Exploration: Hash Table Collisions

Introduction

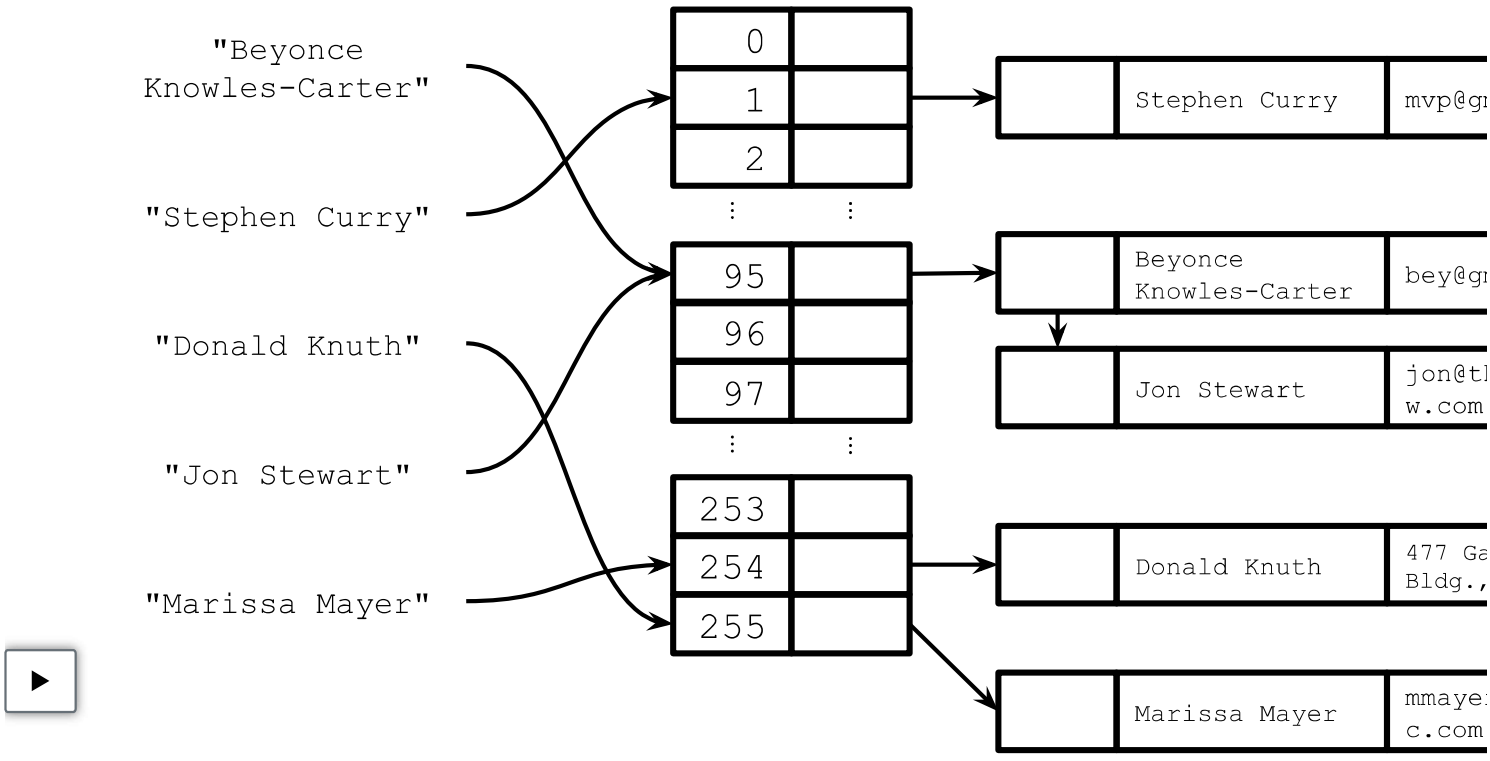
In the last exploration, we discussed how hash functions don't always work exactly how we might want. Sometimes, they will generate the same output for two different inputs. We call this a collision. This is something that we will have to accommodate. This section will discuss the ways we can do that.

Collision resolution with chaining

We can eliminate collisions entirely if we allow multiple keys to share the same table entry. To accommodate multiple keys, linked lists can be used to store the individual keys that map to the same entry. The linked lists are commonly referred to as buckets or chains, and this technique of collision resolution is known as chaining.

When a collision occurs, the new element is simply added to the collection at its corresponding hash index.

