Introduction to Hash Tables

Step 1

We started this discussion about **hashing** with the motivation of achieving O(1) find, insert, and remove operations on average. We then discussed **hash functions**, which allow us to take any object of some arbitrary type and effectively "convert" it into an integer, which we can then use to index into an array. This hand-wavy description of the ability to index into an array is the basic idea of the **Hash Table** data structure, which we will now formally discuss.

DISCLAIMER: Whenever we say "O(1)" or "constant time" with regard to the average-case time complexity of a **Hash Table**, this is **ignoring the time complexity of the hash function**. For primitive data types, **hash functions** are constant time, but for types that are collections of other types (e.g. lists, strings, etc.), good **hash functions** iterate over *all* the elements in the collection. For example, a good **hash function** for strings of length k would iterate over all k characters in the string, making it O(k). Thus, mapping a string of length k to an index in an array is in reality O(k) overall: we first perform a O(k) operation to compute a hash value for the string, and we then perform a O(1) operation to map this hash value to an index in the array.

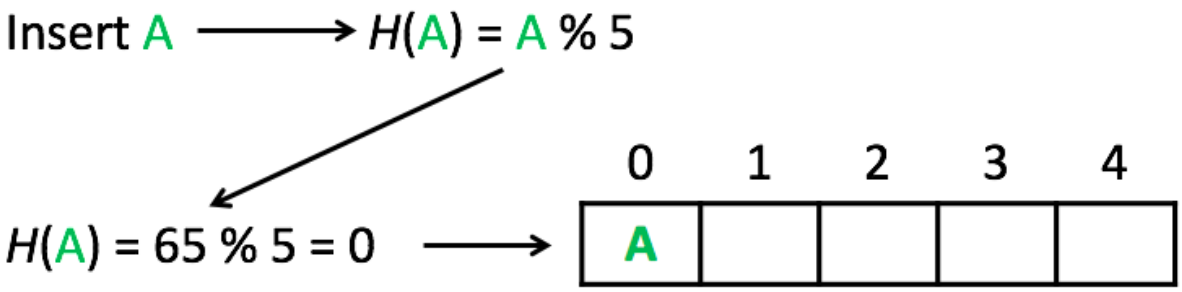
Nevertheless, people say that **Hash Tables** in general have an **average-case** time complexity of **O(1)** because, unlike any of the other generic data structures (i.e., data structures that can store *any* datatype) we have seen thus far, **the average-case performance of a Hash Table is *independent* of the number of elements it stores**. For this reason, from here on out, we will omit the time complexity of computing a hash value when describing the time complexity of a **Hash Table**, which is the norm in the realm of Computer Science. Nevertheless, please keep this disclaimer in mind whenever thinking about **Hash Tables**.

Step 2

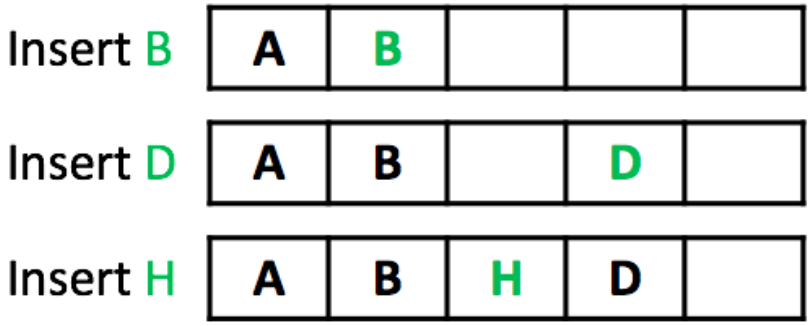
A **Hash Table** can be thought as a collection of items that, on average, can be retrieved really fast. A **Hash Table** is implemented by an array, and more formally, we define a **Hash Table** to be an array of size M (which we call the **Hash Table's** *capacity*) that stores keys k by using a **hash function** to compute an *index* in the array at which to store k.

For example, suppose we have a **Hash Table** with a capacity of $M = 5$ that stores letters as keys. We would choose a **hash function** that takes in the ASCII values of letters and maps them to indices (e.g. 'A' → index 0, 'B' → index 1, etc.). Since $M = 5$, we need to make sure that any characters we insert into the **Hash Table** will always map to a valid index (a number between 0 and 4, inclusive). As a result, we choose the **hash function** to be $H(k) = k \% M$.

