Maps and Hash Tables

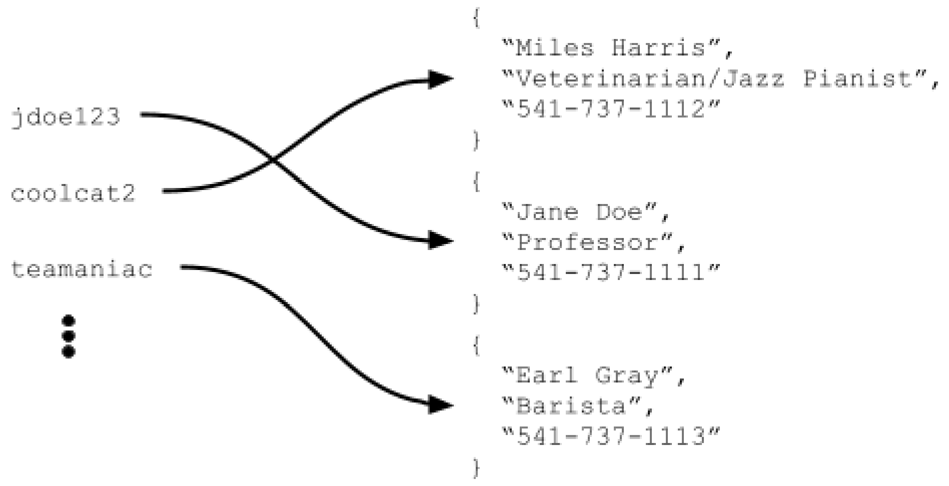
- Trees offer overall good performance on a wide variety of operations.

- Sometimes, though, all we really want to be able to do is add elements to a collection and look them up.

- When insertion and lookup (even removal) are the only operations we need, we can use the map data type.
    - A map is also known as a dictionary or an associative array.

- With a map, each data element is actually composed of two parts:
    - The ***key***, which is the value by which we look items up.
    - The ***value***, which is any and all other data associated with the element.

- For example, if we were building a web application, our user data might be represented in a map, where the key was username or email address and the value was all other data about each user (phone number, address, past purchases, contacts, etc.):


jdoe123
coolcat2
teamaniac
{ "Miles Harris", "Veterinarian/Jazz Pianist", "541-737-1112" }
{ "Jane Doe", "Professor", "541-737-1111" }
{ "Earl Gray", "Barista", "541-737-1113" }

- Or, if we were counting instances of all the words in a document, we could use a map where the key was the unique word and the value was the number of times that word occurred.

- We've seen data structures that would allow us to implement a map structure:
    - We could use a simple array, storing key/value structs.
        - This would give us *O(n)* insertions and lookups (or *O(log n)* lookups, if we ordered the array by key).
    - We could use an AVL tree, also storing key/value structs.
        - This would give us *O(log n)* insertions and lookups.

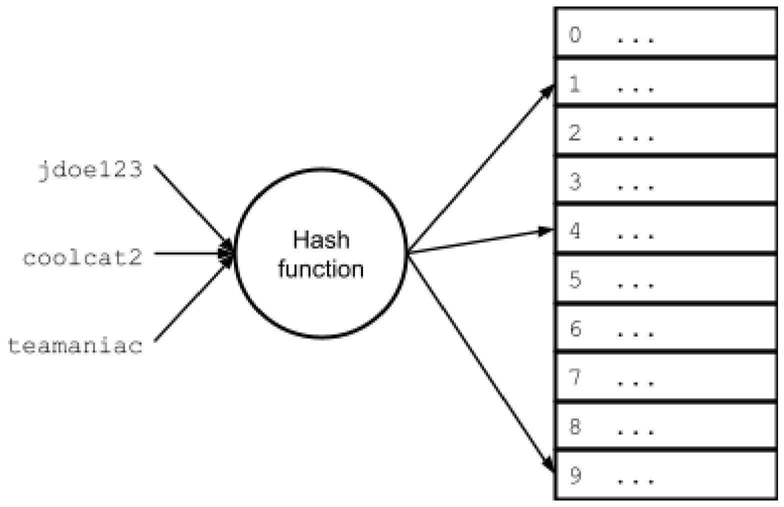
- We want to do better than this using a ***hash table***.

Hash tables

- A hash table is like an array, with a few important differences:
    - Elements can be indexed by values other than integers.
    - More than one element may share an index.

- The key to implementing a hash table is a ***hash function***, which is a function that takes values of some type (e.g. string, struct, double, etc.) and maps them to an integer index value.