Linked lists are a popular choice for maintaining the buckets. Other data structures may be used as well (e.g., a dynamic array or even a balanced binary tree). Here's what a hash table with linked list-based chains might look like:



A hash table where collisions are chained in linked lists

In a chained hash table, accessing the value for a particular key would follow this procedure:

- Compute the element's bucket using the hash function
- Search the data structure at that bucket for the element using the key (e.g., iterate through the items in the linked list).

Adding or removing an element would follow a similar process, except we would add or remove the element to orfrom the appropriate bucket's data structure.

The **load factor** of a hash table is the average number of elements in each bucket:

```
λ=n/m
```

- **λ** Is the load factor
- **n** is the total number of elements stored in the table
- **m** is the number of buckets

In a chained hash table, the load factor can be greater than 1.

In general, as the load factor increases, operations on the table will slow down. For a linked list-based chained table, the average number of links traversed for successful searches is **λ** / 2. For unsuccessful searches, the average number of links traversed is equal to **λ**.

Therefore, to maintain strong performance with a hash table, we often double the number of buckets when the load factor reaches a certain limit (e.g., 8). In other words, the hash table array could be implemented with a dynamic array whose resizing behavior is based on the load factor. How would we actually perform the resize?

Recompute the hash function for each element with the new number of buckets (i.e., for use with the mod operator (%)).

What is the average-case complexity of a linked list-based chained hash table?

- Assume that the hash function has a good distribution.
- The average case for all operations is O(**λ**).
  If the number of buckets is adjusted according to the load factor, then the number of elements is a constant factor of the number of buckets i.e.: `λ = n/m = O(m)/m = O(1).`

In other words, the average case performance of all operations can be kept to constant time. The worst-case complexity is `O(n)`, since all of the elements might end up in the same bucket.

Collision resolution with open addressing

The open addressing method for resolving collisions involves probing for an empty spot in the hash table array if a collision occurs. When using open addressing, all hashed elements are stored directly in the hash table array.

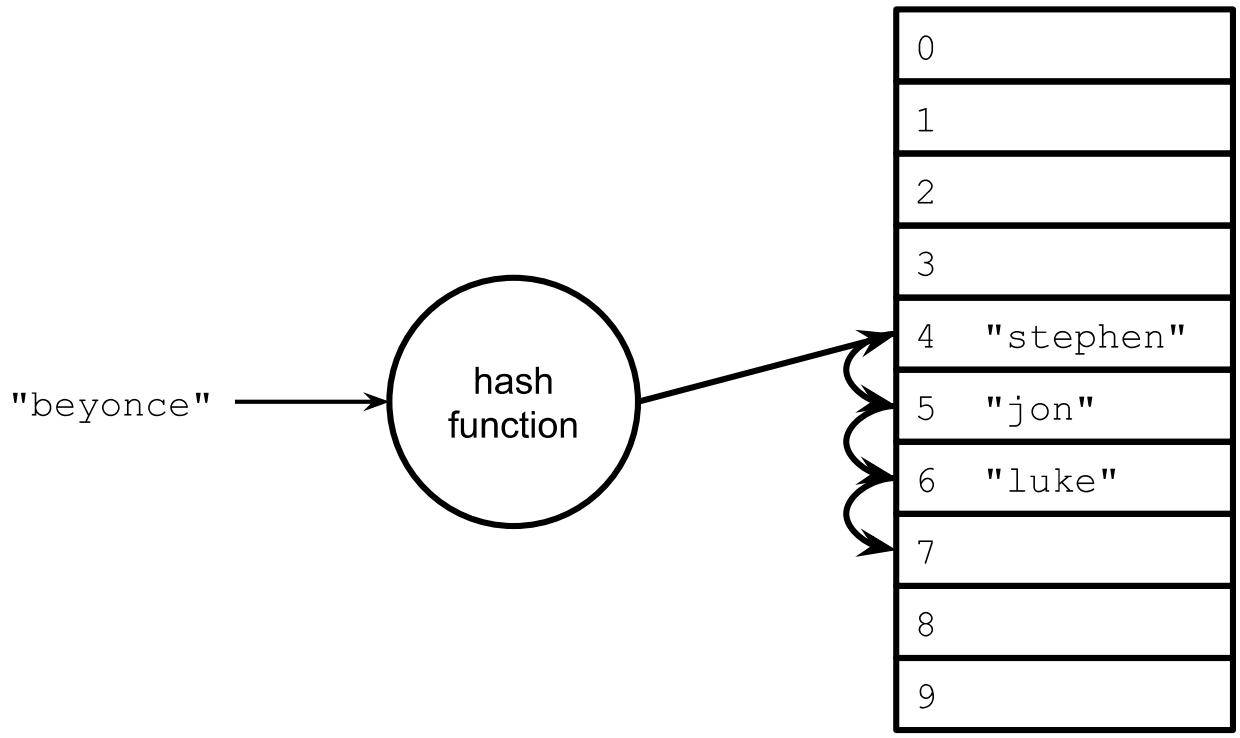
The procedure for inserting an element in an open addressing-based hash table looks like this:

- Use the hash function to compute an initial index $i_{initial}$ for the element.
- If the hash table array at index $i_{initial}$ is empty, insert the element there and stop.
- Otherwise, compute the next index i in the probing sequence and repeat.

