Hash Maps

Step 1

Thus far in the text, we have only discussed *storing keys* in a data structure, and we have discussed numerous data structures that can be used to find, insert, and remove keys (as well as their respective trade-offs). Then, throughout this chapter, we introduced the **Hash Table**, a data structure that, on average, performs *extremely fast* (i.e., constant-time) find, insert, and remove operations.

However, what if we wanted to push further than simply storing *keys*? For example, if we were to teach a class and wanted to store the students' names, we could represent them as strings and store them in a **Hash Table**, which would allow us to see if a student is enrolled in our class in constant time. However, what if we wanted to *also* store the students' grades? In other words, we can already query a student's name against our **Hash Table** and receive "true" or "false" if the student is in our table or not, but what if we want to instead return the student's grade?

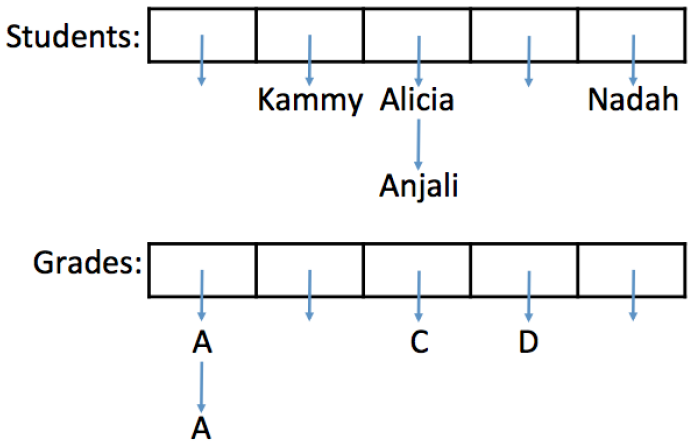
The functionality we have described, in which we query a *key* and receive a *value*, is defined in the **Map ADT**, which we will formally describe very soon. After discussing the **Map ADT**, we will introduce a data structure that utilizes the techniques we used in **Hash Tables** to *implement* the **Map ADT**: the **Hash Map**.

Sad Fact: Most people do not know the difference between a **Hash Table** and a **Hash Map** and as a result they use the terms interchangeably, so beware! As you will learn, though a **Hash Map** is built on the same *premise* as a **Hash Table**, it is a bit different and arguably more convenient in day-to-day programming.

Note: If you have experience programming in Python and have used a Python dictionary, then congratulations! You have been using a **Hash Map** all along and are already one step (no pun intended) ahead of us! If you have no clue what we are talking about, then read on!

Step 2

As we mentioned in the previous step, our goal is to have some efficient way to store students and their grades such that we could query our data structure with a student's name and then have it return to us their grade. We could, of course, store the student names and grades in separate **Hash Tables**, but how would we know which student has which grade?



Consequently, we want to find a way to be able to store a student's name *with* his or her letter grade. This is where the **Map Abstract Data Type** comes into play! The **Map ADT** allows us to *map* keys to their corresponding values. The **Map ADT** is often called an *associative array* because it gives us the benefit of being able to *associatively* cluster our data.

Formally, the **Map ADT**is defined by the following set of functions:

- **`put(<key,value>)`:** perform the insertion, and return the previous *value* if overwriting, otherwise null

- **`has(key)`:** return true if *key* is in the **Map**, otherwise return false
- **`remove(key)`:** remove the (*key*, *value*) pair associated with key, and return *value* upon success or null on failure
- **`size()`:** return the number of (*key*, *value*) pairs currently stored in the **Map**
- **`isEmpty()`:** return true if the **Map** does not contain any (*key*, *value*) pairs, otherwise return false

We have now formally defined the **Map ADT**, but how can we go about actually implementing it?

Step 3

The **Map ADT** can theoretically be implemented in a multitude of ways. For example, we could implement it as a **Binary Search Tree**: we would store *two* items inside each node, the *key* and the *value*, but we would keep the **Binary Search Tree** ordering property based on just *keys*.

However, if we didn't care so much about the sorting property but rather wanted faster *put* and *has* operations (if we desired a constant time-complexity, for example), then the **Map ADT** could also be implemented effectively as a **Hash Table**: we refer to this implementation as a **Hash Map**.

Implementation-wise, a **Hash Map** has the following set of operations:

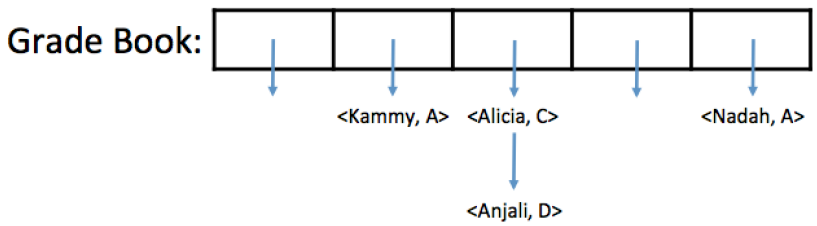
- **`insert(<key,value>)`:** perform the insertion, and return the previous *value* if overwriting, otherwise null
- **`find(key)`:** return the *value* associated with the *key*
- **`remove(key)`:** remove the (*key*, *value*) pair associated with key, and return *value* upon success or null on failure
- **`hashFunction(key)`:** return a hash value for *key*, which will then be used to map to an index of the backing array
- **`key_equality(key1, key2)`:** return true if *key1* is equal to *key2*, otherwise return false
- **`size()`:** return the number of (*key*, *value*) pairs currently stored in the **Hash Map**
- **`isEmpty()`:** return true if the **Hash Map** does not contain any (*key*, *value*) pairs, otherwise return false