- We can then use this value both to store and retrieve data out of an actual array, e.g.:



- Often the hash function computes an index in two steps:

```
hash = hash_function(key)
index = hash % array_size
```

- When choosing or designing a hash function, there are a few properties that are desirable:
    - ***Determinism*** – a given input should always map to the same hash value.
    - ***Uniformity*** – the inputs should be mapped as evenly as possible over the output range.
        - A non-uniform function can result in many ***collisions***, where multiple elements are hashed to the same array index. We'll look more at this later.
    - ***Speed*** – the function should have low computational burden.

- For example, if we were hashing strings, a simple hash function might sum the ASCII values of the characters, e.g.:

```
"eat" ➪ 'e' + 'a' + 't' = 101 + 97 + 116 = 314
```

  - An operation like this is known as a ***folding*** operation.

- This can have problems, though, e.g.:

```
"eat" ➪ 'e' + 'a' + 't' = 101 + 97 + 116 = 314
"ate" ➪ 'a' + 't' + 'e' = 97 + 116 + 101 = 314
"tea" ➪ 't' + 'e' + 'a' = 116 + 101 + 97 = 314
```

- For instances like this, we can use a ***shifting*** operation, which modifies the individual components of a folding operation based on their position.
    - E.g. multiply by 2^0, 2^1, 2^2, 2^3, …

```
"eat" ➪ 'e' + 'a' + 't' =
     1* 101 + 2* 97 + 4* 116 = 759
"ate" ➪ 'a' + 't' + 'e' =
     1* 97 + 2* 116 + 4* 101 = 733
"tea" ➪ 't' + 'e' + 'a' =
     1* 116 + 2* 101 + 3* 97 = 609
```

- Of course, these hash functions we've looked at are much simplified ones that we'd (hopefully) never use in real life.

- Here’s an example of a well-known and widely-used hash function, the DJB hash function (for strings)[1]:

```
unsigned long hash(unsigned char *str) {
  unsigned long hash = 5381;
  int c;
  while (c = *str++) {
    hash = ((hash << 5) + hash) + c;  // hash * 33 + c
  }
  return hash;
}
```

  - This function is simple and fast (though could be faster, e.g. by processing multiple bytes at a time).
  - It produces a good distribution.

Perfect and minimally perfect hash functions

- A ***perfect*** hash function is one that results in no collisions.

- A ***minimally perfect*** hash function is one that results in no collisions for a table size that exactly equals the number of elements.

- For example, consider this collection of strings:

```
"yummy"
"delicious"
"incredible"
"fantastic"
"exquisite"
"nonpareil"
```

- For this specific collection, the following function is minimally perfect:

```
int string_hash(char* str) {
     return (int)(str[0] - 'a') % 6;
}
```

[1] http://www.cse.yorku.ca/~oz/hash.html

- Specifically, we have all the values 0 through 5 covered:

```
string_hash("yummy") → 0        // 'y' - 'a' = 24
string_hash("delicious") → 3    // 'd' - 'a' = 3
string_hash("incredible") → 2   // 'i' - 'a' = 8
string_hash("fantastic") → 5    // 'f' - 'a' = 5
string_hash("exquisite") → 4    // 'e' - 'a' = 4
string_hash("nonpareil") → 1    // 'n' - 'a' = 13
```

- Of course, in practice, we don't usually have such a nicely arranged situation, so it's rare that our hash function will be minimally perfect.
    - For example, even with perfectly uniform random distribution of elements and a hash table with a capacity of 1 million elements, there is a 95% probability of a collision with only 2450 elements.

- This means that, most likely, we'll need to be able to deal with collisions, where the hash function maps more than one element to the same index in our hash array.

- We'll look at two mechanisms for resolving hash conflicts: chaining and open addressing.

