

CS231 Fall 2013 Lecture 36

Stephanie R Taylor

December 2, 2013

1 Administrative Topics

- Yes, there is lab today. It is designed to help you hunt the wumpus.
- Course evaluations begin today. Please fill them out.
- On the course evaluation is a question about whether the course has met the learning goals. As a reminder, they are:
 1. Students understand the advantages and disadvantages of fundamental data structures and can implement them using object-oriented design principles.
 2. Students understand, can implement, and can calculate the time and space efficiency of classic search, sort, and traversal algorithms, including the use of big-Oh notation.
 3. Students understand the tradeoffs between different implementation of data structures and algorithms and can make appropriate design decisions based on application data requirements.
 4. Students can use fundamental data structures and algorithms appropriately to solve a variety of computational problems.
 5. Students can communicate the result of their work and describe an algorithm.

2 Hash tables/maps

Last week, we talked about skip lists, which were efficient data structures for keeping data sorted according to keys. We talked about how we could use it to retrieve a value by its key. Today, we talk about another data structure that is useful for that – a hash table (called a map in Java terminology).

It would be nice if we could create an array whose indices are Strings instead of only ints, which is what Java requires of indices. If we could use Strings as indices then we could have an array

`Record[] table`

and, when we want to look up the record for a name Steve Smith, we could just do:

`table["Steve Smith"]`

to get the record. Even if we could do that, another problem is that we need to choose the size of the array when we first construct it. How big should the array be? If the array is too big, it wastes space. If it's too small, it might cause an error, or we might have to add complicated code to make the array bigger.

Well, we cant do that, but we can do something just as good! What we can do is use a Map.

Generalizing: Maps are more general than arrays because they can use any type of object as an index. The basic operation on a map is a lookup, where you provide a key (of any non-primitive kind), which is like the index into an array and the map returns the corresponding value (of any kind).

Abstracting: Like all ADTs, the map is defined by its set of operations, not by its implementation. As usual, there are many possible implementations for a map. They all have the same behavior, but different performance characteristics.

Map entries The things that go in a map are entries, where each entry contains a key and a value.

The key, K, is like a handle you use to look things up. The value, V, is the information you are storing.

Example: in a dictionary, the word is the key and the value is the definition

Example: in a phone book, the name is the key and the value is the phone number

Example: at the credit card company, the credit card number is the key and the value is the available credit for that account

Map operations

- Construct an empty map.
- Test for an empty map.
- Get the number of entries in the map.
- Add a new entry to the table, where an entry is a pair (K, V)
- Update an entry – change the value associated with a key (same as deleting the old entry and adding a new one)
- Retrieve, or lookup, an entry. You provide K; retrieve returns V
- Remove an entry from the table, given the key K
- Clear all entries from the map.
- Enumerate the entries in the table.

Java Map interface In Java, there is an interface Map. Both the keys and the information are Objects.

Java's Map interface has the following public methods (and a few more not listed here):

- public boolean isEmpty()
- public int size()
- public Object put(Object key, Object value)
- public Object get(Object key)
- public Object remove(Object key)

- `public void clear()`

There are several classes provided for you that implement this interface. One is the `TreeMap` that uses a tree to store the entries. We won't study that class. Instead, we'll look at another class that implements the interface. It is called "`HashMap`".

How are `HashMap` tables implemented?

Use hash tables, which are a remarkable idea that makes it possible (with certain assumptions) to insert, remove, and lookup things in a table in constant time!

Hash tables Store the values in an array. The problem is that the keys aren't integers. Solution: Use a "hash" function to convert the keys into integers.

For example, if the keys are of type string, we could use the hash function:

```
int hash (String s) {
    return s.length();
}
```

This function converts all strings into integers.

Example: to store the key-value pair ("a", "01"):

1. convert the key "a" to 1 using the hash function above
2. store the value "01" at the 1th array location

[Draw a picture of a table with two rows ["a", "01"] and ["the", "111"] and a hash function and array implementation]

Based on this idea, we can now implement a `HashMap` class:

```
class HashMap {
    Object[] data;

    void put(key, value) {
        data[hash(key)] = value;
    }
    Object get(key) {
        return data[hash(key)];
    }
    int hash(Object key) {
```

```

        ...
    }
}

```

But actually this won't work. What's the problem with this approach? The hash function might not be one-to-one. What do we do if two or more keys are converted by the hash function into the same integer? In that case, which of the two values do we store in the array location indexed by that integer? This is called a collision.

Solution 1: Ensure that the hash function is 1-to-1 and so every key converts to a unique integer. Can such functions always be found? [Use address in memory of the object] The hashCode() method in the Object class gives us this, so we can implement the hash function above by just using "return key.hashCode();". What's the problem with this solution? The resulting array is too big. How many addresses are there in memory? [4 billion] Do you want an array that big, even if we are storing only about 50 pairs in the table? Also consider the credit card example. How many credit card numbers are there? [10^{16}] For hashing to be practical, we need to be able to make the array size proportional to the number of entries, not the number of keys.

Is there a hash function that is 1-to-1 and doesn't need a huge array? If the exact set of keys is known ahead of time, it is possible to construct a perfect hash function that stores the n values in an array of size n and converts the n keys into unique values from 0 to $n-1$. But usually we don't know the exact set of keys.

Solution 2: Allow the hash function to be not 1-to-1. This is what is actually done. Real hash functions lose information; that is, they map many input values to the same output value. In other words, many keys get mapped to the same location. In other words, given the output of the hash function, we don't know what the input was.

One way to do this would be to implement the hash function above using `return key.hashCode() % data.length;`.

Two design goals: minimize collisions and properly handle the (few) collisions that occur.

Problems:

1. We can't tell for sure which of several possible entries is stored at a given location. For example, if we stored the value "01" for "a" in array slot 1, but then did a lookup of "I", we would look in array slot 1 but the wrong value is there.

Solution? [we need to store the keys with the information, so we can check. That is, we need to store the (K,V) pair in each slot in the array.]

2. there might be more than one entry vying for the same location (we hope the number of collisions is small, if the array is a few times bigger than the number of entries)

Solution 1 to Problem 2: Linear probing down from the hash index. Ugh.

Solution 2 to Problem 2: separate chaining:

1. each hash table entry is a List of key-value pairs
2. to add an entry, you hash the key, and add the entry to the corresponding list
3. to lookup an entry, hash the key and search the corresponding list

Solution 3 to Problem 2: Use a second hash function if there is a collision. If the second also yields a collision then do separate chaining.

Why do the more complicated Solution 3? It is important to use a hashing function that spreads out the values (and an array that is sufficiently large) so that, statistically, we can expect the number of entries per array location to be 0 or 1, and only occasionally greater than 1. Then, in this case, the time for lookup, insertion, and deletion are constant!

What does the HashMap built-in class do? It uses the built-in hashCode() method to convert the key into an index. That method uses the address of the object in memory to get the hash value and so works well in general, but you can override it if you wish. The default toString() method actually displays this hashCode value (in Hex notation).

What is the time complexity then for HashMap's get, put, and remove? They are $O(1)$ on average, but in worst case are $O(n)$.

Aside: When you create HashMaps using the library HashMap class, don't forget to indicate the type of keys and values you'll be using:

```
HashMap<String ,String> map = new HashMap<String ,String >();
```

Aside: Notice here that to get the best behavior, we combine arrays, lists and pairs to form a new data structure. This shows the value of learning new data structures, and why this course is foundational material to future CS courses.