This process of searching for an empty position is called probing. There are many different probing schemes:

- Linear probing: $i = i_{initial} + j$ (where j = 1, 2, 3, …)
- Quadratic probing: $i = i_{initial} + j^2$ (where j = 1, 2, 3, …)
- Double hashing: $i = i_{initial} + j * h_2(key)$ (where j = 1, 2, 3, …)
  - Here, h_2 is a second, independent hash function.

For example, using linear probing in the scenario below, the key “beyonce” would be inserted at index 7, even though the hash function evaluates to 4 for that key:



Attempting to place a value into an occupied index 4 and probing one index at a time until 7 is found

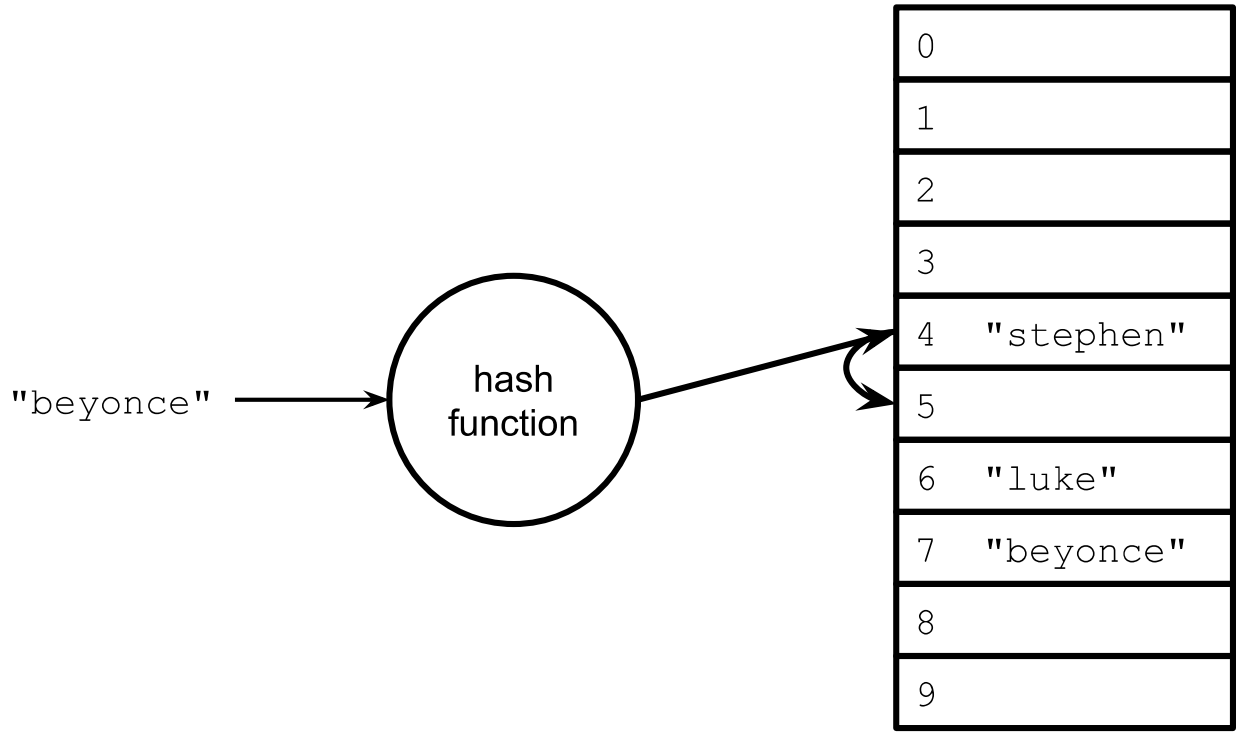
The procedure for searching for an element in an open addressing-based hash table is the same as inserting an element, except we probe until we find either the element we're looking for, or an empty spot, the latter of which means the element doesn't exist in the hash table.

