Hash Table

CS 112 Spring 2020 – March 24

Sesh Venugopal

Shooting for O(1) worst case search time!

Until now, we have seen that the best worst-case time for searching in a set of n keys is $O(\log n)$, attributed to binary search of a sorted array and search in an AVL tree.

But why stop here? Wouldn’t it be great if the search time could be even better than $O(\log n)$, say the ultimate best time possible of $O(1)$?

Use key as index into an array?

We know that an array supports $O(1)$ time random access. But such a random access is based on the *position* of an item in the array, and not on its *key*.

Could we somehow use the *key* of an item as a position index into an array?

Suppose we wanted to store the keys 1, 3, 5, 8, 10, with guaranteed O(1) access to any of them.

If we were to use these keys as indices into an array, we would need an array of size 10. (Actually 11, since the array would come with the 0-index position, which will not be used.)

We would simply mark each cell of this array with *true* or *false,* depending on whether the index for the cell was one of the keys to be stored.

1	2	3	4	5	6	7	8	9	10
T	F	T	F	T	F	F	T	F	T

But there is a catch: the space consumption does not depend on the actual number of entries stored.

Instead, it depends on the *range* of keys—the gap between the largest and smallest keys. For instance, if only two entries were stored, but they were 1 and 10,000, we would need 10,000 units of space.

Suppose we want to store 100 integers in such an array. Without any a priori estimate of the range, we have to be prepared to store pretty much any key that comes along, be it 1 or 1,000,000.

Even if the range were estimated beforehand, there is no reason to expect it to be within a small multiple of 100 so that the space consumption is affordable.

What if we wanted to store strings? For each string, we would first have to compute a numeric key that is equivalent to it. How do we do this? And what if two different strings translate to the same number?

Moreover, the numeric key equivalents of different strings may be in a very large range even though they are of the same length.

This would therefore require a disproportionately large array for storage, wasting much space in the bargain.

Use key to derive index into an array – hash table

It appears that using numeric keys *directly* as indices is out of the question for most applications

The next best thing is to select a certain array size, say *N*, based on space affordability, and use the key of an entry to *derive* an index within the array bounds

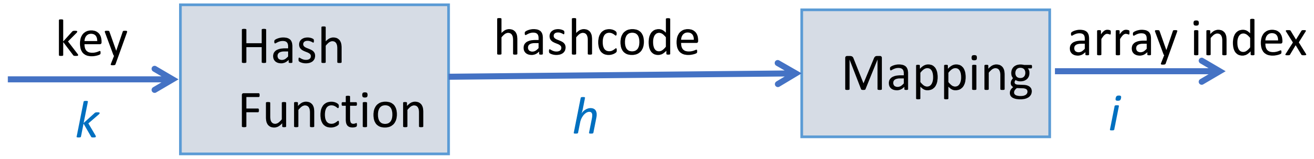
Such an array is called a hash table

Hashing: hashcode, hash function, hash table

Hashing is the process of computing a numeric value, called the hashcode, from a key which could be an object of any type

This hashing is done by a hash function

The hashcode is then mapped to an index in the hash table array, and the key is stored at this index



Here’s an example of hashing and mapping:

