CS 310: Hashing (Part II)

Connor Baker

February 28, 2019

Review: Hash Table Basics

- Store objects in an array
- Use the length of the hash table in the hashing function to get a valid index

$$x_{\text{index}} = |xhc| \bmod \texttt{hta.length}$$

- `.add(T t)`: put the object `t` in the hash table
  - `hta[xindex] = x;`
- `.has(T t)`: check if `t` is in the hash table
  - `return x.equals(hta[xindex]);`
- `.remove(T t)`: delete `t` from the hash table
  - `if has(t) { hta[xindex] = null; }`
- **What's the Big-O?**

Hash Table Collision

- Motivation
  - Put `t` in the table at `hta[xindex]`
  - Problem: What if `hta[xindex]` is occupied?
  - Answer: find some other place to store `t`
- Common approaches
  - Separate chaining
  - Open addressing

Separate Chaining

- If something's already there:
  - Expand that single entry to an internal data structure
    - One which can ideally accommodate multiple objects of the same hash code
    - It should be able to grow if there are additional objects that need to be stored there

Collision Resolution

- Separate chaining: expand the single array entry into a linked list
  - Compute the integer hash code
  - “Bound” to make it a good index (just mod by the length of the table)
  - Find the list to operate on: `list = hta[xindex]`
- `.add(T t)`: put the object `t` in the hash table
  - `list.add(t);`
- `.has(T t)`: check if `t` is in the list
  - `return list.contains(t);`
- `.remove(T t)`: delete `t` from the list
  - `list.remove(t);`

Separate Chaining Analysis

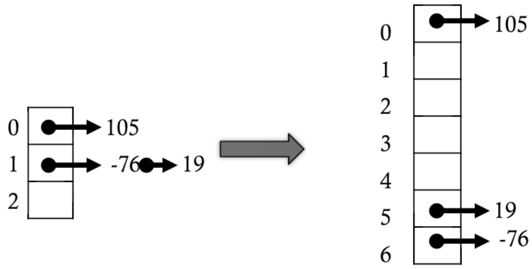
- `.add(T t)` is $O(1)$ assuming adding to the list is $O(1)$ (but that’s only if duplicates are allowed)
- `.remove(T t)` and `.contains(T t)`
  - Run time depends on the number of things in each list
  - They (potentially) look through every element in the longest chain to search for `t`
    - The average case is equivalent to the number of items divided by the length of the table, which yields $O(\text{load})$
  - Worst case?
    - All the elements map to the same index: $O(\text{number of items})$
    - **How do we avoid the worst case?**

Rehashing

- The load is the number of items divided by the length of the hash table
  - A high load means a long average chain length, and a high average chain length means longer runtimes for `.add(T t)`, `.remove(T t)`, and `.has(T t)`
- Rehashing when the load is to high helps us with our average access time
  - Allows us to use a bigger array, a new hash function, and get a lower load
- Basic idea:
  - Allocate a new, larger array (the size should still be prime)
  - Copy over all the items to the new array: this *does* involve re-calculating hash-values for everything and inserting them into the new hash-table

Rehashing Example

- Invoking the following:
  - `.add(105) // 105 % 3 == 0`
  - `.add(19) // 19 % 3 == 1`
  - `.add(-76) // 76 % 3 == 1`
- Rehash when our load > 0.75
- Increase the size to the next prime that is more than double the current size
- Copy the items over, recalculating the hash using the new size of the array
- Load goes from $1 \mapsto 3/7$



Hash Table Overview

- *Separate Chaining*: expand the single array entry into a linked list
  - Compute the integer hash code from the element
  - Bound the values to allow us to easily calculate the index
  - Find the list to operate on
- `.add(T t)`: put `t` in the hash-table
  - Add the item with `list.add(t);`
  - Rehash the entire table if the load is too high
- `.has(T t)`: check if the element is already in the list
  - `return list.contains(t);`
- `.remove(T t)`: delete the element from the list
  - `list.remove(t);`

Big-O Analysis

	`.add()`*	`.has()`	`.remove()`	iteration
Best	$O(1)$	$O(1)$	$O(1)$	$O(n+m)$
Average	$O(n/m)$	$O(n/m)$	$O(n/m)$	$O(n+m)$
Worst	$O(n)$	$O(n)$	$O(n)$	$O(n+m)$

- Hash table with separate chaining (*assuming no duplicates are allowed, and not considering rehashing overhead)
  - The load is n/m
  - n is the number of values in the hash-table
  - m is the number of entries (array capacity)

Separate Chaining is Viable in Practice

- Simple to implement
  - Weiss Figure 20.20
- Reasonably efficient
- A "chain" can be implemented as a different data structure
  - We can use trees, an `ArrayList`, or others
  - Binary search trees can be used if there are no duplicates allowed
- Java's built-in hash tables use separate chaining
  - `java.util.HashSet`, `java.util.HashMap`, and `java.util.Hashtable` all use separate chaining
  - `java.util.HashMap` uses red-black trees when the number of values in one chain is more than eight

