Hash Tables

© Rick Mercer
Hash Tables
A "faster" implementation of a Map

Outline

- Discuss what a hash method does
  - translates a string key into an integer

- Discuss a few strategies for implementing a hash table
  - linear probing
  - quadratic probing
  - separate chaining hashing
# Big Oh Complexity for various Map Implementations

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>put</th>
<th>get</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Array</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorted Array</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unsorted Linked List</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorted Linked List</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary Search Tree</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hash Tables

* Hash table: another structure for storing data
  - Provides virtually direct access to objects based on a key (a unique String or Integer)
    * key could be your SID, your telephone number, social security number, account number, …
    * keys must be unique
    * Each key is associated with (mapped to) an object
Hashing

- Must convert keys such as "555-1234" into an integer index from 0 to some reasonable size
- Elements can be found, inserted, and removed using the integer index as an array index
- Insert (called put), find (get), and remove must use the same "address calculator" — which we call the Hash function
- Good structure for implementing a dictionary
Hashing

- Can make a String or Integer object into a key by "hashing" the key to get an int
- Ideally, every key has a unique hash value
  - Then the hash value could be used as an array index, however,
    - you cannot rely on every key "hashing" to a unique integer
    - but can usually get close enough
    - Still need a way to handle "collisions" "abc" may hash to the same int as "cba"
      - if a lousy hash function is used
Hash Tables: Runtime Efficient

- Lookup time does not grow when n increases
- A hash table supports
  - fast retrieval \(O(1)\)
  - fast deletion \(O(1)\)
  - fast insertion \(O(1)\)
- Could use String keys each ASCII character equals some unique integer
  - "able" = 97 + 98 + 108 + 101 == 404
**Hash method** works something like this:

Convert a String key into an integer that will be in the range of 0 through the maximum capacity - 1.

*Assume the array capacity is 9997*

- **Domain:** "AAAAAAAAAA" .. "zzzzzzzzzz"
- **Range:** 0 .. 9996

- hash(key) = 8482
- hash(key) = 1273
Hash method

- What if the ASCII value of individual chars of the string key added up to a number from ("A") 65 to possibly 488 ("zzzz") 4 chars max

- If the array has size = 309, mod the sum
  "abba": 390 % TABLE_SIZE = 81
  "abcd": 394 % TABLE_SIZE = 85
  "able": 404 % TABLE_SIZE = 95

- These array indices store these keys

  81 -----> abba
  85 -----> abcd
  95 -----> able
A terrible hash method

@<Test
public void testHash() {
    assertEquals(81, hash("abba");
    assertEquals(81, hash("baab");
    assertEquals(85, hash("abcd");
    assertEquals(86, hash("abce");
    assertEquals(308, hash("IKLT");
    assertEquals(308, hash("KLMP");
}

private final int TABLE_SIZE = 309;

public int hash(String key) {
    // return an int in the range of 0..TABLE_SIZE-1
    int result = 0;
    int n = key.length();
    for (int j = 0; j < n; j++)
        result += key.charAt(j); // add up the characters
    return result % TABLE_SIZE;
}
Collisions

- A good hash method
  - executes quickly
  - distributes keys equitably
- But you still have to handle collisions when two keys have the same hash value
  - the hash method is not guaranteed to return a unique integer for each key
    - example: simple hash method with "baab" and "abba"
- There are several ways to handle collisions
  - let us first examine linear probing
Linear Probing
Dealing with Collisions

- **Collision**: When an element to be inserted hashes out to be stored in an array position that is already occupied.

- **Linear probing**: search sequentially for an unoccupied position. *use wraparound*
A hash table after three insertions

using the too simple hash code method

insert objects with these three keys:

"abba"
"abcd"
"abce"

<table>
<thead>
<tr>
<th>0</th>
<th>...</th>
<th>80</th>
<th>81</th>
<th>82</th>
<th>83</th>
<th>84</th>
<th>85</th>
<th>86</th>
<th>308</th>
</tr>
</thead>
</table>

Keys
Collision occurs while inserting "baab"

can't insert "baab" where it hashes to same slot as "abba"

Linear probe forward by 1, inserting it at the next available slot

Try [81]
Put in [82]
Insert "KLMP" "IKLT"
both of which have a hash value of 308

Wrap around when collision occurs at end

```
 0   "IKLT"
...
80
81   "abba"
82   "baab"
83
84
85   "abcd"
86   "abce"
...
308   "KLMP"
```
"baab" still hashes to 81, but since [81] does not hold it, linear probe to [82]. At this point, you could return a reference to it or remove it.
Find and Remove an element

- Follow the same path to find an item
  - If linear search finds an empty hash table slot, the item could not have been found \emph{the search is done}

- To remove an element, follow the same path
  - If found, mark the element deleted somehow

- Three possible states when looking at slots
  - The slot was never occupied (can use in put or may denote the search is over)
  - The slot is occupied \emph{if matches stop} -- or \emph{proceed to next}
  - The slot was occupied, but nothing there now \emph{removed}
  - We could call this a \emph{tombStoned} slot
Linear Probe Implementation

- Could have a linear probing, array based, implementation
  - Each array element references a HashNode with some boolean instance variables to indicate which of the three states it is in active, avail, or TombStoned -- to allow linear probes past removed elements

private class HashTableNode {
  private String key;
  Object data;
  private boolean active, tombstone;

  public HashTableNode() { // All nodes in array initialized this way
    key = null;
    data = null;
    active = false;
    tombstone = false;
  }
}
Array based implementation has Clustering Problem

- Used slots tend to cluster with linear probing

Black areas represent slots in use; white areas are empty slots
Quadratic Probing

- Quadratic probing eliminates the primary clustering problem
- Assume hVal is the value of the hash function
  - Instead of linear probing which searches for an open slot in a linear fashion like this
    
    \[ hVal + 1, \ hVal + 2, \ hVal + 3, \ hVal + 4, \ ... \]
  
  - Add index values in increments of
    
    \[ hVal + 1^2, \ hVal + 2^2, \ hVal + 3^2, \ hVal + 4^2, \ ... \]
Does it work?

- Quadratic probing works if
  - the table size is prime
    - studies show the prime numbered table size removes some of the non-randomness of hash functions
  - and the table is never more than half full
    - probes 1, 4, 9, 16, 32, 64, 128, ... slots away
  - So make your table twice as big as you need
    - insert, find, remove are O(1)
Separate Chaining Hashing

- Separate Chaining Hashing is an alternative to probing
- Maintain an array of linked lists
- Hash to the same place always and insert at the beginning (or end) of the linked list.
  - The linked list needs add and remove methods
An Array of LinkedList Objects Implementation

- An array of linked lists