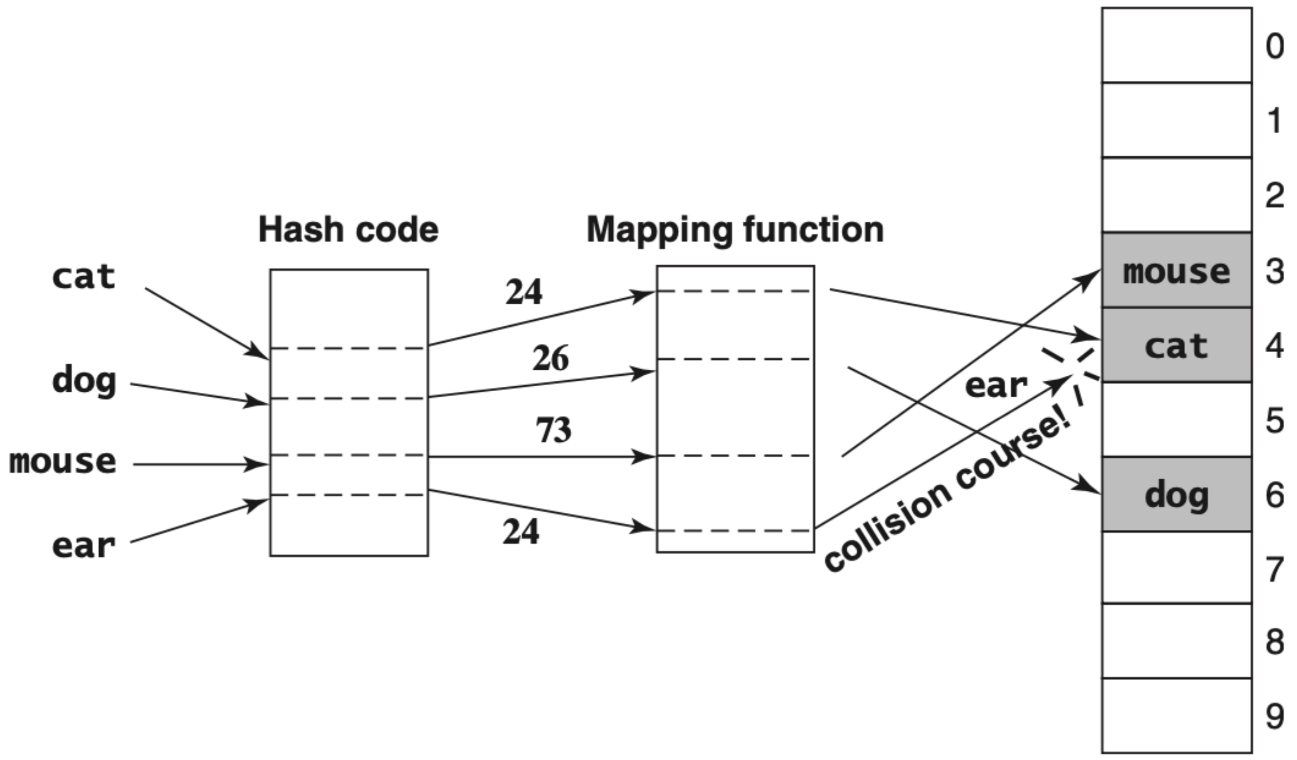
Say we need to store the strings “cat”, “dog”, “mouse”, and “ear” in a hash table

For the hash function, we will use a process that computes the hashcode by simply adding the alphabetic positions of the letters of a word:

$$\text{cat} \equiv 3 + 1 + 20 = 24$$
$$\text{dog} \equiv 4 + 15 + 7 = 26$$
$$\text{mouse} \equiv 13 + 15 + 21 + 19 + 5 = 73$$
$$\text{ear} \equiv 5 + 1 + 18 = 24$$

Say we start with a hash table of size 10.

To map hashcode *h*, we will use the function *h* mod 10

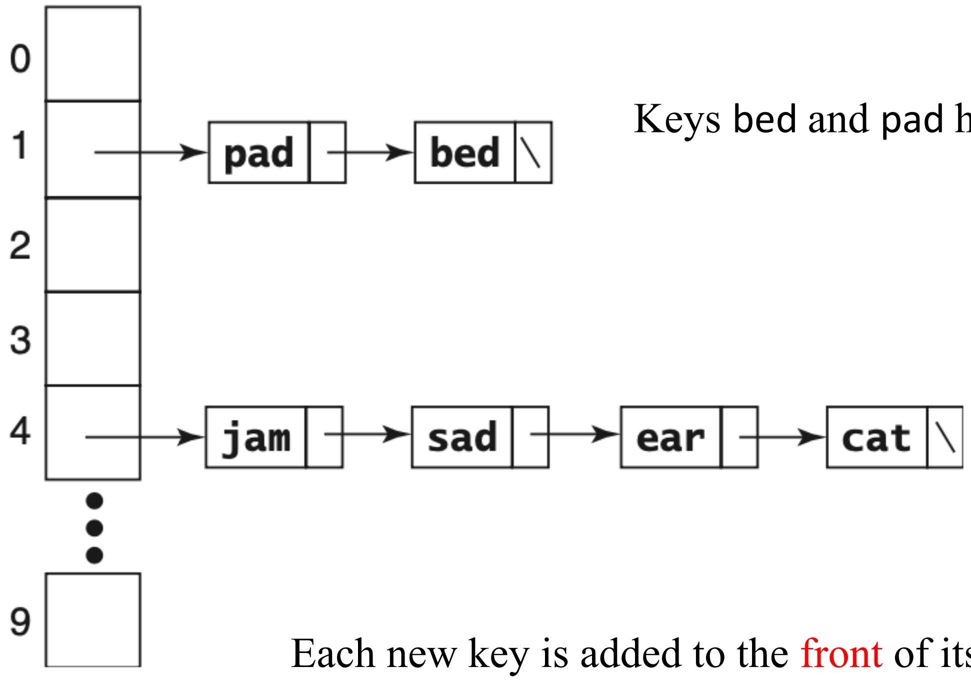


The string ear collides with the string cat at position 4.
How to resolve the collision? In other words, where to store ear?