Inserting the letter 'A' into our **Hash Table** would go through this process:



The rest of our insertions would look like this:

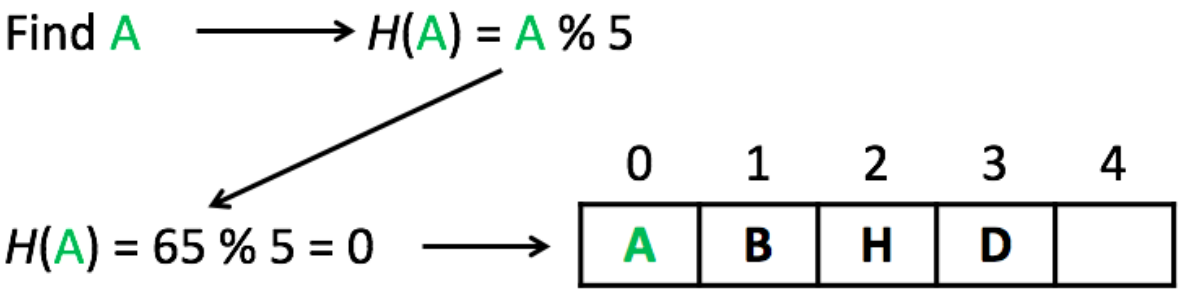


Note: Some of you may have noticed that there is redundancy in our mapping. For example, if we were to attempt to insert the letter 'A' and then the letter 'F', 'F' would have mapped to the slot already occupied by 'A'. More formally, we call this a *collision*. We will discuss how to handle collisions in-depth later on, but for now, just know that they are the cause of any slowness in a **Hash Table**, so we want to design our **Hash Table** in such a way that minimizes them.

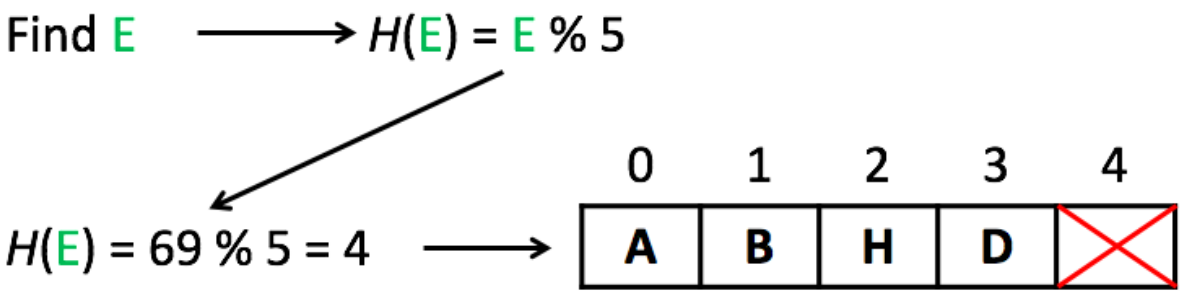
Step 3

Hopefully you found the previous step to be pretty cool: we have now found a way to take a key of *any* type and have it map to a valid index in an array, all thanks to our **hash function**.

Moreover, we are now able to find the key in constant time because we know exactly where we expect it to be, also because of our **hash function**! Finding the letter 'A' in our **Hash Table** (which should be successful) would go through the process below (note how similar it is to insert):



Similarly, attempting to find the letter 'E' in our **Hash Table** (which should fail) would be just as easy:



Note that we didn't have to resort to linearly searching or even using some optimized search algorithm such as binary search to find the key. Instead, we were able to find the key in constant time. It turns out that we got lucky with this simple example, but it turns out that we can formally prove that, if we design our **Hash Table** intelligently, the *average*-case time complexity to find an element is O(1) (we will later see why the *worst*-case time complexity is *not* constant-time).