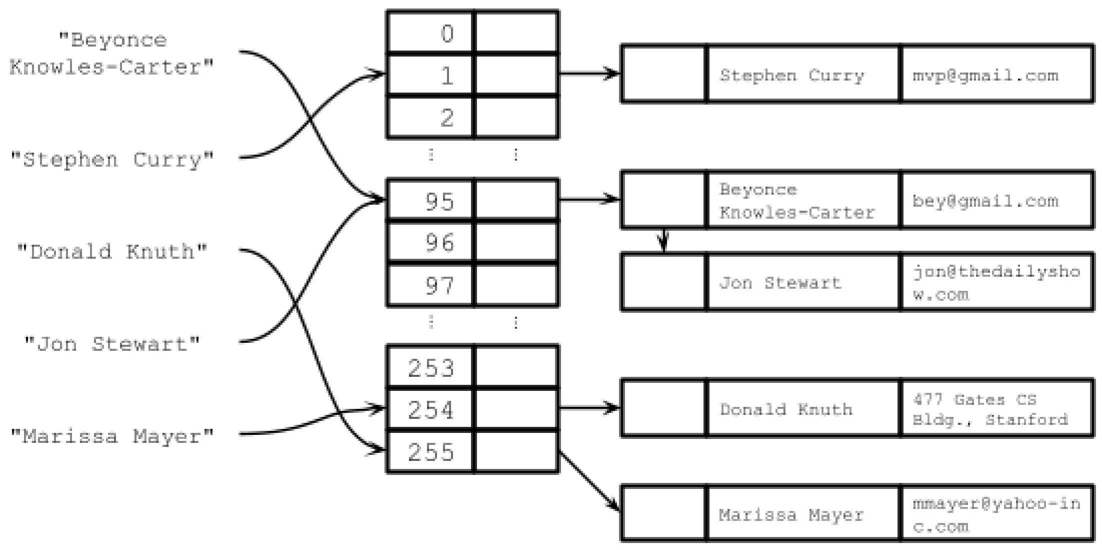
Collision resolution with chaining

- The ***chaining*** method involves storing a *collection* of elements at each index in the hash table array.
    - Each collection is called a ***bucket*** or a ***chain***.

- 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 themselves.
    - Other data structures could 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:



- 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 be similar, except we would add or remove the element to/from the appropriate bucket's data structure.

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

$$\lambda = \frac{n}{m}$$

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

- 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**.

- Thus, to maintain the performance of the 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?
        - Re-compute 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.:

$$\lambda = \frac{n}{m} = \frac{O(m)}{m} = O(1)$$

    - 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.

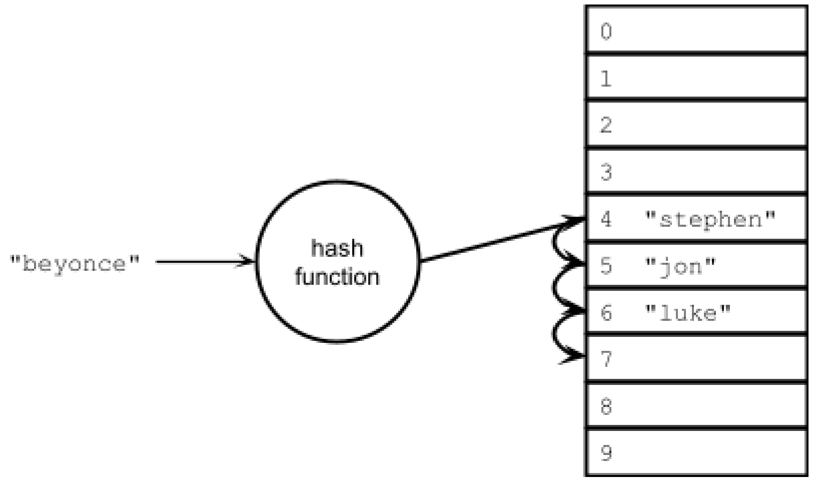
- When using open addressing, all hashed elements are stored directly in the hash table array.

- 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* for the element.

    - If the hash table array at index *i* is empty, insert the element there and stop.
    - Otherwise, increment *i* to the next index in the probing sequence (e.g. *i + 1*) and repeat.

- This process of searching for an empty position is called ***probing***.
    - There are many different probing schemes:
        - ***Linear probing*** – *i = i + 1*
        - ***Quadratic probing*** – $i = i + j^2$ *(j = 1, 2, 3, …)*
        - ***Double hashing*** – $i = i + j * h_2(key)$ *(j = 1, 2, 3, …)*
            - Here, h_2 is a second, independent hash function.

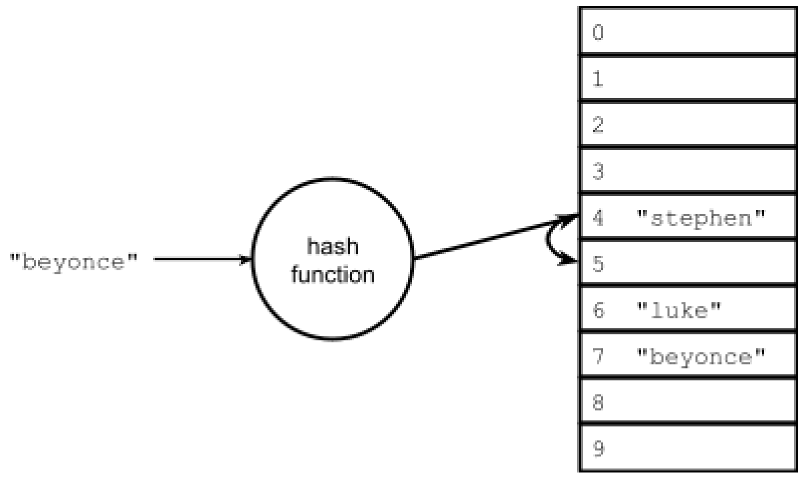
- For example, using linear probing in the below scenario, the key `"beyonce"` would be inserted at index 7, even though the hash function evaluates to 4 for that key:



- The procedure for searching for an element in an open addressing-based hash table is the same as for 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.

