Lecture 27: Mappings

CS 62
Spring 2011
Kim Bruce & Kevin Coogan

Map<K,V>

- Collection of associations between a key and associated value, e.g. name & phone number
  - Though doesn't use Bailey’s Association class
- As usual lots of implementations
- Also called dictionaries after example
  - Look up table!

public interface Map<K,V> {
    public int size();
    public boolean isEmpty();
    public boolean containsKey(Object k);
    public boolean containsValue(Object v);
    public V get(Object k);
    public V put(K k, V v);
    public V remove(Object k);
    public void putAll(Map<K,V> other);
    public void clear();
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();
    public boolean equals(Object other);
    public int hashCode();
}

Map.Entry is essentially Association.

Implementations

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Search</th>
<th>Insert</th>
<th>Delete</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked List</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Sorted Array</td>
<td>O(log n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Balanced BST</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Array[KeyRange]</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>KeyRange</td>
</tr>
</tbody>
</table>

where n is # elts in table, N is # slots in array

Last row is array where keys are subscripts.
Hash Table

• Why is using keys as subscripts bad?
  • Restricts types of keys
  • keys often too sparse
  • Suppose use SS#'s as subscripts to table of students?
• Provide function from keys to subscripts that is denser.

Hash Functions

• Want H: EltType → Subscripts, where
  • H(elt) can be computed quickly
  • if e1 != e2 then H(e1) != H(e2)
    • H is one-to-one
• Called perfect hashing function
  • Hard to find unless know all keys in advance.
• Now adding, finding, removing all O(1)
• So important that hashCode function built-in to Java classes.

Hash Functions

• Look for reasonable function that scatters elements through array randomly so won’t bump into each other.
  • Lose any ordering on keys
• Ideal is to find in time O(1).
• We want to:
  • Find good hashing functions
  • Figure out what to do if 2 elts sent to same locn

• Given hash function must always be tried on real data in order to find out whether it is effective or not.

String-Valued Keys

• Convert from string to digits
  • Can use formula like $Key(xy) = 2^8 \cdot \text{Ord}(x) + \text{Ord}(y)$
    • where $\text{ord}(x) = \text{ascii code (or unicode)}$ for x
    • If use long ints then can get 4 letters into 1 number
    • Java uses for string s:
      $$s[0]*31^{(n-1)} + s[1]*31^{(n-2)} + \ldots + s[0]*31^{(n-1)} + s[1]*31^{(n-2)} + \ldots + s[n-1]$$
  • Simple alternative, add together ord of all letters
    • Problem: words with same letters mapped to same place
      • E.g.: miles, slime, smile
• Similar w/other structured types
  • Combine hash of pieces, but not depend on structure
Cutting Down

- If hash code too large for table:
  - Choose digits from certain positions of key.
    - E.g., last 4 digits of SS#
  - Let \( H'(key) = H(key) \mod \text{TableSize} \)
    - generally best if TableSize is prime.
  - Square the key and then select certain bits.
    - Usually the middle half of the bits is taken.
    - Multiplication ensures all digits used in computation
  - Folding:
    - Break key into pieces and add them up

Well-defined Hash Functions

- Require that if \( K1.equals(K2) \) returns true then \( H(K1) = H(K2) \)
  - Consider fractions 2/3, 4/6 represented in Fraction class w/instance vbles num, denom.
  - If \( H(2/3) \neq H(4/6) \) then put into table in different places -- might not find if one in table and look up other.
  - Hence, if redefine equals then must redefine hashCode so \( x.equals(y) \Rightarrow x.hashCode() == y.hashCode() \)

Important!!

- How important?
  - Eclipse include automated way of generating equals and hashCode methods under “Source” menu.

What if get Hash Clashes?

- Home address of key \( K \) is \( H(K) \).
- Suppose have two keys \( K1 \neq K2 \),
  - but \( H(K1) = H(K2) \), i.e., have same home address
- What happens when insert both into hash table?
  - Note original key and value must both be stored!!
- Two ways out:
  1. Rehash as needed to find an empty slot (open addressing)
  2. External chaining
Open Addressing

- Find home address
- If filled, keep trying until find open slot
  - Three types of entries in table:
    - empty, represented by null
    - deleted, marked by inserting special reserved object
    - normal reference to an object.

