems?
Making the String Hash Easy to Compute

- Use Horner’s Rule

```c
int hash(String s) {
    h = 0;
    for (i = s.length() - 1; i >= 0; i--) {
        h = (s[i] + 128*h) % tableSize;
    }
    return h;
}
```

How to Design a Hash Function

- Know what your keys are
- Study how your keys are distributed
- Try to include all important information in a key in the construction of its hash
- Try to make “neighboring” keys hash to very different places
- Prune the features used to create the hash until it runs “fast enough” (very application dependent)

Collisions

- Pigeonhole principle says we can’t avoid all collisions
  - try to hash without collision w keys into n slots with m > n
  - try to put 7 pigeons into 5 holes

- What do we do when two keys hash to the same entry?
  - open hashing: put little dictionaries in each entry
    img: shove extra pigeons in one hole!
  - closed hashing: pick a next entry to try

Open Hashing Code

```c
dictionary findBucket(Key k) {
    return table[hash(k) % table.size];
}

void delete(Key k) {
    findBucket(k).delete(k);
}

Value find(Key k) {
    return findBucket(k).find(k);
}

void insert(Key k, Value v) {
    findBucket(k).insert(k, v);
}
```

Load Factor in Open Hashing

- Search cost
  - unsuccessful search:
  - successful search:

- Desired load factor:
Closed Hashing or Open Addressing

What if we only allow one Key at each entry?
- two objects that hash to the same spot can’t both go there
- first one there gets the spot
- next one must go in another spot

Properties
- \( \lambda \leq 1 \)
- performance degrades with difficulty of finding right spot

Probing
- Probing how to:
  - First probe - given a key \( k \), hash to \( h(k) \)
  - Second probe - if \( h(k) \) is occupied, try \( h(k) + f(1) \)
  - Third probe - if \( h(k) + f(1) \) is occupied, try \( h(k) + f(2) \)
  - And so forth

- Probing properties
  - we force \( f(0) = 0 \)
  - the \( i^{th} \) probe is to \( (h(k) + f(i)) \) mod size
  - if \( i \) reaches size - 1, the probe has failed
  - depending on \( f() \), the probe may fail sooner
  - long sequences of probes are costly!

Load Factor in Linear Probing

- For any \( \lambda \leq 1 \), linear probing will find an empty slot
- Search cost (for large table sizes)
  - successful search: \( \frac{1}{2} \left( 1 + \frac{1}{1 - \lambda} \right) \)
  - unsuccessful search: \( \frac{1}{2} \left( 1 + \frac{1}{1 - \lambda^2} \right) \)

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

Linear Probing Example

<table>
<thead>
<tr>
<th>Insert</th>
<th>Load Factor</th>
<th>Probes</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>0.4265</td>
<td>1</td>
</tr>
<tr>
<td>93</td>
<td>0.5531</td>
<td>1</td>
</tr>
<tr>
<td>40</td>
<td>0.7143</td>
<td>3</td>
</tr>
<tr>
<td>47</td>
<td>0.8571</td>
<td>4</td>
</tr>
<tr>
<td>109</td>
<td>1.4286</td>
<td>6</td>
</tr>
<tr>
<td>55</td>
<td>2.6316</td>
<td>8</td>
</tr>
</tbody>
</table>

Linear Probing

- \( f(i) = i \)
- Probe sequence is
  - \( h(k) \) mod size
  - \( h(k) + 1 \) mod size
  - \( h(k) + 2 \) mod size
  - ...

- findEntry using linear probing:
  ```java
  bool findEntry(Key k, Entry entry) {
    int probePoint = hash(k);
    do {
      if (probePoint < 0) probePoint += size;
      entry = table[probePoint];
    } while (entry.isEmpty() && entry.getKey() != k);
    return !entry.isEmpty();
  }
  ```

Quadratic Probing

- \( f(i) = i^2 \)
- Probe sequence is
  - \( h(k) \) mod size
  - \( (h(k) + 1) \) mod size
  - \( (h(k) + 4) \) mod size
  - \( (h(k) + 9) \) mod size
  - ...

