CS 112: Data Structures

Sesh Venugopal

Hash Table – Part 1

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 there? Wouldn't it be great if the worst-case search time could be even better than $O(\log n)$, say the ultimate best time possible time of $O(1)$?

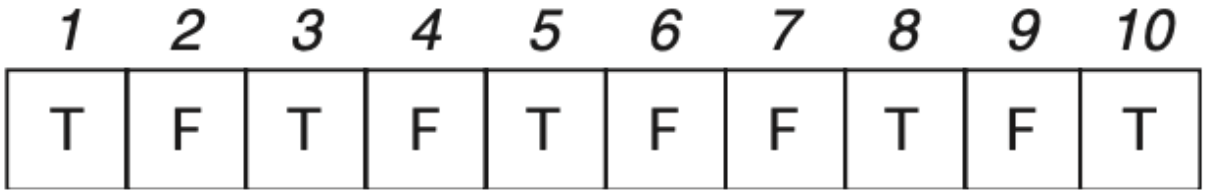
We know that an array supports $O(1)$ time random access. But such a random access is based on the *position* of an item in the array, and not on its *key*.

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.



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 10,001 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.

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

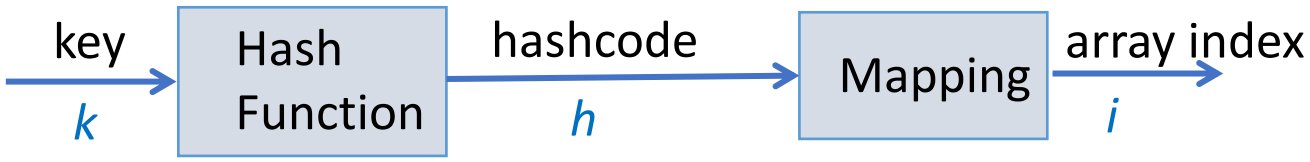
Such an array is called a hash table

Use key to derive index into an array – 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:

$$\text{cat} \equiv 3 + 1 + 20 = 24$$

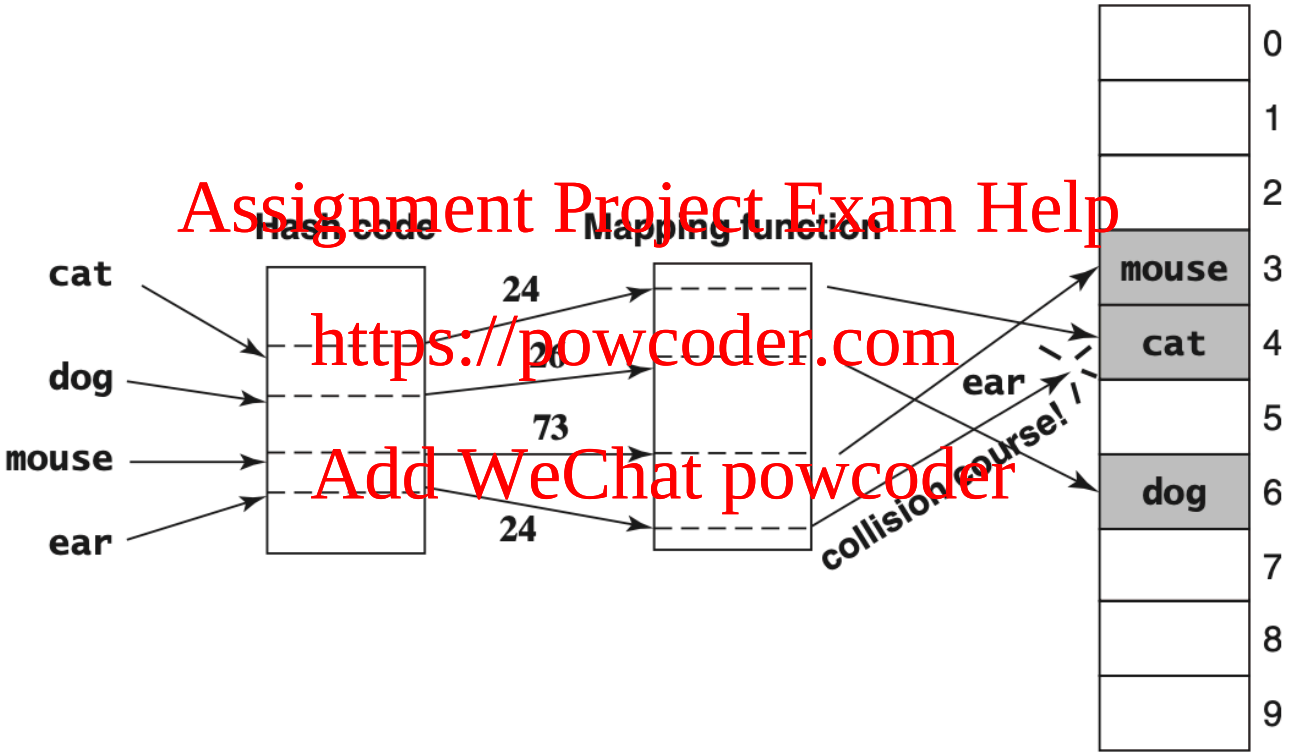
$$\text{dog} \equiv 4 + 15 + 7 = 26$$

$$\text{mouse} \equiv 13 + 15 + 21 + 19 + 5 = 73$$

$$\text{ear} \equiv 5 + 1 + 18 = 24$$

Say we start with a hash table of size 10.

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

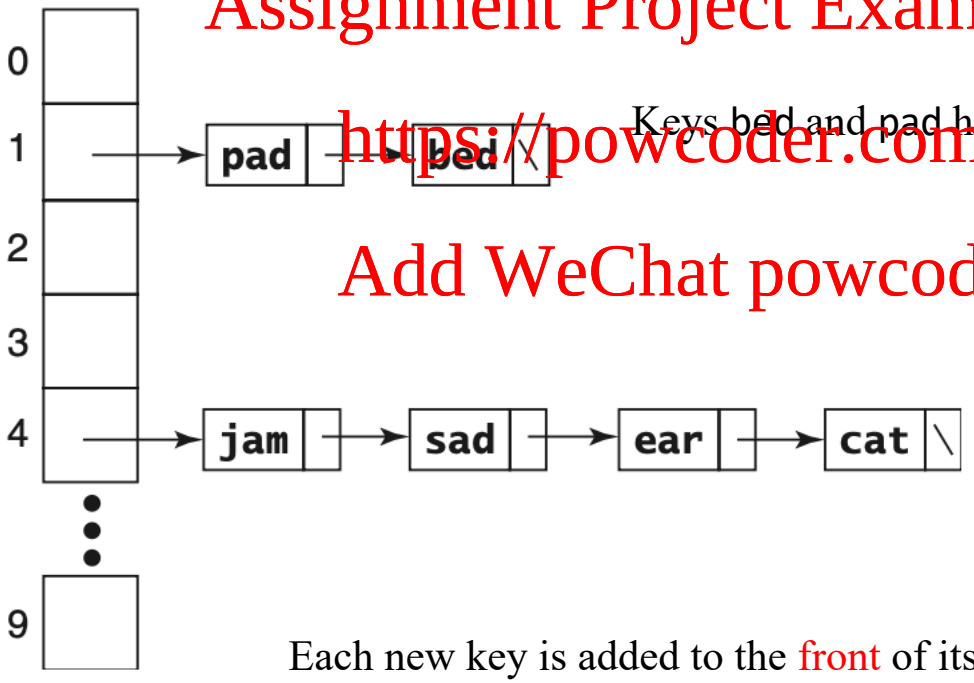


The string ear collides with the string cat at position 4.
How to resolve the collision? In other words, where to store ear?

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:



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