Lecture 16: HashTables
10:00 AM, Mar 2, 2020

Contents

1 Speeding up Lookup 1

2 HashTables 2
   2.1 Java HashMaps ...................................................... 2
   2.2 Any Type Can Serve as Key ........................................... 3

3 Hash Functions 4
   3.1 Writing hashCode methods ........................................... 5
   3.2 Collisions ............................................................... 5

4 Building HashTables for Ourselves 6

Motivating Question

Arrays give us constant-time lookup on data based on array indices. Can we get near-constant time lookup for data that is indexed on something other than array positions, such as strings?

Objectives

By the end of these notes, you will know:

- What hash tables are
- About hash functions and collisions, which are key concepts associated with hash tables

By the end of these notes, you will be able to:

- Use Java’s HashMap class

1 Speeding up Lookup

Imagine that we are tasked with creating a campus directory that allows people to quickly locate faculty offices based on faculty names. We could make some sort of list of objects that store names and locations, but that will involve some sort of searching (which at best can be $O(\log n)$ for $n$ faculty.
If we had the locations in an array, and a fast way to map faculty names to array indices, we could do this in constant time. But how might we quickly map names to array indices?

Enter hashtables. Hashtables are a data structure that solves precisely this problem: they map keys (such as names) to values (such as office locations), using arrays under the hood for time efficiency. This lecture introduces hashtables and begins to show how they work under the hood.

# Hashtables

## Java HashMaps

To see hashtables in action, let’s do an example using Java’s version, which are called hashmaps. We’ll present this as a sequence of lines of code within a main method to illustrate the methods and core behaviors of HashMaps.

For our example, we’ll start simpler than customers and instead map names of Professors to their offices. For example, ”Kathi” should map to 309, and ”Tim” to 355. To get started, we import the hashmap data structure and define a hashmap called offices (we’re putting this inside a main method so we can use println to easily see how the hashmap behaves):

```java
import java.util.HashMap;

public class HTDemo {
    public static void main(String[] args) {
        HashMap<String, Integer> offices = new HashMap<String, Integer>();
    }
}
```

When we create a hashmap, we provide two types in the angle brackets: first the type of the key, then the type of the value we want to store. Here, we are saying that offices will map Strings to Integers.

Now we want to map ”Kathi” to 309. We’ll use the put method, then use a println to see the value (as we go through this demo, each sequence of lines just gets inserted at the end of the main method).

```java
offices.put("Kathi", 309);
System.out.println("Kathi is in "+ offices.get("Kathi"));
```

Now let’s check what happens if we look up ”Tim”, even though we haven’t yet put a value for ”Tim” into the HashMap:

```java
System.out.println("Tim is in "+ offices.get("Tim"));
```

Here, Java prints out null. So Java return null, rather than throw an exception, if the requested key is not in the HashMap.

Now let’s try adding a second office for ”Kathi”:

```java
offices.put("Kathi", 519);
System.out.println("Kathi is in "+ offices.get("Kathi"));
```
What could happen here? We could (a) get both 309 and 519 as values, (b) just get the original value 309, (c) just get the latest value 519, or (d) get an error because there’s already an entry for "Kathi".

We get just 519, with no mention of 309. This points out a key property of hash tables:

*Hash tables store at most one value per key*

In practice, you may want to know what value got kicked out of the hashtable if you store a new value. To support this, the `put` method in Java actually returns whatever value was already stored for the given key:

```java
Integer prevVal = offices.put("Kathi", 519);
System.out.println("Kathi is in " + offices.get("Kathi") + " but was in " + prevVal);
```

### 2.2 Any Type Can Serve as Key

Let’s add one more professor to the HashMap: there’s a professor named "Tim" in the history department whose office is number 202.

```java
offices.put("Tim", 355); // CS Tim
offices.put("Tim", 202); // History Tim
```

Oops! Wait – we have two professors with the same key, so only one (the second entry) will be in the HashMap. To accurately capture Brown professors, we need keys that reflect more information, such as the name and the professor’s department. Let’s define a class for Professors to use as richer keys:

```java
public class Professor {
    public String name;
    public String dept;
    public String currCourse;

    Professor(String name, String dept, String course) {
        this.name = name;
        this.dept = dept;
        this.currCourse = course;
    }

    @Override
    public boolean equals(Object other) {
        if (other instanceof Professor) {
            Professor otherP = (Professor)other;
            return otherP.name.equals(this.name) && otherP.dept.equals(this.dept);
        } else {
            return false;
        }
    }
}
```
In this class, we’re also storing the Professor’s current course. Since courses can change from semester to semester, the class includes an equals method that only considers name and department.

