Chapter 6

Hashtables

6.1 Introduction

We've been through various versions of lists and binary search trees with O(n) and O(log n) search times respectively; even with certain constraints they only get up to O(log n). Can we do better than this for searching? Well the fact that I'm asking this questions suggests yes and the fact that you've already been introduced to this topic also suggests yes. We can use hashing to achieve this. They're called hash tables and they are just like tables in real life. No, not the furniture kind.

6.2 Description

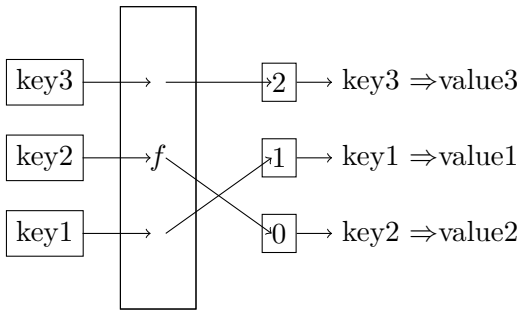
Hash tables are a data structure that maps keys to values. You may imagine it like this:

key1	value1
key2	value2
key3	value3

The keys are hashed to calculate the location where the data will actually be stored in the underlying data structure.

6.3 Hashing

Hashing is used to distribute the key value pairs across the underlying data structure. All it is is a function that takes your keys and gives you a number. For example, to expand on our previous visualization of a hash table, we may see something like this:



As you can see, the hashing function f takes the keys 1-3 and turns them into numbers. In other words:

$$f(key1) = 1$$
$$f(key2) = 0$$
$$f(key3) = 2$$

When we apply f to one of those keys, we are certain to get a specific output. But you may be wondering about a few cases:

$$f(a) = x, \text{ and } f(b) = x$$
$$g(a) = x, \text{ and } g(a) = y$$

(6.5) shouldn't happen because that would ruin the entire hashing idea; could you imagine putting key3 into the first example hashing function and getting 0 and 2 sometimes? Awkward! However, (6.1) is possible.

6.4 Collision Resolution

$$f(a) = x, \text{ and } f(b) = x$$

Two inputs map to the same output. This is virtually unavoidable for any function when given enough keys. Can we think of a simple mathematical example?

$$\begin{aligned} \text{given } f(x) &= 7 \\ f(0) &= 7 \\ f(1) &= 7 \\ etc. & \end{aligned}$$

Here is a hash function that always returns 1; not very good, but it illustrates the point quite well. As such, there are two categories of collision resolution.

6.5 Closed Addressing

Closed addressing i.e. separate chaining is where we create buckets to hold multiple entries at each index. The buckets can be any kind of data structure that supports searching, insertion, and removal e.g. lists or binary search trees. Let's look at an example given the simple hashing function.

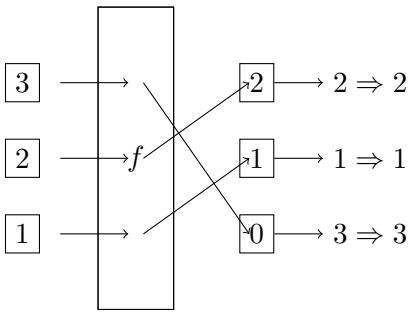
$$f(x) = x\%3$$

Functions typically end with modulo to limit the possible outputs in order to fit them properly into an array which is one possible underlying data structure behind a hash table.

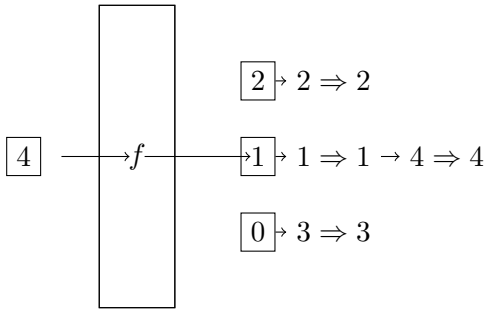
6.5.1 Closed Addressing Insertion

Let's try inserting key value pairs in the form of $x \Rightarrow x$ where x is the integers between 1 - 5 inclusive i.e. $1 \Rightarrow 1, 2 \Rightarrow 2$ (the numbers are keys for themselves for simplicity).

Let's insert 1 - 3 because those are simple and don't cause any collisions. $f(1) = 1, f(2) = 2, f(3) = 0$.



Alright, so now we insert 4. Where does 4 go? $f(4) = 1$ so it will go to the structure holding all the values for index 1.



