CSCI 136
Data Structures &
Advanced Programming

Fall 2018
Lecture 34
The 2070567s
Reminders

• No lab today!

• Final exam
  • Monday, December 17 at 9:30 in TPL 203
  • Covers everything, with strong emphasis on post-midterm
  • Study guide, sample exam will be posted on handouts page

• Friday: Wrapping Up and course evaluations
Last Time

- Finish Dijkstra’s algorithm
- Maps
  - Revisit Naïve implementation from Lab 2
  - structure5.Hashtable (finally)
    - Hash functions
Today’s Outline

• structure5.Hashtable (finally)
  • Hash functions
  • “Load factor”
  • Collisions and how to handle them
• You should also read Ch 15 for more info
Final Topic: Maps and Hashing
Map Interface

Methods for Map<K, V>

- int size() - returns number of entries in map
- boolean isEmpty() - true iff there are no entries
- boolean containsKey(K key) - true iff key exists in map
- boolean containsValue(V val) - true iff val exists at least once in map
- V get(K key) - get value associated with key
- V put(K key, V val) - insert mapping from key to val, returns value replaced (old value) or null
- V remove(K key) - remove mapping from key to val
- void clear() - remove all entries from map
Hashing in a Nutshell

- Assign objects to “bins” based on key
- When searching for object, go directly to appropriate bin (and ignore the rest)
- If there are multiple objects in bin, then search for the correct one
- Important Insight: Hashing works best when objects are evenly distributed among bins
Implementing a HashTable

• How can we represent bins?
• Slots in array (or Vector, but arrays are are faster)
  • Initial size of array is a prime number
• How do we find a key’s bin number?
  • We use a hash function that converts keys into integers
  • In Java, all Objects have public int hashCode()
Implementing HashTable

• How do we add Associations to the array?
  • `array[o.hashCode() % array.length] = o;`?

  • Warning
    • "computer".hashCode() = -599163109 (sad!)

• Collisions make life hard

• Two approaches
  • Open Addressing
    • Linear or Quadratic Probing; Double Hashing
  • External chaining
Inserting

- If a collision occurs at a given bin, just scan forward (linearly) until an empty slot is available
- We will call a contiguous region of full bins a run
- If you are looking for a (K,V)-pair, scan linearly through the run until you find it or reach the end of the run
- Let’s implement put(key, val) and get(key)…
public V put (K key, V value) {
    int bin = key.hashCode() % data.length;
    while (true) {
        Association<K,V> slot = (Association<K,V>) data[bin];
        if (slot == null) {
            data[bin] = new Association<K,V>(key, value);
            return null;
        }
        if (slot.getKey().equals(key)) { // already exists!
            V old = slot.getValue();
            slot.setValue(value);
            return old;
        }
        bin = (bin + 1) % data.length;
    }
}
public V get (K key) {
    int bin = key.hashCode() % data.length;
    while (true) {
        Association<K,V> slot = (Association<K,V>) data[bin];
        if (slot == null)
            return null;
        if (slot.getKey().equals(key))
            return slot.getValue();
        bin = (bin + 1) % data.length;
    }
}
Linear Probing

- NaiveProbing.java
  - We specify a dummy hash function: index of first letter of word
  - Initial array size = 8
  - Add “air hockey” to hash table
  - Add “doubles ping pong”
  - Add “quidditch”

- What happens when we remove “air hockey”, and then lookup “quidditch”?
  - Our run was broken up!
  - We need a “placeholder” for removed values to preserve runs…

- See Hashtable.java in structure5
  - Uses array of HashAssociation<K,V> with “reserved” flag
Open Addressing Variants

- Using linear probing with a hash function $h(k)$
  - The $i^{\text{th}}$ probe uses address $p(k,i) = (h(k) + i) \mod n$
  - One could use $p(k,i) = (h(k) + c \times i) \mod n$
    - For some $c$ relatively prime to $n$

- Even better: Quadratic Probing
  - Here $p(k,i) = (h(k) + c_1 i + c_2 i^2) \mod n$
    - Where $c_1$ and $c_2$ are relatively prime to $n$

- Even even better: Double Hashing
  - Here we have primary and secondary hash functions $h_1(k)$ and $h_2(k)$ and
  - $p(k,i) = (h_1(k) + i h_2(k)) \mod n$
    - Now $h_2(k)$ should be relatively prime to $n$!
Open Addressing Drawbacks

• Downsides of open addressing?
  • What if array is almost full?
    • Looooong runs for every lookup…
    • Items out of place if we don’t re-index after removing (placeholders create artificially long runs)

• How can we avoid these problems?
  • Keep all values that hash to same bin in a Structure
    • Usually a SLL
  • *External chaining* “chains” objects with the same hash value together
External Chaining