Back to the HTDemo class: let’s define a HashMap from Professor to Integer:

```java
public class HTDemo {
    public static void main(String[] args) {
        // previous content can still be here ...
        Professor kCS = new Professor("Kathi", "cs", "cs18");
        Professor tCS = new Professor("Tim", "cs", "cs18");
        Professor tHist = new Professor("Tim", "history", "hist1266D");

        HashMap<Professor, Integer> offices2 = new HashMap<Professor, Integer>();
        offices2.put(tCS, 355);
        offices2.put(tHist, 202);
        System.out.println("Tim CS is in " + offices2.get(tCS));
    }
}
```

With Professor as the key, both “Tim” professors can be in the HashMap. You can’t call

```java
offices2.get("Tim")
```

because the keys in offices2 are Professor objects, not Strings. But this example shows how it can be useful to have user-defined classes as keys. Any class, interface, or data structure can serve as the value as well (you could create a hashmap that mapped each department name to a hashmap that mapped its professors to offices, for example!).

## 3 Hash Functions

We started this lecture talking about how hashtables are like arrays, except that there’s an extra “step” that maps keys to array indices. This step is critical: we need a function that maps a key to an array index in constant time, otherwise hashtable access can’t be constant time. Where does this function come from?

In Java, every class comes with a method called `hashCode` that maps objects of the class to integers. Let’s try this on our Professor objects:

```java
System.out.println("Tim CS hashcode: " + tCS.hashCode());
System.out.println("Tim Hist hashcode: " + tHist.hashCode());
```

If you try this, you will see two large integers printed. By default, Java creates hashcodes based on an object’s location in memory.

Remember that hash functions are used to map keys to indices. These hashcodes seem too large to be array indices. So how do these turn into indices?

There’s actually one more step: the hashcodes get *compressed* into array indices. The easiest way to do this is to take the modulus of the hashcode by the length of the array (% is the mod operator in Java):
key.hashCode() % array.length;

The compression step is built into a hash table implementation. It is NOT something that that user of HashMap needs to worry about. The hashCode method, however, is something you can control if necessary.

### 3.1 Writing hashCode methods

We’ve said that Java’s default `hashCode` method is based on each object’s location in memory. What if we have two distinct key objects that are identical according to `equals` however? For example:

```java
Professor tCS18 = new Professor("Tim", "cs", "cs18");
Professor tCS1951 = new Professor("Tim", "cs", "cs1951Y");
```

(Tim is teaching two courses this semester; these are the objects for his two courses)

Imagine that we had used one of Tim’s objects to store his office in the HashMap, then tried to look up his office with the other:

```java
offices2.put(tCS18, 355);
offices2.get(tCS1951)
```

The get would (almost certainly) fail – the two objects have different locations in memory, and the mapping to indices is based on memory locations by default. But we want the get to succeed, because we are talking about the same human Tim.

Whenever you have a key for which you overrode the `equals` method, you all need to override the `hashCode` method. Good `hashCode` methods are important – you want them to distribute your keys across the indices (what if they don’t? More on that in a moment). One reasonably good way to do this in practice is to multiply each field that is used in the `equals` method by a different prime, then sum them up. For example:

```java
@override
public int hashCode() {
    return (this.name.hashCode() * 3) + (this.dept.hashCode() * 11);
}
```

This approach is sufficient in practice for most situations. If you are working on a professional-grade system for which the HashCode is particularly critical, someone on the team with expertise in probabilities and distributions would be charged with coming up with a hash function based on profiles of the keys. That isn’t your current situation, so the method we showed here suffices.

### 3.2 Collisions

We motivated the need for writing our own `hashCode` on the concern that the objects for Tim’s two courses would almost certainly map to different array incides. But remember that the compression
step computes the hashCode mod the array length. Isn’t it mathematically possible that two hashCodes would mod to the same array index?

Absolutely, and this does arise in practice. When two keys map to the same array index, it is called a collision. Now we have two objects to store in the same spot in the array, but we said that the array holds only one value per index. What happens?

Hash table implementations typically use one of two techniques for handling collisions. In one (called chaining), the array actually stores a list of items that mapped to the same index. In the other (called open addressing), the implementation searches for an empty array slot in which to store the value. When you are using built-in HashMaps as a programmer, you don’t have to worry about this issue. If you are implementing the HashTable data structure, however, you do.

Which brings us to our next topic ....

4 Building HashTables for Ourselves

To help you understand how hash tables work under the hood, we will build a hash table data structure for ourselves (as we did for linked lists). To avoid confusion with Java’s HashMap version, we will refer to ours as a dictionary (the term used in several languages, including Python), and we will change the names and behaviors of some of the methods.

Based on what we have seen, what does a hashtable implementation entail?

- There’s an array to hold the values
- There are keys and a hash function to map keys to (possibly large) integers
- There’s a compression function to turn the hash function results into indices
- The hash table must manage collisions (which is when two keys map to the same index in the array)

For collisions, we will implement the Chaining method. This means that the array contains not one value, but a list of values that map to the same index. Actually, it’s a bit more subtle. Since multiple keys could land on the same index, to find an object based on its key, the array needs to store both the keys and the values in the list at each index. We do with a data structure (in our case, a simple class with two fields) that pairs keys and values. So really, our array is something like:

```
LinkedList<KeyValuePair<K,V>>[size] theArray;
```

where K and V are the type parameters for the key and value that you want to store in the hash table. size is the size that you want for the array.

Next lecture, we will pick up here, looking at the details of implementing hash tables more closely.
Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: https://cs.brown.edu/courses/cs018/feedback