Hash table lecture

Prof Bill - Apr 2018

This lecture is an introduction to hash tables for CSC 210, Spring 2018.

My goals with these notes are:

- Supplement my lecture
- Supplement our textbook, Muganda Ch 19.3-19.4
- Provide links to more hash table help out there on the internets

thanks... yow, bill

Outline

A. Introduction

What is our **motivation** for using (and understanding) hash tables?
Example: your web browser.

In 1994, there were fewer than 3,000 websites online. By 2014, there were more than 1 billion. That represents a 33 million percent increase in 20 years. That's nuts!

- www.theatlantic.com/technology/archive/2015/09/how-many-websites-are-there/408151/

From 3,000 websites to a billion! That's almost a 1M X increase in websites.

- If browser performance was O(N), then finding websites would be 1M times slower now than 20 years ago

- But browsers aren't getting slower over time as N grows, so this operation must be O(1), constant.

Hash tables are an abstract data type (ADT) that provides O(1) access for nearly any kind of data. This… is our motivation.

Key Terms

You need to understand these terms if you're to "get" hash tables.

hash table, hash function, hash code
map, key-value pair (K, V)
collision, collision resolution, linear probing, open addressing, chaining, closed addressing
load factor, resize, rehash
Java: hashCode(), HashMap, HashSet
quadratic probing, double hashing

B. References

These lecture notes are supplemental to other textbooks and websites available to you.
Or vise versa.

These visualizations are fun and **very helpful** in learning how hash tables work:

- ❏ Hash table with chaining: www.cs.usfca.edu/~galles/visualization/OpenHash.html
- ❏ Hash table with probing: www.cs.usfca.edu/~galles/visualization/ClosedHash.html

Our textbook: Starting Out With Java, 3rd Edition by Muganda and Gaddis; a.co/0Lg1ylp

Another textbook I've used, Data Structures and Algorithms in Java, 6th Edition by Goodrich, et al; a.co/1T0dnHP and www.wiley.com/en-us/Data+Structures+and+Algorithms+in+Java%2C+6th+Edition-p-9781118771334

This Princeton textbook is quite good: Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne. Hash tables are covered in Ch 3 Searching, algs4.cs.princeton.edu/home

CMU lecture (very good!), www.cs.cmu.edu/~adamchik/15-121/lectures/Hashing/hashing.html

Wikipedia and this Wikibook is pretty good, en.wikibooks.org/wiki/Data_Structures/Hash_Tables

Java HashMap API: docs.oracle.com/javase/8/docs/api/index.html?java/util/HashMap.html

C. I want more of that great array performance

We love the O(1) performance of arrays!

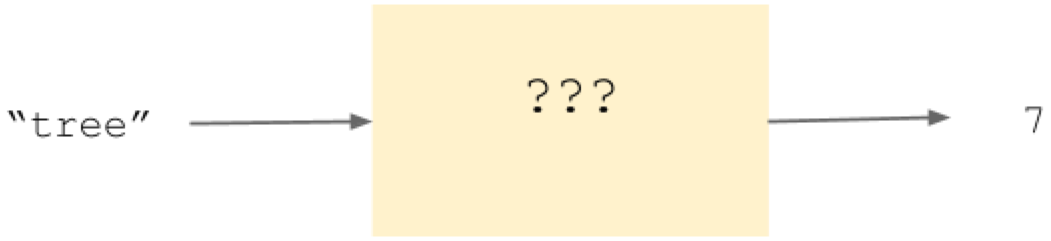
```
get:   x = array[7]
put:   array[13] = y
```

...whereas get and put of the i-th item in a linked list is O(N). (yuk)

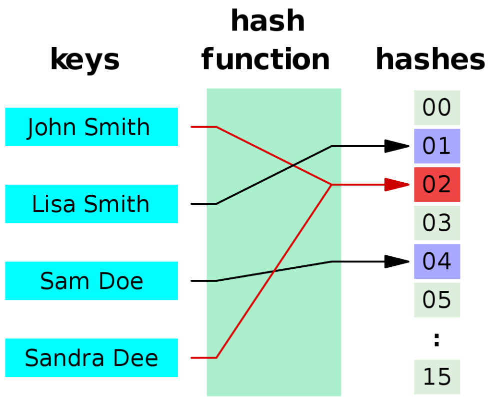
But, is seems like we're stuck if things aren't integers?!?!

```
get:   x = array[ “tree”]   // wrong
put:   array[“squirrel”] = y   // huh?!?
```

Solution – we need a mystery box that can convert objects (like Strings) to integers!



This mystery box is called a **hash function**.
The input is an object. The output is an integer. Magic!



With a hash function, we can use arrays to efficiently store pretty much anything.
You call this array (you guessed it)... a **hash table**.

D. Hash table = happy times ☺

So, a **hash table** is an array used to get/put objects with average O(1) performance.
Other names for this are associative array, dictionary, or map.

Typically, hash tables store **key-value pairs (K, V)**. You can also think of this as a search key and its data. For example, key may be student name and value is the Student object.