Hash Code Review

- We define a hash function to map any object to a manageable integer as its hash code
  - `.hashCode()` defined in `java.lang.Object`
- Hash contract: objects that are identical must have the same hash code
- We need hash functions that are easy to compute and distribute well
- Java provides implementations for built-in types

Hash Code in Java

- Built in for boxed and container types (`Integer`, `Integer[]`, etc.)
- The output is a 32-bit integer
- Straightforward for types with a size of no more than 32 bits
  - If that's true, it's trivial to map every unique value to a different integer
    - This works for our `Integer`, `Boolean`, and `Character` types
  - Types with a larger size use the following trick
    - Exclusive OR the two halves to reduce the reference from 8 bytes to 4 bytes
      - `(int) (this.longValue() ^ (this.longValue() >>> 32))` (that's the logical shift right)

Aggregate Type: String

- The hash code for a string object is computed by taking the β-expansion where the most-significant-place is the ASCII value of the first index of the string (and the base is 31)
  - It follows then that the hash value of the empty string is zero
- Discussion
  - Is 31 special?
    - "According to Joshua Bloch's Effective Java (a book that can't be recommended enough, and which I bought thanks to continual mentions on stackoverflow):

        > The value 31 was chosen because it is an odd prime. If it were even and the multiplication overflowed, information would be lost, as multiplication by 2 is equivalent to shifting. The advantage of using a prime is less clear, but it is traditional. A nice property of 31 is that the multiplication can be replaced by a shift and a subtraction for better performance: `31 * i == (i << 5) - i`.
        > Modern VMs do this sort of optimization automatically.

        (from Chapter 3, Item 9: Always override hashcode when you override equals, page 48)"
        (Taken from `https://stackoverflow.com/a/299748`)

    - Getting more into the math, using a prime number for the modulo is important because (I think – my abstract algebra is weak to non-existent) the Integers modulo a prime number are a field, whereas with other non-prime numbers they are only a ring.
      This is also typically why we use addition and multiplication – the integers under modulo are closed under these operations.
      An added benefit of using a prime number for both hash function and the modulo is that they are *highly* unlikely to share factors in common. (Common factors in the hash function and the modulo result in more collisions.)
    - "You can read Bloch's original reasoning under "Comments" in `http://bugs.java.com/bugdatabase/view_bug.do?bug_id=4045622`. He investigated the performance of different hash functions in regards to the resulting "average chain size" in a hash table. `P(31)` was one of the common functions during that time which he found in K&R's book (but even Kernighan and Ritchie couldn't remember where it came from). In the end he basically had to choose one and so he took `P(31)` since it seemed to perform well enough. Even though `P(33)` was not really worse and multiplication by 33 is equally fast to calculate (just a shift by 5 and an addition), he opted for 31 since 33 is not a prime:

      > Of the remaining four, I'd probably select `P(31)`, as it's the cheapest to calculate on a RISC machine (because 31 is the difference of two powers of two). `P(33)` is similarly cheap to calculate, but it's performance is marginally worse, and 33 is composite, which makes me a bit nervous.

      So the reasoning was not as rational as many of the answers here seem to imply. But we're all good in coming up with rational reasons after gut decisions (and even Bloch might be prone to that)."
      (Taken from `https://stackoverflow.com/a/35304979`)

  - **Can you write code to compute the hash code faster instead of following the formula strictly?**
    - I mean, probably. I like the trick with exclusive or for hashing references (which are typically 64 bits on most modern machines).

Polynomial Hash Code

- String uses a polynomial hash code

  $$a_0\beta^{n-1} + \cdots + a_{n-1}\beta^0$$

  - Java uses $\beta = 31$ for strings
- Optimize
  - We can regroup a polynomial of any degree and reduce the power calculation to multiplication
  - Example of regrouping a degree 3 polynomial so that we don't have to use exponentiation:

    $$a_0\beta^3 + a_1\beta^2 + a_2\beta + a_3$$

    becomes

    $$(((a_0\beta)\beta + a_1)\beta + a_2)\beta + a_3$$

Aggregate Type Hashing

- Other aggregate types
  - Use the String approach: create a polynomial has code using each of the elements, computing each element's hash individually
- The poor man's strategy: `x.toString().hashCode()`

Object Hash Code Example

```
// Composite hashCode from all attributes
public class Student {
    private String name;
    private int age;
    private double grade;
    ...
    // Note: Default is memory address, not proper hashCode!!!
    @Override
    public int hashCode( ) {
        int hash = 17; // pick prime constants
        hash = 31 * hash + name.hashCode();
        hash = 31 * hash + ((Integer) age).hashCode();
        hash = 31 * hash + ((Double) grade).hashCode();
        return hash;
    }
}
```