And how about 5? $f(5) = 2$ so it will go to the structure holding the values for index 2.

$$2 \mapsto 2 \Rightarrow 2 \rightarrow 5 \Rightarrow 5$$

$$1 \mapsto 1 \Rightarrow 1 \rightarrow 4 \Rightarrow 4$$

$$0 \mapsto 3 \Rightarrow 3$$

6.5.2 Closed Addressing Searching/Deletion

Alright that was pretty simple. Let's try deleting some stuff from the table like the value 7. First we need to check if the table have the 7 in it. Apply the function to get the index: $f(7) = 1$ and get the structure holding everything under index 1.

$$1 \Rightarrow 1 \rightarrow 4 \Rightarrow 4$$

Do you see $7 \Rightarrow 7$ in there? Nope, so 7 is not in this hash table and there's nothing else to do. How about 4? $f(4) = 1$ and we reach the same buckets:

$$1 \Rightarrow 1 \rightarrow 4 \Rightarrow 4$$

Now we just have to remove 4 from the underlying data structure and that depends on what the data structure is.

6.6 Open Addressing

Open addressing does not use buckets, but rather utilizes probing to find free slots in the underlying data structure. Probing, like the name implies, is where the hash table will examine its slots in a consistent fashion until it finds a free one, which is where the data will be stored. Examples of probing are

- linear probing
- quadratic probing
- double hashing

One downside to probing is that you can only store as many entries as there are spaces, however, implementations will typically resize on the fly so there is no limit.

6.6.1 Linear Probing Insertion

Let's go back to our previous example with the function $f(x) = x\%3$. Let's insert 3, 6, and 9. After inserting 3 into the empty table, it will look like this:

$$\boxed{2}$$

$$\boxed{1}$$

$$\boxed{0} \rightarrow 3 \Rightarrow 3$$

Now when we insert 6, we will notice a collision because $f(6) = 0$. With linear probing, what we do is then use $f(6) + 1 = 1$ which ends up like this

$$\boxed{2}$$

$$\boxed{1} \rightarrow 6 \Rightarrow 6$$

$$\boxed{0} \rightarrow 3 \Rightarrow 3$$

And now for 9. $f(9) = 0$ is taken, $f(9) + 1 = 1$ is taken, so we move to $f(9) + 2$ which is free

$$\boxed{2} \rightarrow 9 \Rightarrow 9$$

$$\boxed{1} \rightarrow 6 \Rightarrow 6$$

$$\boxed{0} \rightarrow 3 \Rightarrow 3$$

6.6.2 Linear Probing Searching/Deletion

Searching is like insertion; you check $f(x)$, $f(x) + 1$, $f(x) + 2$... until you come across an empty cell or the correct key. However, it is important to keep this in mind when removing. Can you just remove an entry from the hash table and be done with it? No!

Consider this example (a horizontal table to save space, 0 indexed, same key value scenario i.e. $x \Rightarrow x$, with $f(x) = x\%10$). This table is generated by inserting the numbers left to right.

20	_	22	33	42	55	16	65	44	9

Now remove 55 and try to search for 44. What happens? $f(44) = 4$ so we start at index 4 (42). That's not right so we go to $f(44) + 1 = 5$. Well that's an empty space since we removed 55 so I guess 44 isn't in the table... but obviously it is! How can we avoid this problem?

What we do is after removing 55, we look to the right for a key that hashes to a number less than or equal to its index (5) and move it there. We see 66 work in this case $f(65) = 5$ so we move 65 to 55's old spot.

20	_	22	33	42	65	16	_	44	9

And now we have another empty space. Time to keep going! Right away we see $f(44) = 4$ which is less than the index 66 was at (7). So move 44 to index 7

20	_	22	33	42	65	16	44	_	9

We wrap around to 20. Now this is the interesting case; if we just follow the previous rules, we would move 20 to the empty spot and then be done because there is a free spot after 20. But searching for 20 wouldn't work. So we actually don't pick 20 because it is as far left as it can be already. When we move over 20, we find an empty space and stop, thus ending the shifts. By shifting things to their earliest possible locations, we can now properly execute searches after deleting.

A simpler way to deal with deletion is just to mark each cell with one of 3 states: empty, occupied, deleted.

6.6.3 Quadratic Probing

Quadratic probing is very similar to linear hashing. While linear hashing may be though of as two functions f and g such that

$$g(x) = \text{hashing function}$$
$$f(i) = g(x) + i$$

where i starts at 0 and is incremented until a space is found, quadratic probing can be represented by these functions

$$g(x) = \text{hashing function}$$
$$f(i) = g(x) + i^2$$

6.6.4 Double Hashing

And just to expand again, double hashing can be represented as

$$g(x) = \text{hashing function 1}$$
$$h(x) = \text{hashing function 2}$$
$$f(i) = g(x) + i * h(x)$$

6.7 Performance Analysis and Load Factors

Removed for now.

6.8 Examples

Removed for now.