Hash table ADT:

Operation	Java HashMap methods

create(size, load_factor)	`HashMap<K, V>()`
	`HashMap( int size, float loadFactor)`

value get(key)	`V get( Object key)`
put(key, value)	`put( K key, V value)`
value remove(key)	`V remove( Object key)`

All these operations should average O(1) performance.
We'll talk about the worst case later.

E. (is for) Examples

Here's a very small (bogus) example. It's a hash table of capacity=5. Assume a Person class with name, age, and zip code. The key is the Person name; the value is the Person object.

Person p1 = {"Bill", 29, 60565}; **put("Bill", p1)**;
hashCode("Bill") = 12,800,653; hash = 12,800,653 % 5 = 3

0	1	2	3	4
			"Bill" {"Bill,29,60565}	

Person p2 = {"Moz",78,60007}; **put("Moz",p2)**;
hashCode("Moz") = 9,695,555; hash = 9,695,555 % 5 = 0

0	1	2	3	4
"Moz" {"Moz",78,60007}			"Bill" {"Bill,29,60565}	

Person p3 = {"TY",19,78212}; **put("TY",p3)**;
hashCode("TY") = 67,097,511; hash = 67,097,511 % 5 = 1

0	1	2	3	4
"Moz" {"Moz",78,60007}	"TY" {"TY",19,78212}		"Bill" {"Bill,29,60565}	

Person p4 = **get("Bill")**;
hashCode("Bill") = 12,800,653; hash = 12,800,653 % 5 = 3

0	1	2	3	4
"Moz" {"Moz",78,60007}	"TY" {"TY",19,78212}		"Bill" {"Bill,29,60565}	

Person p5 = **get("Godfrey")**;
hashCode("Godfrey") = 22,900,682; hash = 22,900,682 % 5 = 2

0	1	2	3	4
"Moz" {"Moz",78,60007}	"TY" {"TY",19,78212}		"Bill" {"Bill,29,60565}	

Return is p5 = null

F. How do hash functions work?

Hash functions seem to be the “magic” here. How are objects turned into integers?
There are two steps: 1) hash, and then 2) modulo.

1) hash – Example: hashCode for String in Java, a multiplier of each char s[i] is summed.

$$h(s) = \sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i}$$

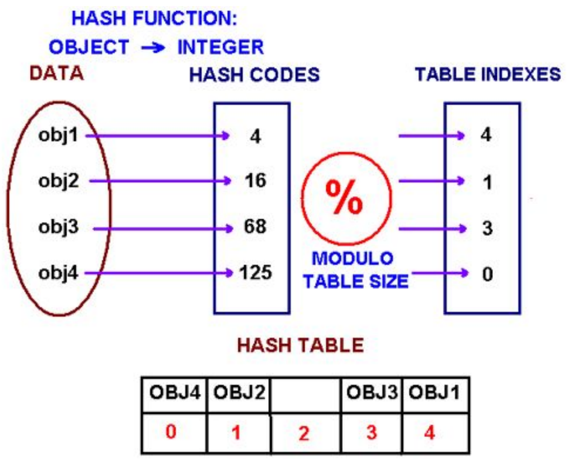
Or, a little less math-y... it's the sum of unicode value of each character (Si) multiplied by a power of 31 (a nice prime number).

$$\texttt{hashCode(S) = S0 * } 31^{(n-1)} \texttt{ + S1 * } 31^{(n-2)} \texttt{ + ... + Sn-1 * } 31^{0}$$

There are many, many different hash functions: en.wikipedia.org/wiki/Hash_function
More important – objects are represented as numbers in your program; you can always mangle these numbers in some fashion to come up with a hash code.

2) modulo – Once you've hashed your object, the next step is to modulo that number by the size of your hash array, M (in Java, hash % M). This will give you a number in the range of [0-M], which is now an acceptable index into the hash array.

```
hashcode( S) = hash( S) % M
```



To work properly, a hash function is required to have these three properties:

1. easy/fast to compute – defeats O(1) otherwise
2. random/uniform distribution – so we minimize minimize hash code repeats
3. reproducible – we get the same hash for an object each time

G. Oops. Collisions!

There's a problem with our hash function scheme. When I hash two different objects, there's nothing preventing them from having the same hash code. (yikes!)
Example: With an array of size 17... hash 2 strings, resulting in the same hash code:

```
hashCode( “Prof”) % 17 = 11
hashCode( “Bill”) % 17 = 11
```

Two objects with the same hash code, that's called a **collision**.
The algorithm we use to properly handle collisions is called **collision resolution**.
The 2 most common ways to do this are: linear probing and chaining

Linear probing

Algorithm – Collision at spot N in your array... increment (N+1) until you find an open array slot. This is also called **open addressing**.

Starting at our hash code index into the array, looking for an open (non-null) slot is pretty straightforward. But there are 2

1. **circular –** Our iteration must be circular... at the end of the array, we cycle back to the beginning, slot 0; use modulo the size of the array (% array.length)
2. **remove woes –** We have to be careful removing a node, just leaving the slot null may break a probing chain. So, it's more complicated. Slots with removed nodes are flagged, so that probing continues past that slot. We can, however, use the slot for adding a value to the table.

