Joanna Klukowska
joannakl@cs.nyu.edu

Lecture 8: Hashing and Hash Tables

Reading materials

Dale, Joyce, Weems: 10.6

OpenDSA: Chapter 7

Liang: only in Comprehensive edition, chapter 28

Topics Covered

Joanna Klukowska
joannakl@cs.nyu.edu

1 Introduction to Hash Tables

1.1 Motivation

We want a data structure that allows us to access existing elements and insert new elements in O(1) operations. Is it possible? Theoretically, yes, in practice, we can get close, but there is always a trade off between memory and time usage.

1.2 What are hash tables?

A hash table is a look-up table that, when designed well, has nearly O(1) average running time for a find or insert operation. More precisely, a **hash table** is an array of fixed size containing data items with unique keys, together with a function called a **hash function** that maps keys to indexes in the table/array.

Example:

If the keys are integers and the hash table is an array of size 127, then the function `hash(key)`, defined by

```
hash(key) = key % 127
```

maps numbers to their modulus in the finite field of size 127.

Notice:

- for each key (a number in the above example) there is only one possible value of `hash(key)`,
- multiple keys may have the same value of `hash(key)` (i.e. the hash function is not one-to-one):
  - for example the keys 10, 137, and 264 all map to the same array location because 10 % 127 = 137 % 127 = 264 % 127 = 10.

Conceptually, a hash table is a very general structure: it is a table `H` containing a collection of (key, value) pairs with the property that `H` may be indexed by the `key` itself. We usually reference an element of an array `A` by writing something like `A[i]`, using an integer index value `i`. With a hash table, we replace the index value "`i`" by the `key` contained in location `i`. For example, if `H` contains the set of pairs

```
("Italy", "Rome")
("Japan", "Nagano")
("Canada", "Banff")
("France", "Paris")
("Belgium", "Bruges")
("Hungary", "Budapest")
("Portugal", "Porto")
```

Joanna Klukowska
joannakl@cs.nyu.edu

then we could hypothetically write a statement such as

```
print H["Italy"]
```

and `Rome` would be printed, or

```
print H["Portugal"]
```

and `Porto` would be printed (these, of course, are not valid Java statements).

As long as we know the `key` associated with the data item, we can access it in the table in, at least theoretically, O(1) time. Similarly, when we want to insert a new data item into the table, we simply determine its key and add it to the table at the location `H[key]`.

Problem: Well, sometimes we cannot do this, can we? Think of trying to add another pair `(''Italy'', ''Florence'')` to the above table.

2 Using Java Provided Hash Tables

Java provides a `Map` interface and several possible implementations of it. When we use those interfaces and classes we need to make sure that a good hash function is implemented for the objects that are used as keys in the hash table.

2.1 `Map<K,V>` interface

http://docs.oracle.com/javase/8/docs/api/java/util/Map.html

```
interface Map<K,V>
```

An object that maps keys of type `K` to values of type `V` (K and V are generic types, this is a good example of using non-standard generic letters that have their own meaning in the `Map` structure - you can use whatever letters you wish when you implement your own hash tables). **A map cannot contain duplicate `key`s; each `key` can map to at most one value.**

There are several methods in the `Map` interface that we are going to be using and you should be familiar with:

`V put(K key, V value)` Associates the specified `value` with the specified `key` in this map. Returns the previous value associated with `key`, or `null` if there was no mapping for key.

`V get(Object key)` Returns the value to which the specified `key` is mapped, or `null` if this map contains no mapping for the `key`.

`V remove(Object key)` Removes the mapping for a `key` from this map if it is present. Returns the previous value associated with key, or null if there was no mapping for key.

`Set<K> keySet()` Returns a `Set` view of the keys contained in this map.

Joanna Klukowska
joannakl@cs.nyu.edu

2.2 `Hashtable<K,V>` class

http://docs.oracle.com/javase/8/docs/api/java/util/Hashtable.html

```
class Hashtable<K,V> implements Map<K,V>
```

This class implements a hash table, which maps keys to values. Any non-null object can be used as a key or as a value.

To successfully store and retrieve objects from a hash table, the objects used as keys must implement the `hashCode` method and the `equals` method.

Source Code: see `Capitals.java` for an example how the Hashtable class can be used with String keys and String values.

2.3 `HashMap<K,V>` class

http://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

```
class Hashtable<K,V> implements Map<K,V>
```

This class implements a hash table, which maps keys to values. The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.

To successfully store and retrieve objects from a hash table, the objects used as keys must implement the `hashCode` method and the `equals` method.

2.4 What does really happen?

Java provided hash tables hide the implementation details (as they should) and it is not clear really how the values are stored and retrieved. There are possibly ”infinitely” many different keys - how can all these values be stored in a table?

In the next section we will try to answer these questions.

3 Implementing Hash Tables

What is used for storing a hash table? Usually arrays are used to store hash tables. (But three might be other structures stores within each locations in the array.)

What is the type of such array? The answer to this question depends on the implementation choices that are made in answer to questions below. In general, a single array location has to contain the (key, value) pair. Sometimes a single array location can store multiple (key, value) pairs.

Joanna Klukowska
joannakl@cs.nyu.edu

How is the location in the array decided based on the key? This is where the hash function mentioned in Section 1 comes in. Hash function is applied to the key and it returns the index of an array location in which the given (key, value) pair should be stored. The details of different hash functions will be discussed below.

What happens if hash function puts two different keys into the same array location? A **collision** occurs when two different keys are sent to the same array location. Below, we will discuss several different approaches of **collision resolution** (deciding what to do when collision occurs).

Can the array be full and what do we do when it is? The array can certainly become too small to store all the data. **Rehashing** is the technique used to create a new array and copy all the values to a new array.

