Using Hash Tables and Hash Maps

Step 1

The fourth Data Structure we will discuss for implementation of a lexicon is the **Hash Table**. To refresh your memory, below are some important details regarding **Hash Tables** and **Hash Maps** that we can use to implement a lexicon:

- To insert an element into a **Hash Table**, we use a **hash function** to compute the **hash value** (an integer) representing the element. We then mod the **hash value** by the size of the **Hash Table's** backing array to get an index for our element
- Ignoring the time it takes to compute hash values, the **average-case time complexity** of finding, inserting, and removing elements in a well-implemented **Hash Table** is **O(1)**
- The time complexity to compute the **hash value** of a string of length *k* (assuming our **hash function** is good) is **O(*k*)**
- A **Hash Map** is effectively just an extension of a **Hash Table**, where we do all of the **Hash Table** operations on the *key*, but when we actually perform the insertion, we perform the (*key*, *value*) pair. For our purposes of making a lexicon, it could make sense to have *keys* be words and *values* be definitions
- It is impossible for us to iterate through the elements of an *arbitrary* **Hash Table** in a meaningful order.

Now that we have reviewed the properties of the **Hash Table**, we can begin to discuss how to actually use it to implement the three lexicon functions we previously described.

Step 2

Below is pseudocode to implement the three operations of a lexicon using a **Hash Table**. In all three functions below, the backing **Hash Table** is denoted as `hashTable`.

```
find(word):   // Lexicon's "find" function
    return hashTable.find(word) // call the backing Hash Table's "find" function
```

```
insert(word): // Lexicon's "insert" function
    hashTable.insert(word)      // call the backing Hash Table's "insert" function
```

```
remove(word): // Lexicon's "remove" function
    hashTable.remove(word)      // call the backing Hash Table's "remove" function
```

STOP and Think: Does our choice in collision resolution strategy have any effect on the performance of our lexicon?

Step 3

EXERCISE BREAK: What is the **worst-case** time complexity of the **`find`** function defined in the previous pseudocode, ignoring the time it takes to compute an element's hash value?

Step 4

EXERCISE BREAK: What is the **average-case** time complexity of the **`find`** function defined in the previous pseudocode, ignoring the time it takes to compute an element's hash value?

To solve this problem please visit https://stepik.org/lesson/31308/step/4

Step 5

EXERCISE BREAK: What is the **worst-case** time complexity of the **`insert`** function defined in the previous pseudocode, ignoring the time it takes to compute an element's hash value?

To solve this problem please visit https://stepik.org/lesson/31308/step/5

Step 6

EXERCISE BREAK: What is the **average-case** time complexity of the **`insert`** function defined in the previous pseudocode, ignoring the time it takes to compute an element's hash value?

To solve this problem please visit https://stepik.org/lesson/31308/step/6

Step 7

EXERCISE BREAK: What is the **worst-case** time complexity of the **`remove`** function defined in the previous pseudocode, ignoring the time it takes to compute an element's hash value?

To solve this problem please visit https://stepik.org/lesson/31308/step/7

Step 8

EXERCISE BREAK: What is the **average-case** time complexity of the **`remove`** function defined in the previous pseudocode, ignoring the time it takes to compute an element's hash value?

To solve this problem please visit https://stepik.org/lesson/31308/step/8

Step 9

CODE CHALLENGE: Implementing a Lexicon Using a Hash Table

In C++, the `unordered_set` container is implemented as a **Hash Table**. Below is the C++ `Lexicon` class we have declared for you:

```
class Lexicon {
    public:
        unordered_set<string> hashTable; // instance variable Hash Table object
        bool find(string word);          // "find" function of Lexicon class
        void insert(string word);        // "insert" function of Lexicon class
        void remove(string word);        // "remove" function of Lexicon class
};
```

If you need help using the C++ `unordered_set`, be sure to look at the C++ Reference.

Sample Input:

```
N/A
```

Sample Output:

```
N/A
```

To solve this problem please visit https://stepik.org/lesson/31308/step/9

Step 10

CODE CHALLENGE: Implementing a Dictionary Using a Hash Map

Recall that the only difference between a *lexicon* and a *dictionary* is that a *lexicon* is just a list of words, whereas a *dictionary* is a list of words *with their definitions*. In C++, the `unordered_map` container is implemented as a **Hash Map**. Below is the C++ `Dictionary` class we have declared for you, where *keys* are words and *values* are definitions:

```
class Dictionary {
    public:
        unordered_map<string,string> hashMap; // instance variable Hash Map object
        string find(string word);             // "find" function of Dictionary class
        void insert(string word, string def); // "insert" function of Dictionary class
        void remove(string word);             // "remove" function of Dictionary class
};
```

If you need help using the C++ `unordered_map`, be sure to look at the C++ Reference.

Sample Input:

```
N/A
```

Sample Output:

```
N/A
```

To solve this problem please visit https://stepik.org/lesson/31308/step/10

Step 11

As you can see, we further improved our **average-case** performance by using a **Hash Table** to implement our lexicon. We have to perform an **O(*k*)** operation on each word in our lexicon (where *k* is the length of the word) in order to compute the word's hash value, so we say that our **average-case time complexity** is **O(1)** if we don't take into account the hash function or **O(*k*)**.

However, keep in mind that the order in which elements are stored in a **Hash Table's** backing array does not necessarily have any meaning. In other words, it is impossible for us to iterate through the elements of an *arbitrary* **Hash Table** in a meaningful (e.g. alphabetical) order.

In terms of memory efficiency, recall that, with a **Hash Table**, we try to maintain the **load factor** (i.e., the percentage full the backing array is at any given moment) relatively low. Specifically, ~70% is typically a good number. Formally, because we will have allocated approximately $m = \frac{n}{0.7}$ slots to store *n* elements and because $\frac{1}{0.7}$ is a constant, our **space complexity** is **O(*n*)**. However, even though the space usage scales linearly, note that we will always be wasting approximately 30% of the space we allocate, which can be a pretty big cost as our dataset gets large (30% of a big number is a smaller, but still big, number).

Also, note that, by simply using a **Hash Map** instead of a **Hash Table**, we were able to create a *dictionary* that stores (*word*, *definition*) pairs as opposed to a *lexicon* that only stores *words*. As a result, we have the added functionality of being able to find a word's definition in addition to simply seeing if a word exists, but without worsening our performance. It should be noted that we could have theoretically made somewhat trivial modifications to any of the previous (and to any of the upcoming) implementation strategies to store a *dictionary* instead of a *lexicon* (i.e., simply store (*word*, *definition*) pairs in our data structure and performing all searching algorithms using just the *word*), but we would have to overload various C++ specific functions—such as the comparison function—to be able to use existing C++ data structure implementations.

Because we have to compute the hash values of our strings, we can safely say that the time complexity of this implementation is **O(*k*)** in the ***average* case**. Is there some way to make this O(*k*) time complexity a *worst*-case time complexity? Also, can we somehow gain the ability to iterate through our elements in alphabetical order? In the next section, we will discuss the **Multiway Trie**, a data structure that allows us to obtain both.