- findEntry using quadratic probing:
  ```java
  bool findEntry(Key k, Entry entry) {
    int probePoint = hash(k), numProbes = 0;
    do {
      entry = table[probePoint];
      probePoint = (probePoint + 2*numProbes - 1) % size;
    } while (entry.isEmpty() && entry.getKey() != k);
    return !entry.isEmpty();
  }
  ```
Quadratic Probing Example

Insert(76)
76%7 = 6

Insert(40)
40%7 = 5

Insert(48)
48%7 = 6

Insert(5)
5%7 = 5

Insert(55)
55%7 = 6

Quadratic Probing Succeeds (for $\lambda \leq \frac{1}{2}$)

- If size is prime and $\lambda \leq \frac{1}{2}$, then quadratic probing will find an empty slot in size/2 probes or fewer.
  - Show for all $0 \leq i, j \leq \text{size}/2$ and $i \neq j$:
    1. $(h(x) + i^2) \mod \text{size} \neq (h(x) + j^2) \mod \text{size}$
    2. By contradiction: suppose that for some $i \neq j$:
       1. $(h(x) + i^2) \mod \text{size} = (h(x) + j^2) \mod \text{size}$
       2. $i^2 \mod \text{size} = j^2 \mod \text{size}$
       3. $(i^2 - j^2) \mod \text{size} = 0$
       4. $[(i + j)(i - j)] \mod \text{size} = 0$
       5. But how can $i + j = 0$ or $i - j = \text{size}$ when $i \neq j$ and $i, j \leq \text{size}/2$?
       6. Same for $i - j \mod \text{size} = 0$

Quadratic Probing May Fail (for $\lambda > \frac{1}{2}$)

- For any $i$ larger than size/2, there is some $j$ smaller than $i$ that adds with $i$ to equal size (or a multiple of size). D’oh!

Load Factor in Quadratic Probing

- For any $\lambda \leq \frac{1}{2}$, quadratic probing will find an empty slot; for greater $\lambda$, quadratic probing may find a slot.
- Quadratic probing does not suffer from primary clustering.
- Quadratic probing possibly suffers from secondary clustering.

Double Hashing

$f(i) = i \cdot \text{hash}_2(x)$

- Probe sequence is
  1. $h_1(k) \mod \text{size}$
  2. $(h_1(k) + h_2(x)) \mod \text{size}$
  3. $(h_1(k) + 2 \cdot h_2(x)) \mod \text{size}$
  4. …

- Code for finding the next linear probe:
  ```java
  bool findEntry(Key k, Entry entry) {
    int probePoint = hash1(k), hashIncr = hash2(k);
    do {
      entry = table[probePoint];
      probePoint = (probePoint + hashIncr) % size;
    } while (!entry.isEmpty() && entry.getKey() != k);
    return !entry.isEmpty();
  }
  ```
A Good Double Hash Function…

…is quick to evaluate.
…differs from the original hash function.
…never evaluates to 0 (mod size).

One good choice is to choose a prime \( R < \text{size} \) and:

\[
\text{hash}_2(x) = R - (x \mod R)
\]

Double Hashing Example

<table>
<thead>
<tr>
<th>Insert</th>
<th>Probes</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>1</td>
</tr>
<tr>
<td>93</td>
<td>1</td>
</tr>
<tr>
<td>40</td>
<td>1</td>
</tr>
<tr>
<td>47</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>55</td>
<td>2</td>
</tr>
</tbody>
</table>

 probes: 1 1 1 2 1 2

Load Factor in Double Hashing

- For any \( \lambda < 1 \), double hashing will find an empty slot (given appropriate table size and \( \text{hash}_2 \)).
- Search cost appears to approach optimal (random hash):
  - successful search: \( \frac{1}{\lambda} \ln \frac{1}{1-\lambda} \)
  - unsuccessful search: \( \frac{1}{1-\lambda} \)
- No primary clustering and no secondary clustering
- One extra hash calculation

Deletion in Closed Hashing

- Must use lazy deletion!
- On insertion, treat a deleted item as an empty slot

The Squished Pigeon Principle

- An insert using closed hashing cannot work with a load factor of 1 or more.
- An insert using closed hashing with quadratic probing may not work with a load factor of \( \frac{3}{2} \) or more.
- Whether you use open or closed hashing, large load factors lead to poor performance!
- How can we relieve the pressure on the pigeons?

Rehashing

- When the load factor gets “too large” (over a constant threshold on \( \lambda \)), rehash all the elements into a new, larger table:
  - takes \( \mathcal{O}(n) \), but amortized \( \mathcal{O}(1) \) as long as we (just about) double table size on the resize
  - spreads keys back out, may drastically improve performance
  - gives us a chance to retune parameterized hash functions
  - avoids failure for closed hashing techniques
  - allows arbitrarily large tables starting from a small table
  - clears out lazily deleted items
It’s all about tradeoffs!