Just like a **Hash Table**, a **Hash Map** uses a **hash function** for the purpose of being able to access the addresses of the tuples inserted. Consequently, in a **Hash Map**, keys must be hashable and have an associated equality test to be able to check for uniqueness. In other words, to use a custom class type as a key, one would have to overload the hash and equality member functions.

Step 4

When we find, insert, or remove (*key*, *value*) pairs in a **Hash Map**, we do *everything exactly like* we did with a **Hash Table**, but with respect to the *key*.

For example, in the **Hash Map insertion** algorithm, we are given a (*key*, *value*) pair, and we perform the entire insertion operation *identically* to the **Hash Table** insertion algorithm, except when we actually place the new object into the backing array, in a **Hash Table**, we just store the *key*, but in a **Hash Map**, we store the *key* and the *value* together. Below is an example of a **Hash Map** that contains multiple (*key*, *value*) pairs:



When we want to **find** elements, we perform the exact same "find" algorithm as we did with a **Hash Table**, but again with respect to the *key* (which is why our "find" function only had *key* as a parameter, not *value*), and once we find the (*key*, *value*) pair, we simply return the *value*. For example, in the example **Hash Map** above, if we were to perform the "find" algorithm on "Kammy," we would perform the

regular **Hash Table** "find" algorithm on "Kammy," and when we find the pair that has "Kammy" as its *key*, we would return the *value* (in this case, 'A').

Just like with finding elements, if want to **remove** elements from our **Hash Map**, we perform the **Hash Table** "remove" algorithm with respect to the *key* (which is why our "remove" function only had *key* as a parameter, not *value*), and once we find the (*key*, *value*) pair, we simply remove the pair.

Step 5

In case the previous step was too "hand-wavy" with regard to how we go about inserting elements, let's look at the pseudocode for the **Hash Map** operations below. Note that a **Hash Map** can be implemented using a **Hash Table** with any of the collision resolution strategies we discussed previously in this chapter. In all of the following pseudocode, the **Hash Map** is backed by an array `arr`, and for a (*key*, *value*) pair `pair`, `pair.key` indicates the *key* of `pair` and `pair.value` indicates the *value* of `pair`.

In the **insert** operation's pseudocode below, we ignore collisions (i.e., each key maps to a unique index in the backing array) because the actual insertion algorithm would depend on which collision resolution strategy you choose to implement.

```
insert(key,value): // insert <key,value>, replacing old value with new value if key exists
    index = hashFunction(key)
    returnVal = NULL

    // if key already exists, save the old value
    if arr[index].key == key:
        returnVal = arr[index].value // we want to return the old value instead of NULL

    // perform the insertion
    arr[index] = <key,value>
    return returnVal
```

With respect to insertion, originally, in a **Hash Table**, if a key that was being inserted already existed, we would abort the insertion. In a **Hash Map**, however, attempting to insert a key that already exists will *not* abort the insertion. Instead, it will result in the original value being overwritten by the new one.

The pseudocode for the **find** operation of a **Hash Map** is provided below. Note that this "find" algorithm returns the *value* associated with *key*, as opposed to a Boolean value as it did in the **Hash Table** implementation.

```
find(key): // return value associated with key if key exists, otherwise return NULL
    index = hashFunction(key)
    if arr[index].key == key:
        return arr[index].value
    else:
        return NULL
```

The pseudocode for the **remove** operation of a **Hash Map** is provided below. Just like with the pseudocode for the insertion algorithm above, in the pseudocode below, we ignore collisions (i.e., each key maps to a unique index in the backing array) because the actual remove algorithm would depend on which collision resolution strategy you choose to implement.

```
remove(key): // remove <key,value> if key exists and return value, otherwise return NULL
    index = hashFunction(key)
    returnVal = NULL

    // if key already exists, save the old value
    if arr[index].key == key:
        returnVal = arr[index].value // we want to return the value instead of NULL

    // perform the removal
    delete arr[index]
```

Step 6

In practice, however, we realize that you will more often than not be using the built-in implementation of a **Hash Map** as opposed to implementing it from scratch, so how do we use C++'s **Hash Map** implementation?

In C++, the implementation of a **Hash Map** is the unordered_*map*, and it is implemented using the **Separate Chaining** collision resolution strategy. Just to remind you, in C++, the implementation of a **Hash Table** is the unordered_*set*.