What happens if we reach the end of the array while probing? **Simply wrap around to the beginning.** So, we can rewrite the probing schemes as below:

- Linear probing: $i = (i_{initial} + j) \% m$ (where, j = 1, 2, 3, …)
- Quadratic probing: $i = (i_{initial} + j^2) \% m$ (where, j = 1, 2, 3, …)
- Double hashing: $i = (i_{initial} + j * h_2(key)) \% m$ (where, j = 1, 2, 3, …)

Where m is the total number of spots in the hash table array.

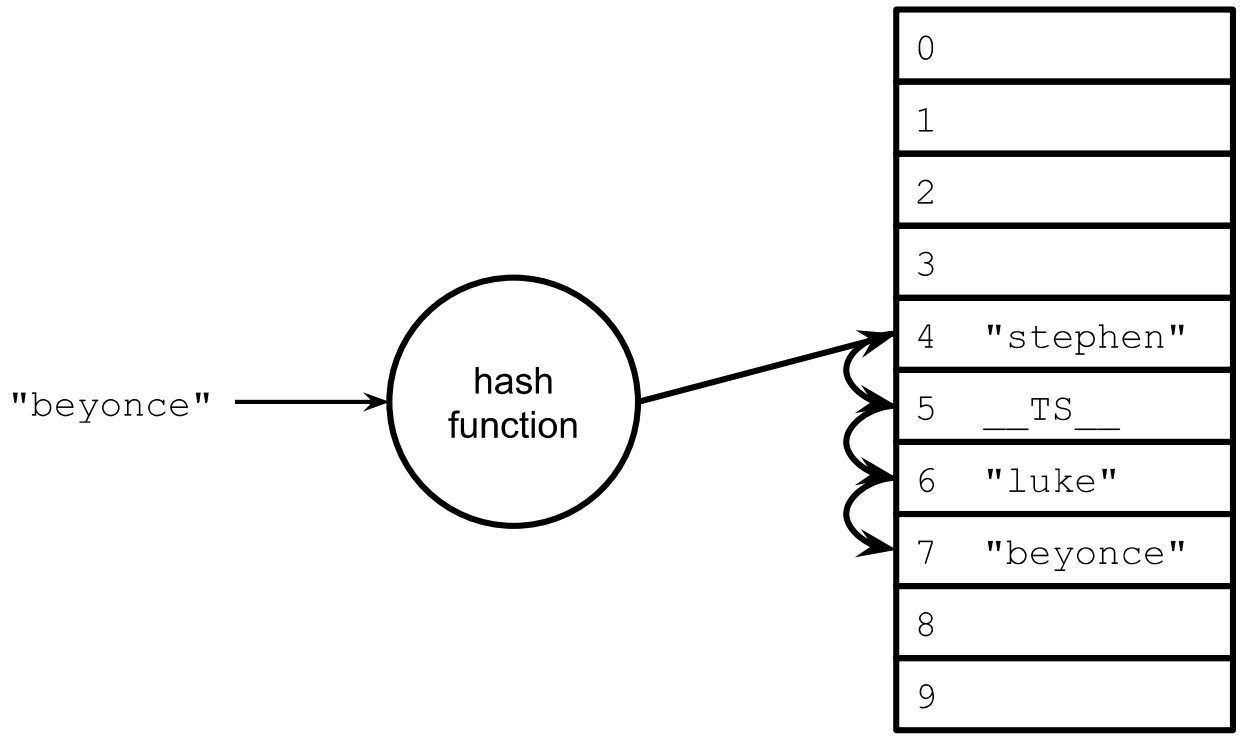
Now, what happens when we remove an element? This could disrupt probing for elements after it. For example, what if we removed "jon" in the example above and then searched for "beyonce"?



