Lecture 9: Hash tables

CS 2112 Spring 2012
Andrew Myers

Collection implementations

<table>
<thead>
<tr>
<th></th>
<th>Unsorted sets &amp; maps</th>
<th>Resizable arrays (array)</th>
<th>Sorted sets (search tree)</th>
<th>Stacks (array)</th>
<th>Priority queues (tree, heap)</th>
</tr>
</thead>
<tbody>
<tr>
<td>add, push</td>
<td>O(1)</td>
<td>O(lg n)</td>
<td>O(1)</td>
<td>O(lg n)</td>
<td></td>
</tr>
<tr>
<td>get, contains, put</td>
<td>O(1)</td>
<td>O(lg n)</td>
<td></td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td>remove, pop</td>
<td>O(1)</td>
<td>O(lg n)</td>
<td>O(1)</td>
<td>O(lg n)</td>
<td></td>
</tr>
</tbody>
</table>

Can we get the O(1) performance of arrays on general keys?

Direct Address Table

- Want a map from keys to values
- Suppose we can convert keys to different small integers
  - Example: Addresses on my street
    - Start at 1, go to 88
    - A few lots don’t have houses
- Make an array as large as the set of keys
- To find an entry, we just index to that entry of the array
  - Use null or special value to indicate absence
- Lookup operations take O(1) time!

Problem

- Want O(1) operations but with general keys
  - E.g., look up employee records by social security #
- Direct address table?
- Problem: too many SS numbers
  - Will have 10,000,000,000 mostly empty entries…
Hash functions

- Idea: define a (cheap) function to map keys onto a small range of array indices (“buckets”)
- Given an array of size 12:

  452-3425-112 (Social security number)

  \[ h(x) \]

  0, 1, 2, …, 11

Collisions

- Problem: hash function may create collisions between two different keys

  452-3425-112
  \[ h(x) \]
  7

  563-2332-917
  \[ h(x) \]
  7

1. Cheap but avoids collisions: a function that looks as random as possible
2. Need a way to deal with collisions when they (inevitably) happen

Examples of Hash Functions

\( \text{int} \rightarrow \{0, 1, \ldots, 99\} \)

- Bad: use only part of the key
  - constant functions: \( h(x) = 7 \)
  - two most significant digits: \( h(379988) = 37 \)
  - two least significant digits: \( h(379988) = 88 \)
- Better: Use all the information in the key
  - sum of digit pairs mod 100: \( h(379988) = 37 + 99 + 88 \mod 100 = 24 \)
  - square number and take middle digits
- Best: Every change to the argument key produces an unpredictable, apparently random change to result
  - MD5 hash function, CRC (cyclic redundancy check) on key data

Collision resolution #1

- Chained buckets: array elements are linked lists
- Walk down linked list till you find
- Expected length of linked list is proportional to load factor
  - Load factor = \# elements / \# buckets
  - Good load factor ~ 1-2 for chained buckets
Implementing maps

- Map is just a set of key/value pairs
  - A String→int map with chained buckets:

Collision resolution #2: open addressing

- Just use an array of elements.
- If you find the wrong element, search elsewhere in array.
- Simple: walk forward till you find it.

Performance

- Affected by many factors:
  - Size of array relative to number of data items
    - Consider limit where there is only 1 bucket
      - as bad as simple linked lists!
  - Quality of hash function
    - Good hash functions do not lead to clustering of data → low collision rate
**Analysis for Hashing with Chaining**

- Analyzed in terms of *load factor* $\lambda = n/m = \text{(items in table)}/\text{(table size)}$
- Claim $U$ is the same as the average number of items per table position $= n/m = \lambda$
- We count the expected number of *probes* (key comparisons)
- Claim $S$ = number of probes for a *successful* search $= 1 + \lambda/2$
- Goal: Determine $U$ = number of probes for an *unsuccessful* search

**The hashCode method**

- Want to store arbitrary objects, not just integers
- All Java objects have `hashCode()` method for use by hash tables
  ```java
  int hashCode();
  ```
  - By default: memory address of object
- HashCode needs to capture important information
- Hash table can handle information diffusion (randomness)

**Pitfalls**

- Easy to define a hash function that doesn’t seem very random
- E.g., pick the first character of string keys
  - What if all strings have the same first char?
- E.g., use the memory address
  - All addresses = 0 mod 4 or mod 8.
  - Hash table effectively four times as small if modular hashing used with power of two size
  - The Java default...

**Some reasonably good hash functions**

- Modular hashing: $h(k) = k \mod m$ for some $m=\#\text{buckets}$
  - But: avoid $m$ = power of 2. Prime $m$ is good
- Multiplicative hashing: $h(k) = (ka/2^q) \mod m$ for appropriately chosen values of $a$, $m$, and $q$.
  - Similar to random number generator
  - Multiplier $a$ should be large and “random”
  - $q$ is crucial and typically forgotten
  - Cheaper than modular hashing, works fine with power-of-2 bucket array
Universal Hashing

- Idea: choose randomly from a large collection of hash functions
- Parameterized family of numeric functions
  - e.g., \( f_{abc}(x) = ax^2 + bx + c \mod 100 \)
- Pick \( a,b,c \) at random!
- Works as well or better than hand-crafted hash functions in most cases!
- Disadvantage: no persistence

Testing a Hash Function

- If bucket \( i \) contains \( x_i \) elements, then the clustering is \( (\sum x_i^2)/n) - n/m \).
- Clustering < 1: hashing is better than random
- Clustering > 1: worse than random
- Clustering = \( k \): roughly \( k \) times slower than random
  - E.g., randomly picking every other bucket gives clustering of 2.

Observations

- Hashing is popular in practice because code is easy to write and maintain and performance is typically excellent
- Performance depends on two key factors:
  - load factor \( \lambda \) = number of entries/size of array
  - choice of hash function
  - if \( \lambda \) appropriately small and hash function is chosen well, get expected \( O(1) \) complexity for all operations
- Chained hashing is faster, less fragile -- used in Java Collections
  - \texttt{java.util.HashMap} implements \texttt{java.util.Map}
  - \texttt{java.util.HashSet} implements \texttt{java.util.Set}

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 we
  - Create a new table, twice as big and
  - Re-insert all the data
- Easy to see operations usually take time \( O(1) \)
  - But sometimes we copy the whole table
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></th>
<th>Copying Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Everything has just been copied</td>
<td>$n$ inserts</td>
</tr>
<tr>
<td>Half were copied previously</td>
<td>$n/2$ inserts</td>
</tr>
<tr>
<td>Half of those were copied previously</td>
<td>$n/4$ inserts</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Total work</td>
<td>$n + n/2 + n/4 + ... = 2n$</td>
</tr>
</tbody>
</table>