- What happens if we reach the end of the array while probing?
    - Simply wrap around to the beginning.

- 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"`?

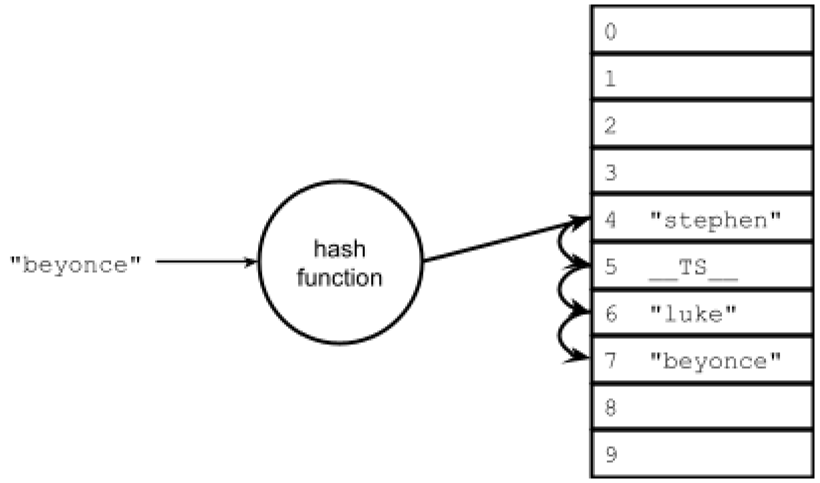


- Without taking an extra measures, `"beyonce"` would not be found because the probe would end at the now-empty index 5.

- To get around 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, but it doesn't halt search for an existing element.

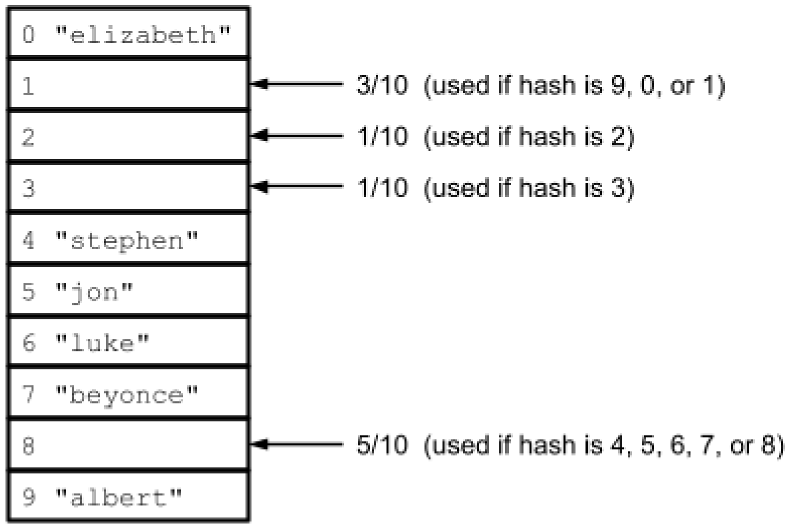
- With a tombstone value `__TS__` inserted for the removed `"jon",` the search above for `"beyonce"` could proceed as normal:



- One problem we can run into with open addressing is ***clustering***, where elements are placed into the table into 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 the 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 a given hash table:



- Using quadratic probing and especially double hashing can help to reduce clustering in an open addressing scheme.

- Using open addressing, a table's load factor cannot exceed 1.

- 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.

- How can we analyze the complexity of open addressing?
    - 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 $\frac{m-n}{m}$.
        - There are *m* total slots and *n* filled slots, so *m* - *n* open spots.
        - Let's call this probability *p, i.e.* $p = \frac{m-n}{m}$.

    - If the first probe fails, the probability that the second probe succeeds is $\frac{m-n}{m-1} \geq \frac{m-n}{m} = p$.
        - 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 $\frac{m-n}{m-2} \geq \frac{m-n}{m} = p$.
        - 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*.
    - Here, we are dealing with a *geometric distribution*[2], so the expected number of probes until success is:

$$\frac{1}{p} = \frac{1}{\frac{m-n}{m}} = \frac{1}{1-\frac{n}{m}} = \frac{1}{1-\lambda}$$

- In other words, the expected number of probes for any given operation is $O(\frac{1}{1-\lambda})$.

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

[2] https://en.wikipedia.org/wiki/Geometric_distribution