Encountering an empty index due to a removal

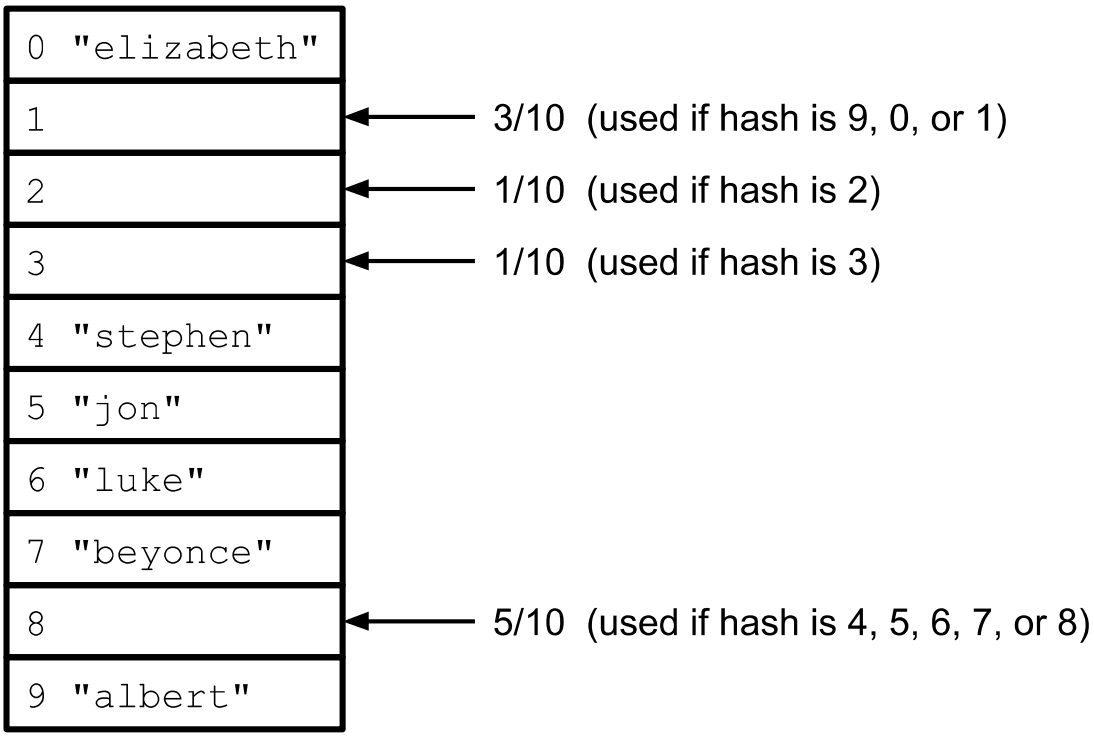
Without taking an extra measure, “beyonce” would not be found because the probe would end at the now-empty index 5. To handle this problem, we use a special value known as the tombstone. Now when an element is removed, we insert the tombstone value. This value can be replaced when adding a new entry, since if there is any value there, it won't halt the search for an element.

With a tombstone value *_TS_* inserted for the removed “jon”, the search above for “beyonce” could proceed as normal:



Probing skipping over the _TS_ entry and correctly finding the value it is looking for

One problem we can run into with open addressing is clustering, which is where elements are placed into the table in clusters of adjacent indices. For example, using linear probing, the probability of a new entry being added to an existing cluster increases as the size of the cluster increases, since a larger cluster yields more chances for a collision. The figure below illustrates the probability of a new element being placed into each of the open spots in the given hash table:



Probabilities of hitting an existing cluster based on cluster size

Using quadratic probing and, especially, double hashing, can help to reduce clustering in an open addressing scheme. When using open addressing, a table's load factor cannot exceed 1. Just as a dynamic array was doubled in size when necessary, a common solution to a full hash table is to move all values into a new and larger table when the load factor becomes greater than some threshold, such as 0.75. A new table is created, and every entry in the old table is rehashed, this time dividing by the new table size to find the index to use in the new table.

We want to maintain a low load factor so we can avoid collisions. However, when the load factor is very low, that means we have a lot of unused space allocated. In other words, *there is a tradeoff between speed and space with open addressing*.

Complexity Analysis of Open Address Hashing

Let's assume truly uniform hashing.

To insert a given item into the table (that's not already there), the probability that the first probe is successful is `(m-n)/m`.

- There are `m` total slots and `n` filled slots, so `m - n` open spots.
- Let's call this probability `p`, i.e., `p=(m-n)/m`.

If the first probe fails, the probability that the second probe succeeds is `(m-n)/(m-1)`

- There are still `m - n` remaining open slots, but now we only have a total of `m - 1` slots to look at, since we've examined one already.

If the first two probes fail, the probability that the third probe succeeds is `(m-n)/(m-2)`.

- There are still `m - n` remaining open slots, but now we only have a total of m - 2 slots to look at, since we've examined two already.

And so forth. In other words, for each probe, the probability of success is at least `p` because `(m-n)/(m-c) >= (m-n)/m = p`.

Here, we are dealing with a geometric distribution, so the expected number of probes until success is:

$1/p = 1/((m-n)/m) = 1/(1-n/m) = 1/(1-\lambda)$

In other words, the expected number of probes for any given operation is `O(1/(1-λ))`.

This suggests that, if we limit the load factor to a constant and reasonably small number, our operations will be O(1) on average. For example, if we have `λ = 0.75`, then we would expect 4 probes, on average. For `λ = 0.9`, we would expect 10 probes.