/* example on the board! */

Chaining

Algorithm – Collision at spot N in your array... add the object to a linked list (bucket) at that spot. This is also called **closed addressing**.

We also call this approach “the easy one”... there are no tricks here.

Advantage: This is an easy approach. Create a linked list at each array slot, add collisions to the end of the list.
Disadvantage: An array at each array slot... yeesh, that's pretty expensive! Also, if we have too many collisions, then our hash table suddenly turns into a linked list, O(N).

/* example on the board! */

H. This hash table is too crowded

Think about it... as your hash table gets more crowded, collisions become more likely.
If your hash table is 10% full (and your hash function is truly random/uniform), then the chance of a collision for putting a new key is 10%. If the hash table is half full, your chances rise to 50%.

As such most hash tables track their **load factor**, the percentage of array slots currently being used in the hash table.

If the load factor is too high, then we **resize** the array for our hash table. We don't want to wait till our array is totally full, too many collisions. An easy rule of thumb: resize when the load factor exceeds 75%.

Resizing your array is two steps:

1. Create a new array, twice as big as the last one
2. Rehash every item into the new array. You must do this, as the size of the array is reflected in your hash code (% array.length).

If you didn't resize, the performance of your crowded hash table would approach that of a list, O(N). Indeed, this is the **worst-case performance** for hash table operations, O(N)

I. This page left intentionally blank

en.wikipedia.org/wiki/Intentionally_blank_page

J. Java ♡ hash tables

Hash tables are an important part of Java.

hashCode() method

In Java, you can create a hash function for your class by overriding Java's hashCode() method. Java loves hash tables so much that hash functions are built-in to every Object.

The hashCode() method is defined in Object and therefore inherited by every other object.

```
int hashCode()
```

Details:

- ➢ Like String, most classes will override the hashCode() method if there is an expectation of them being used in a hash table.
- ➢ The default hashCode() in Object uses the pointer to the object in constructing its hash code. This is often undesirable. Sort of like == versus equals().
- ➢ Two objects that are equal in Java (using the equals() method) should also return the same hashCode().

HashMap, HashSet

These are the two most popular hash tables in Java:

HashMap – a JCF Map implemented using a hash table,
docs.oracle.com/javase/8/docs/api/index.html?java/util/HashMap.html

HashSet – a JCF Set implemented using a hash table,
docs.oracle.com/javase/8/docs/api/index.html?java/util/HashSet.html

K. Conclusion

Conclusion #1: Hash tables are a data structure VIP. You must understand them!

Here's a good practical hash table quote of the day (QOTD):

> **QOTD**
> "Hash tables are the most commonly used non-trivial data structures, and the most popular implementation on standard hardware uses linear probing, which is both fast and simple."
> - en.wikipedia.org/wiki/Linear_probing

Conclusion #2: Hash tables with linear probing. That's the $$$.
thanks… yow, bill

L. Appendix: Terms

First off, I skipped a couple terms because… oh never mind (smile):

- ➔ **double hashing –** this is another collision resolution scheme; on collision, try a second hash function; not very popular because you still need to handle collisions after that

- ➔ **quadratic probing –** sometimes hash functions cluster hash codes together, which is bad; quadratic probing is different that linear probing in that it doesn't just increment, it skips ahead by a larger amount (presumably to avoid the congestion). For example:

```
H + 1, H + 22, H + 32, H + 42 + ... H + k2
```

Ok, let's go...

hash table – an array that uses a hash function to efficiently store key-value pairs
hash function – a function that maps an object into an integer for use in a hash table
hash code – the return value (integer) of a hash function

map – data structure that holds key-value pairs; keys are mapped onto values
key-value pair (K, V) – a value and its associated search key

collision – when two objects result in the same hash code
collision resolution – algorithms that handle hash table collisions when they happen
linear probing – a collision algorithm where we increment through an array to find an available slot for an object
open addressing – algorithm like linear probing, where all array slots are available
chaining – a collision algorithm that uses a linked list at each array slot to store collisions
closed addressing – algorithm like chaining, where only the hash code slot is used

load factor – the percent of array that is occupied in a hash table
resize – increasing the array size in a hash table, usually when the load factor gets too large
rehash – hashing objects again with a new array size after resizing
hashCode() – hash function method defined for Object in Java, used in HashMap & HashSet
HashMap – a JCF Map interface implemented using a hash table
HashSet – a JCF Set interface implemented using a hash table

M. Appendix: Pseudocode

This appendix provides pseudocode for hash tables with linear probing and chaining.

Remember, our hash table operations are:

```
create
put( v, k)
v get( k)
v remove( k)
```

I decided to put my pseudocode into a separate document.
It's on the class website, wtkrieger.faculty.noctrl.edu/csc210-spring2018.
thanks… yow, bill

www.zippythepinhead.com