Collison resolution using chaining

The obvious (and standard) approach to resolve collisions is to store all keys that map to the same location in a linked list off that location.

This is called chaining, each linked list is a chain:



Keys bed and pad hash to location 1.

Keys cat, ear, sad, and jam all hash to location 4.

Each new key is added to the front of its chain.
(There is *no ordering* among keys in the same chain.)

Worst Case Running Times for insert/search/delete in a hash table with n keys

The hash function runs in O(1) time, since it works on a given key only, independent of all the other keys in the hash table.

The mapping function is assumed to be the modulus function, which runs in O(1) time

With these in place, we have the following running times:

- Worst case time to insert is O(1):
  Hash + map + insert at front of linked list at mapped array index

- Worst case time to search is O(n):
  - Hash + map is O(1) time
  - In the worst case, all n keys map to the same array index, and the length of this single chain becomes n.
  - Searching for a key means sequentially searching in this chain, for an O(n) worst case time

- Worst case time for delete is O(n) since it depends on search

We started out with a quest to find a structure that would search in O(1) time in the worst case, but the hash table is a far cry from it – worst case search time is O(n)!

But there is hope! We need to dig a bit deeper, and to do this, we need to work with what is 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 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, is 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(λ)

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

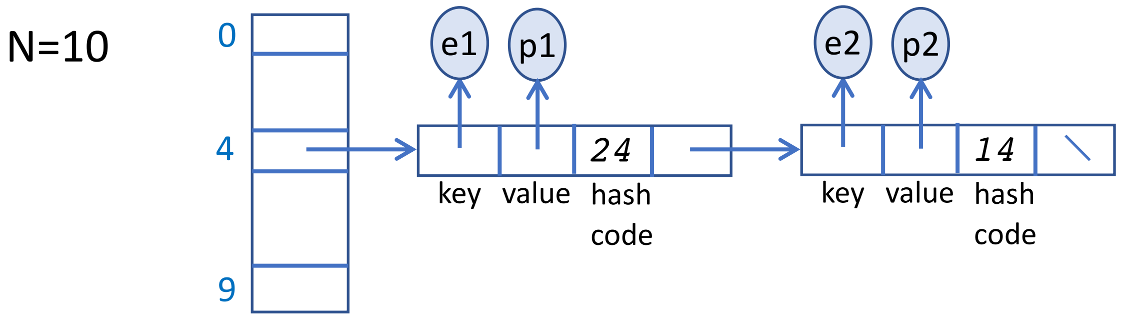
Since the hashcode has already been computed once for a key, and it won't change just because the array size is different, it pays to store the hashcode along with the key in the hashtable – it would be a waste of time to recompute all hashcodes again.

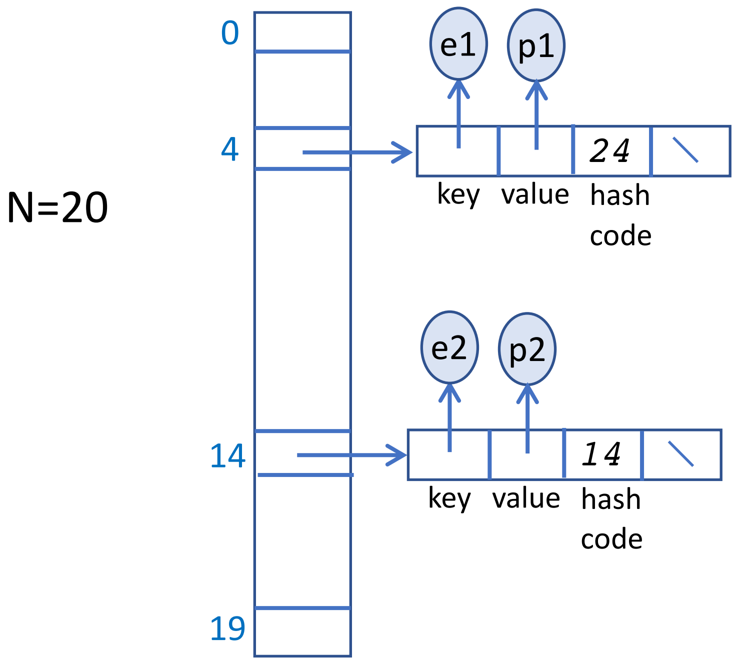
In fact, a standard hash table implementation stores (key,value,hashcode) triplets.

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.





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

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 -> Venugopal -> Mar 24) - 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 -> Venugopal -> Mar 24)

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`