CS 112: Data Structures

Sesh Venugopal

Hash Table – Part 2

We started out with a quest to find a structure that would search in O(1) time in the worst case, but found that the worst case search time is actually O(n)

We are going to keep at it, and find a way to get closer to our objective, with the help of a quantity called the *load factor* of a hash table

Load Factor

The load factor (notated as lambda or alpha) is n/N

$$\lambda = n/N$$

where n is a number of keys in the hash table (i.e. in all the chains put together), and N is the size of the hash table array

So, for instance, if the hash table size N is 10, and the number of keys in all the chains is 100, the load factor is 100/10 = 10

What we would ideally want is for all the chains to be of equal length – this would give the best possible performance

If all chains were of equal length, then the n keys would be evenly distributed over all the array locations, so that each chain would be of length $n/N = \lambda$

So if the load factor was, say, 2, then there would be ideally 2 keys in each chain

How do we ensure such an even distribution? It is completely up to the hash function because the mapping depends on the hashcode generated by the function

EXPECTED search time

We need a "good" hash function, one that would *uniformly distribute keys* over the array locations (this includes mapping the hashcode to an index into the hash table array)

Assuming such a good hash function, we can EXPECT each chain to be of length λ

And then the EXPECTED running time of search would be $O(\lambda)$

But, if want an O(1) time, we don't want the load factor to be a variable depending on n – we want it to be a constant.

Wanting the load factor to be a constant means setting a numeric threshold for it that should not be crossed.

So, for instance, say we set the threshold to 2.5.

If the table size, N, is 10, then as long as number of keys, n, is less than or equal to 25, the load factor will be under or at the threshold

In other words, we can keep inserting keys up to 25 times. On the 26th insert, the load factor will cross the threshold, to 26/10 = 2.6 – at this point we need to take corrective action

What corrective action to take when the load factor crosses the imposed threshold?

Since the load factor = n/N, and we can't limit the keys that are inserted (that's up to the user), the only course of action is to increase N so that the load factor drops to below the threshold

The conventional approach is to double the size of the array, so that we set up a new hash table array with size N*2

After setting up a new array, we will need to move all keys from the old table to new, but we can't just move a chain from the old table to the same index in the new table!

Say for instance a key's computed hashcode was 14. In a hash table of size 10, it would have mapped to index 14%10 = 4. But in a hash table of size 20 (double the old size), it would map to index 14.

So the key would need to be added to the chain at index 14 in the new table.

Also, different keys in a chain may have to be moved to different indices, i.e. remapped

For instance, consider key K1 with hashcode 14 and key K2 with hashcode 24. Both of these would be mapped to index 4 in a hash table of size 10

But in a table of size 20, K1 would map to index 14, and K2 would map to index 4

Rehashing

So, in summary, when the load factor crosses the threshold, then:

- Allocate a new array which is double the size of the current array
- Remap all keys to the new array, and set up new chains

This process is called REHASHING

Since the hashcode has already been computed once for a key, and it won't change just because the array size is different, it pays to store the hashcode along with the key in the hashtable – it would be a waste of time to recompute all hashcodes again.

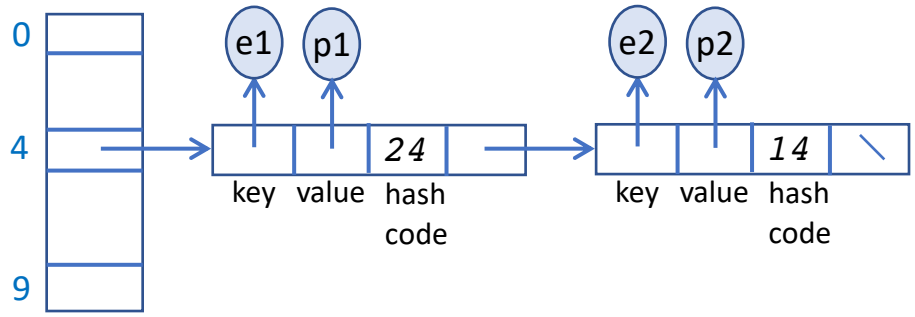
In fact, a standard hash table implementation stores (key,value,hashcode) triplets.

For example, a website might store users in a hash table with key=email, value=password, plus the hashcode for the email.

So that when a user registers on the website, the email is hashed (which computes hashcode) and mapped to an array index, and a new (email,password,hashcode) triplet is inserted at the front of the chain at that index

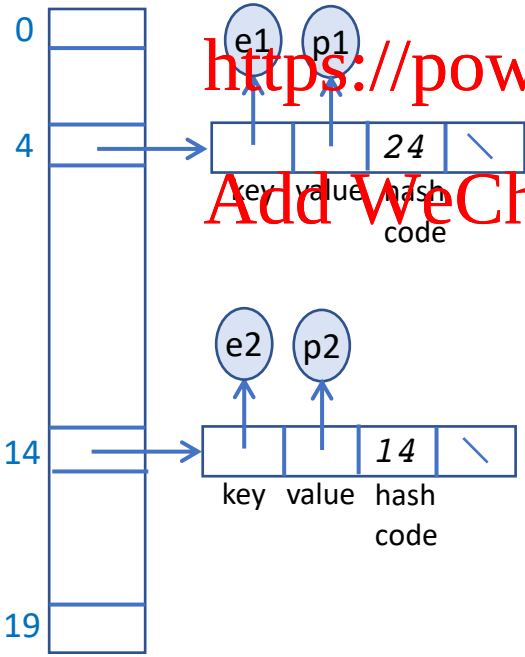
And when a user logs in to the site, the email they entered is hashed and mapped to an array index (as for insert), then searched for in the chain at that index. If there is a match, the associated password is retrieved and matched against the password the user entered.

N=10



Rehashing would take O(n) time, since each of the n keys will need to be remapped to the new table

N=20



EXPECTED search time is O(1)

Since we set the load factor threshold to a constant that is independent of the number of keys in the hash table,

If the hash function distributes keys uniformly over the table,

the *expected* running time for search is O(1)!

Java Implementation of Hash Table

java.util.HashMap

This is a generic class with two generic type arguments, one for the key and the other for the value:

So


```
HashMap<String,ArrayList<Integer>>
```


means key is of type String, and value is an array list of integers

Usage of this class is detailed in the `HashMapDemo` program (Resources -> Week 10) - the code has ample commentary on various aspects of manipulating a `HashMap`

java.util.HashSet

This is a generic class with a single type argument. It is used when you don't have a key,value separation and just want to store objects

So `HashSet<Point>`

means the hash table stores Point objects – each Point instance serves as both key and value

Usage of this class is detailed in the `HashCodeDemo` program (Resources -> Week 10)

Computing the hash code

A hash table class (such as HashMap or HashSet) doesn't itself implement a hash function

When a key is inserted/searched in a hash table, the hash table calls the key's `hashCode` method to get the hash code (and then does the mapping using modulus table size, and rest of the logic for insert/search)

The good news is that the String class has a `hashCode` method, so if the key is not a String type, it can get the string equivalent using its `toString` method, then call `hashCode` on the string equivalent – see how this is done in the `Point` class in `HashCodeDemo`