Step 4

EXERCISE BREAK: Which index will the character 'a' hash to in the **Hash Table** below? Note:

- $M = 7$

- $H(k) = k \% M$, where we use key k's ASCII value in the mod operation
- We are using 0-based indexing

To solve this problem please visit https://stepik.org/lesson/30915/step/4

Step 5

In general, a **Hash Table** needs to have the following set of functions:

- **`insert(key)`**
- **`find(key)`**
- **`remove(key)`**
- **`hashFunction(key):`** to produce a hash value for a key to use to map to a valid index
- **`key_equality(key1, key2):`** to check if two keys *key1* and *key2* are equal

Below is pseudocode for the "insert" operation of a **Hash Table**. Note that this is a simplified version of the "insert" operation as we do not allow a key to be inserted if another key is already at its index (i.e., if we have a *collision*). We will discuss later how to handle collisions, but for now, we will disallow them. In the pseudocode below, `arr` is the array that is backing the **Hash Table**, and `H` is the **hash function** that maps `key` to a valid index.

```
insert(key): // Insert key into the Hash Table
    index = H(key)
    if arr[index] is empty:
        arr[index] = key
```

Below is pseudocode for the "find" operation of a **Hash Table**. Note that this "find" operation is also simplified, as it is based upon the simplified premise of the "insert" function above (i.e., no collisions allowed). In the pseudocode below, `arr` is the array that is backing the **Hash Table**, and `H` is the **hash function** that maps `key` to a valid index.

```
find(key): // Return true if key exists in the Hash Table, otherwise return False
    index = hashFunction(key)
    return arr[index] == key
```

Step 6

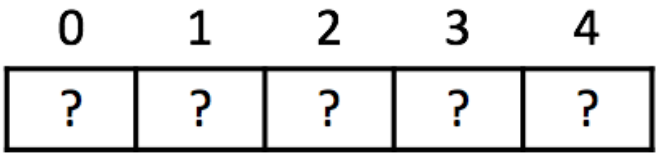
EXERCISE BREAK: Fill in the missing elements in the following **Hash Table** after executing all the operations below in the order they appear. Note:

- $M = 5$
- Key k is of type `string`
- The insert algorithm you will use is the one mentioned previously, in which we don't allow insertion if there is a collision
- $H(k)$ is the C++ function below:

```
int H(string k) {
    char letter = k.at(0); // get the character at index 0 of k
    return letter % M;
}
```

OPERATIONS:

1. Insert "Orange"
2. Insert "Apple"
3. Insert "Cranberry"
4. Insert "Banana"
5. Insert "Strawberry"

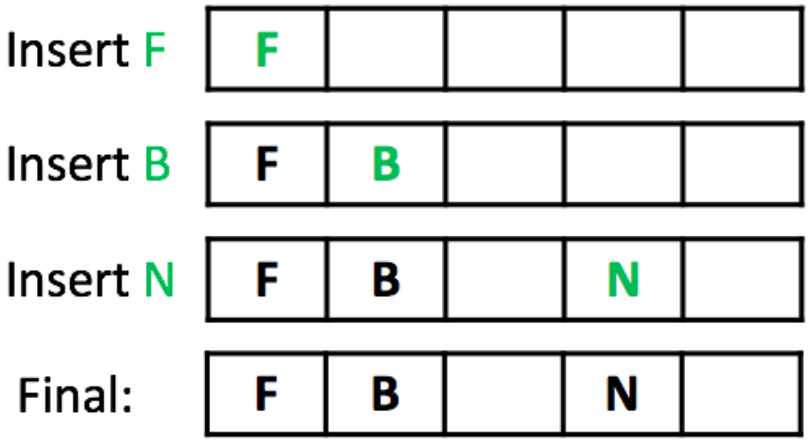


To solve this problem please visit https://stepik.org/lesson/30915/step/6

Step 7

So far, we have seen that inserting and finding an item can be done in constant time (under the assumptions we made earlier), which is extremely useful. However, what if we wanted to iterate over the items we have inserted in the **Hash Table** in sorted order?

STOP and Think: Can you think of a good (i.e., efficient) way to to print out the keys of the Final **Hash Table** below in alphabetical order, where the **hash function** is defined to be H(*k*) = *k* % M?