• Instead of runs, we store a list in each bin

\[
data[i \ldots j] = [(K_1, V_1), (K_2, V_2), \ldots, (K_n, V_n)]
\]

• get(), put(), and remove() only need to check one slot’s list

• No placeholders!
Probing vs. Chaining

What is the performance of:

- \text{put}(K, V)
  - LP: $O(1 + \text{run length})$
  - EC: $O(1 + \text{chain length})$

- \text{get}(K)
  - LP: $O(1 + \text{run length})$
  - EC: $O(1 + \text{chain length})$

- \text{remove}(K)
  - LP: $O(1 + \text{run length})$
  - EC: $O(1 + \text{chain length})$

- How do we control cluster/chain length?
Load Factor

• Need to keep track of how full the table is
  • Why?
  • What happens when array fills completely?
• Load factor is a measure of how full the hash table is
  • LF = (number of elements) / (table size)
• When LF reaches some threshold, double size of array (typical threshold: 0.6)
  • Challenges?
Doubling Array

- Cannot just copy values
  - Why?
  - Hash values may change
  - Example: suppose \( \text{key} \cdot \text{hashCode}() == 11 \)
    - \( 11 \% 8 = 3; \)
    - \( 11 \% 16 = 11; \)
- Result: must recompute all hash codes, reinsert into new array
Good Hashing Functions

• Important point:
  • All of this hinges on using “good” hash functions that spread keys “evenly”

• Good hash functions
  • Fast to compute
  • Uniformly distribute keys

• Almost always have to test “goodness” empirically
Example Hash Functions

- What are some feasible hash functions for Strings?
  - First char ASCII value mapping
    - 0-255 only
    - Not uniform (some letters more popular than others)
  - Sum of ASCII characters
    - Not uniform - lots of small words
    - smile, limes, miles, slime are all the same
Example Hash Functions

• String hash functions
  • Weighted sum
    • Small words get bigger codes
    • Distributes keys better than non-weighted sum
  • Let’s look at different weights…
$$\sum_{i=0}^{s.length()} s.charAt(i)$$

Hash of all words in UNIX spelling dictionary (997 buckets)
\[ \sum_{i=0}^{n} s\text{.charAt}(i) \times 2^i \]
\[ \sum_{i=0}^{n} \text{s.charAt}(i) \times 256^i \]

This looks pretty good, but $256^i$ is big…
\[ \sum_{i=0}^{n} s.\text{charAt}(i) \times 31^i \]

Java uses:
\[ \sum_{i=0}^{n} s.\text{charAt}(i) \times 31^{(n-i-1)} \]
Hashtables: $O(1)$ operations?

• How long does it take to compute a String’s hashCode?
  • $O(s.length())$

• Given an object’s hash code, how long does it take to find that object?
  • $O(\text{run length})$ or $O(\text{chain length})$ PLUS cost of .equals() method

• Conclusion: for a good hash function (fast, uniformly distributed) and small load factor, we say operations take $O(1)$ time
  • But that’s not strictly true….
## Summary

<table>
<thead>
<tr>
<th></th>
<th>put</th>
<th>get</th>
<th>space</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsorted vector</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>unsorted list</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>sorted vector</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>balanced BST</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>array indexed by key</td>
<td>O(1)*</td>
<td>O(1)*</td>
<td>O(key range)</td>
</tr>
</tbody>
</table>

*PolitiFact Rating: not quite Pants on Fire
For external chaining

• Assuming the hashing function is equally likely to hash to any slot

Theorem: A search will take $O(1 + m/n)$ time, on average

• $n$ is table size, $m$ is number of keys stored

• True for both successful and unsuccessful searches
  • Based on expected chain length
What Can We Say For Sure?!

For open addressing

• Assuming that all probe sequences are equally likely [which is unlikely!]

• Assuming load factor $0 < \alpha < 1$

Theorem: An unsuccessful search will perform, on average, $O(1 + \alpha)$ probes

Theorem: A successful search will perform, on average, $O\left(\frac{1}{\alpha} \log \frac{1}{1-\alpha}\right)$ probes

More probe sequences $\Rightarrow$ better average case
Perfect Hashing

In certain cases, it is possible to design a hashing scheme such that

• Computing the hash takes $O(1)$ time
• There are no collisions
  • Different keys always have different hash values

This is called a *perfect hashing scheme*
Perfect Hashing

If keyspace is smaller than array size

• Handcraft the hashing function
  • Ex: Reserved words in programming languages
• Make array really big
  • Ex: All ASCII strings of length at most 4
  • Hash is 32 bit number
  • Array of size 4.3 billion will suffice