Variants of Open Addressing: Linear Probing

- Linear rehashing:
  - Look at successive slots until find open one.
  - \( \text{Probe}(i) = (i+1) \mod \text{TableSize} \)
  - Using \( k \neq 1 \) doesn't seem to help
- Problems with clustering

Primary Clustering

- Primary clustering occurs when more than one key has the same home address.
  - If the same rehashing scheme is used for all keys with same home address then new key will collide with all earlier keys with the same home address when the rehashing scheme is employed.
  - In example happened with A, A2, A1, A3, A4.

Secondary Clustering

- Secondary clustering occurs when a key has a home address which is occupied by an element which originally got hashed to a different home address,
  - in rehashing got moved to the address which is the same as the home address of the new element.
- In example, happened with D, E, ZA
Deleting

- What happens when delete A2 and then search for A1?
- Must mark deletions as “reserved” & try to fill when possible
  - See “locate” code in class Hashtable<K,V>
  - See containsKey, get, put, remove

A  A1  A2  A3  D  A4  ZA  GA  G  E  ...  ...
   A  d  A1  A3  D  A4  ZA  GA  G  E  ...  ...

Quadratic Probing

- Use \((\text{home} + j^2) \mod \text{TableSize}\) on jth rehash
  - Helps with secondary clustering, but not primary
  - Can result in case where don’t try all slots
    - E.g., TableSize = 5, and start with H = 1. Rehashings give 2, 0, 0, 2, 1, 2, 0, 0, ...
    - The slots 3 and 4 will never be examined to see if they have room.

Double Hashing

- Use second hash function on key to determine delta for next try.
  - E.g., \(\text{delta(Key)} = (\text{Key} \mod (\text{TableSize} - 2)) + 1\)
  - Should help with primary and secondary clustering.
  - Ex: Spose \(H(n) = n \mod 5\). Then \(H(1) = H(6) = H(11)\).
    - However, delta(1) = 2, delta(6) = 1, and delta(11) = 3.

External Chaining

- Each slot in table holds unlimited # elts
  - Each slot is list -- implemented as desire
  - For good performance, list should be short
    - so no need for balanced binary search tree -- waste of time
- Advantages
  - Deleting simple
  - # elts in table can be > # slots
  - Avoids problems of secondary clustering
  - Primary clustering?
Analysis

• Behavior of the hash clash strategies depends on the load factor of the table.
• Load factor $\alpha = \# \text{elts in table}/\text{size of table}$
  • ranges between 0 and 1 with open addressing
  • can be $> 1$ with external chaining.
• Higher the load factor, the more likely your are to have clashes.

Performance

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Unsuccessful</th>
<th>Successful</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear rehashing</td>
<td>$\frac{1}{2} \left(1 + \frac{1}{1-\alpha}\right)^2$</td>
<td>$\frac{1}{2} \left(1 + \frac{1}{1-\alpha}\right)$</td>
</tr>
<tr>
<td>Double hashing</td>
<td>$\sqrt{\frac{1}{1-\alpha}}$</td>
<td>$-\frac{1}{\alpha} \log(1-\alpha)$</td>
</tr>
<tr>
<td>External hashing</td>
<td>$\alpha + e^{-\alpha}$</td>
<td>$1 + \frac{1}{2} \alpha$</td>
</tr>
</tbody>
</table>

Entries represent number of compares needed to find elt or demonstrate not there.

Performance for $\alpha = .9$

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Unsuccessful</th>
<th>Successful</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear rehashing</td>
<td>55</td>
<td>5.5</td>
</tr>
<tr>
<td>Double hashing</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>External hashing</td>
<td>3</td>
<td>1.45</td>
</tr>
</tbody>
</table>

Figure 15.12 The shape of the theoretical performance curves for various hashing techniques. (These graphs demonstrate theoretical predictions and not experimental results which are, of course, dependant on particular data and hashing functions.) Our hash table implementation uses linear probing.