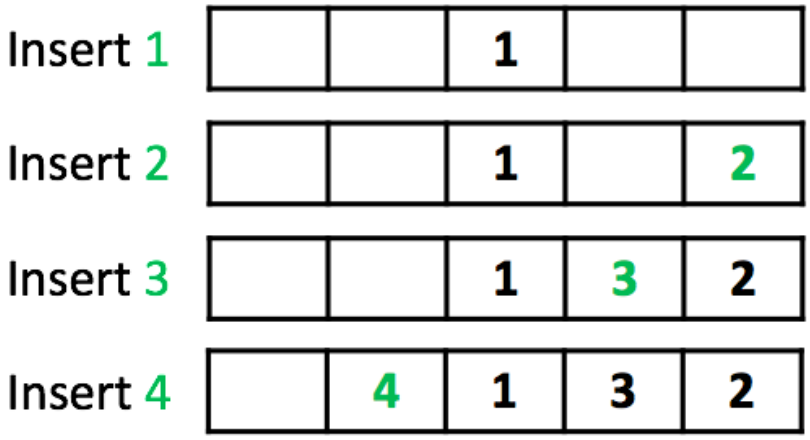


Step 8

Hopefully you didn't spend *too* much time on the **STOP and Think** in the previous step, because the answer is that there really is unfortunately *no* good way of printing the elements of a **Hash Table** in sorted order. Why? Because keys are **not** sorted in this data structure! This is very different from a lot of the data structures that we have discussed so far (such as Binary Search Trees) in which we maintained an ordering property at all times.

So, in the example given in the previous step, if we were to iterate through the keys in our **Hash Table**, we would expect the output to be "F B N."

What if we were to insert the numbers 1-4 in an arbitrary **Hash Table**? Could we then expect the keys to be in sorted order? The answer is no! Why? Because we have no clue exactly what hash value the underlying **hash function** is producing in order to map the key to an index. For example, if $H(k) = 2^k \% M$, then we would expect to see the following:



You might be wondering why we would even be using a **hash function** as complicated as $H(k) = 2^k \% M$, and you might suspect that we're just being unrealistic for the sake of argument. However, it turns out that using a **hash function** more complicated than just modding by M to perform the index mapping helps randomize the indices to which keys map, which helps us reduce collisions on average.

Step 9

In C++, a **Hash Table** is called an `unordered_set`. The name should be pretty intuitive because we are storing a *set* of keys, and the set is *unordered*. Below is example C++ code to actually use a **Hash Table**:

```
#include<unordered_set> // include the C++ implementation of a Hash Table
#include<string>
#include<iostream>
using namespace std;

int main() {
    unordered_set<string> animals = {"Giraffe", "Polar Bear", "Toucan"};
}
```

If we wanted to easily iterate over the keys we stored, we could use an iterator. For example, we could add this to the end of `main()`:

```
for(auto it = animals.begin(); it != animals.end(); ++it) {
    cout << *it << endl;
}

/* Output:
Toucan
Giraffe
Polar Bear
*/
```

Equivalently, we could have used a for-each loop. For example, we could add this to the end of `main()`:

```
for(auto s : animals) {
    cout << s << endl;
}

/* Output:
Toucan
Giraffe
Polar Bear
*/
```

Notice that the output above is unsorted, just as we expected! If you were to iterate over the elements of a **Hash Table** and they *happened* to be iterated over in sorted order, know that it is purely by chance. There is absolutely no guarantee that the elements will be sorted, and on the contrary, because of the fancy **hash functions** used to help minimize collisions, they will actually most likely not be.

Note: If we wanted to insert a key that was an instance of a custom class, we would need to overload C++'s **hash function** and **key equality** methods in order to have the `unordered_set` be able to properly **hash** the custom class.

Step 10

So far, all of our insertions have been working smoothly in **constant time** because we have avoided discussing what happens when a key indexes to a slot that is already occupied (i.e., when we encounter a *collision*). However, this is arguably the most important part of implementing an optimized **Hash Table.** Why? More often than not, we are mapping from a large space of possible keys (e.g. the numbers 1 to 1,000,000 for a **Hash Table** storing yearly household incomes, a large number of strings representing every possible human name for a **Hash Table** storing Google employee information, etc.) to a much smaller space: the slots of the array that is backing the **Hash Table**. Since computers do not have unlimited memory, a **Hash Table's** capacity is in fact *much much* smaller than the vast set of possible keys. As a result, we will inevitably encounter *collisions*: two different keys indexing to the same **Hash Table** location.

Earlier in this lesson, our pseudocode for the insert operation of a **Hash Table** simply refused to insert a key if a collision were to occur. In practice, however, we would be facing a terrible data structure if we couldn't insert all the values we wanted. As a result, we need to invest time into thinking about the best possible ways to *avoid* collisions in the first place, as well as *what to do* if a collision were to occur. For the time being, we'll keep the topic of *resolving* collisions on hold just a bit longer, but in the next lesson, we will use probability theory to discuss the causes of collisions in more depth, and we will think about how to choose an optimal **capacity** and a good indexing **hash function** for a **Hash Table** to help us *avoid* collisions.