Going all the way back to the initial goal of implementing a grade book system, the C++ code to use a **Hash Map** would be the following:

```
unordered_map<string, string> gradeBook = {
                { "Kammy", "A"},
                { "Alicia", "C"},
                { "Anjali", "D"},
                { "Nadah", "A"}
 };
```

If we wanted to add a new student to our grade bookk, we would do the following:

```
gradeBook.insert({"Bob", "B"});

/* Our new hash map would look something like this:
             {  { "Kammy", "A"},
                { "Bob" , "B"},
                { "Alicia", "C"},
                { "Anjali", "D"},
                { "Nadah", "A"}  };

   Note how there is no ordering property, as expected */
```

If we wanted to check Nadah's grade in our grade book, we would do the following:

```
cout << gradeBook["Nadah"] << endl; // [] operator returns the value stored at the key

/* Output:
   A
*/
```

Although we have mentioned many times that there is no particular ordering property when it comes to a **Hash Map** (as well as a **Hash Table**), we can still *iterate* through the inserted objects using a for-each loop like so:

```
for (auto student : gradeBook) {

   std::cout << student.first << ": " << student.second << std::endl; // .first returns the key
                                                                      // .second returns the value
}

/* Output:
   Bob: B
   Kammy: A
   Anjali: D
   Alicia: C
   Nadah: A
*/
```

Step 7

EXERCISE BREAK: Which of the following statements regarding the code below are true?

```
unordered_map<string, string> groceryList = {
                { "Fruit", "Bananas"},
                { "Cereal", "Frosted Flakes"},
                { "Vegetable", "Cucumbers"},
                { "Juice", "Cranberry"}
 };

groceryList.insert({"Ice Cream", "Chocolate Chip"});
groceryList.insert({"Juice", "Carrot"});
```

To solve this problem please visit https://stepik.org/lesson/31220/step/7

Step 8

It is also important to note that, in practice, we often use a **Hash Map** to implement "one-to-many" relationships. For example, suppose we want to implement an "office desk" system in which each desk drawer has a different label: "pens", "pencils", "personal papers", "official documents", etc. Inside each particular drawer, we expect to find office items related to the label. In the "pens" drawer, we might expect to find our favorite black fountain pen, a red pen for correcting documents, and that pen we "borrowed" from our friend months ago.

How would we use a **Hash Map** to implement this system? Well, the *drawer labels* would be considered the *keys*, and the *drawers* with the objects inside them would be considered the corresponding *values*.

The C++ code to implement this system would be the following:

```
unordered_map<string, vector<OfficeSupply>> desk = {
                { "pens", {favPen, redPen, stolenPen} },
                { "personal papers", {personalNote}   }
};
```

In the **Hash Map** above, we are using *keys* of type `string` and *values* of type `vector<OfficeSupply>` (where `OfficeSupply` is a custom class we created). Note that the *values* inserted into the **Hash Map** are **NOT** `OfficeSupply` objects, but `vectors` of `OfficeSupply` objects.

If we now wanted to add a printed schedule to the "personal papers" drawer of our desk, we would do the following:

```
desk["personal papers"].push_back(schedule);

/*
   desk["personal papers"] returns the {personalNote} vector
   push_back adds a schedule object of type OfficeSupply to the {personalNote} vector

   our hash map now looks like this:
   {
       { "pens", {favPen, redPen, freePen} },
       { "personal papers", {personalNote, schedule} }
   }
*/
```

Note: If we wanted to calculate the worst-case time complexity of finding an office supply in our desk, we would now need to take into account the time it takes to find an element in an unsorted `vector` (which is an Array List) containing *n* `OfficeSupply` objects, which would be O(*n*) in itself, *not* including the time it takes to find the correct `vector` in our **Hash Map** in the first place. If we wanted to ensure constant-time access across `OfficeSupply` objects, we could also use a **Hash Table** instead of an Array List (yes, we are saying that you can use `unordered_set` objects as *values* in your `unordered_map`).

To easily output which pens we have in our desk, we could use a for-each loop:

```
for (auto pen : desk["pens"]) {
   cout << pen <<std::endl;
}
/* Output:
   favPen
   redPen
   freePen
*/
```

Step 9

EXERCISE BREAK: Which of the following statements regarding the **Hash Map** data structure are true? (Select all that apply)

To solve this problem please visit https://stepik.org/lesson/31220/step/9

Step 10

We began this chapter with the motivation of obtaining *even faster* find, insert, and remove operations than we had seen earlier in the text, which led us to the **Hash Table**, a data structure with **O(1)** find, insert, and remove operations in the **average case**. In the process of learning about the **Hash Table**, we discussed various properties and design choices (both in the **Hash Table** itself as well as in **hash functions** for objects we may want to store) that can help ensure that we actually experience the constant-time performance on average.

We then decided we wanted even *more* than simply *storing* elements: we decided we wanted to be able to *map* objects to other objects (map *keys* to *values*, specifically), which led us to the **Map ADT**. Using our prior knowledge of **Hash Tables**, we progressed to the **Hash Map**, an extremely fast implementation of the **Map ADT**.

In practice, the **Hash Table** and the **Hash Map** are probably the two most useful data structures you will encounter in daily programming: the **Hash Table** allows us to store our data and the **Hash Map** allows us to easily cluster our data, both with great performance.