3.1 `(key, value)` Node

As we did with linked lists, we will combine the key and the value into a node object. Depending on the choices made for collision resolution, such a node may or may not need to store a reference to another node. But in either situation, the node needs to store the `key` and the `value`.

```
class HashNode < K, V > {

    private K key;
    private V value;
    ...

}
```

THINK ABOUT: Why do we need to store the key in addition to the value, once we find the correct location for the pair based on the key?

3.2 Hash Function

Based on unique keys we need to be able to compute the location in the array (or hash table) at which the given `(key, value)` combination can be stored. Why can't we just use the key itself as an index? Well, it might be too big, negative or not an integer. We use a hash function (called `hashCode()` in Java) to decide the location. What should such hash function do?

```
hash(key) = ?
```

Joanna Klukowska
joannakl@cs.nyu.edu

3.2.1 Properties of a Good Hash Function

A hash function is supposed to chop up its argument and construct a value out of the chopped up little pieces. Good hash functions make the original key hard to reconstruct from the computed hash value. To be good, a hash function should

- be easy to compute (for speed),
- repeatable and
- randomly disperse keys evenly throughout the table, making sure that no two keys map to the same index.

Easy to compute generally means that the function is an O(1) operation, practically independent of the input size and hash table size. For example, if the function tried to find all of the prime factors of a given number in order to compute the hash function, this would not be easy to compute. Being easy to compute is a fuzzy concept.

Repeatable means that making two calls to a hash function with the same argument should produce the same result. It would be much easier to compute values quickly and disperse them well if you could use random numbers, but there is not way to recover the data from the hash table once it is saved there.

Dispersing the keys evenly means that there is as much distance between successive pairs of keys as possible. For example, if the hash table is of size 1000 and there are 200 keys in it, they should each be about five addresses apart from their neighbors.

In principle, if the set of keys is finite and known in advance, we can construct a perfect hash function, one that maps each key to a unique index. In practice, the set of keys is rarely known at the time of writing the program and may not be finite.

Example (bad hash function): If we have the integer keys

$$112,\ 46,\ 75,\ 515$$

we would want a function that maps them to the numbers0,1,2, and 3 uniquely. Suppose that hash(key) is a function that returns the sum of the decimal digits in the key and if that sum has more than one digit itself we add them together again . In this case we have

$$\text{hash}(112) = 1 + 1 + 2 = 4,$$
$$\text{hash}(46) = \text{hash}(4 + 6) = \text{hash}(10) = 1,$$
$$\text{hash}(75) = h(7 + 5) = \text{hash}(12) = 3, \text{ and}$$
$$\text{hash}(515) = h(5 + 1 + 5) = \text{hash}(11) = 2.$$

This is a perfect hash function for the above four keys. But it is very poor hash function for larger number of keys. Why is it bad:

- It matches all keys to one digit indexes of an array. What if we have 250 keys?
- It completely ignores information about position of digits in the key so 155, 515, 551 are all assigned the same index.

Joanna Klukowska
joannakl@cs.nyu.edu

3.2.2 Good Hash Functions

Well, the definition of a good hash function depends on what it is used for and what the size of the table is. There are a lot of different methods of computing the index based on an integer and of computing the integer based on the object itself (assuming the object is not the integer to begin with).

Java provides a method called `hashCode()` for all its classes that can be used as keys into hash tables (`String`, `Integer`, ...). For example, the hashCode() method in the String class is implemented as follows:

```
/**
 * Returns a hash code for this string. The hash code for a
 * String  object is computed as
 *      s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
 * using integer arithmetic, where s[i] is the
 * i'th character of the string,  n is the length of
 * the string, and ^ indicates exponentiation.
 * (The hash value of the empty string is zero.)
 *
 * @return  a hash code value for this object.
 */
public int hashCode() {
    int h = hash;
    if (h == 0 && value.length > 0) {
        char val[] = value;

        for (int i = 0; i < value.length; i++) {
            h = 31 * h + val[i];
        }
        hash = h;
    }
    return h;
}
```

The Object class has the following documentation/implementation for the hashCode() method (remember that every class inherits the hashCode() method from the Object class):

```
    /**
     * Returns a hash code value for the object. This method is supported for
     * the benefit of hash tables such as those provided by java.util.HashMap.
     *
     * The general contract of hashCode is:
     *  -  Whenever it is invoked on the same object more than once during
     *     an execution of a Java application, the hashCode method
     *     must consistently return the same integer, provided no information
     *     used in equals comparisons on the object is modified.
     *     This integer need not remain consistent from one execution of an
     *     application to another execution of the same application.
     *  -  If two objects are equal according to the equals(Object)
     *     method, then calling the hashCode method on each of
     *     the two objects must produce the same integer result.
     *  -  It is not required that if two objects are unequal,
     *     according to the java.lang.Object#equals(java.lang.Object)
     *     method, then calling the hashCode method on each of the
     *     two objects must produce distinct integer results.  However, the
     *     programmer should be aware that producing distinct integer results
     *     for unequal objects may improve the performance of hash tables.
     *
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
22      * As much as is reasonably practical, the hashCode method defined by
23      * class Object does return distinct integers for distinct
24      * objects. (This is typically implemented by converting the internal
25      * address of the object into an integer, but this implementation
26      * technique is not required by the Java programming language.)
27      *
28      * @return  a hash code value for this object.
29      * @see     java.lang.Object#equals(java.lang.Object)
30      * @see     java.lang.System#identityHashCode
31      */
32
33     public native int hashCode();
```

For your own classes, you should implement your own `hashCode()` method if the objects of the class can be used as keys into a hash table.

The computation of a hash value based on an integer value often uses some computation with a prime number followed by modulus operation that used the size of the table.

Assume that N is the number of elements that can be stored in the hash table. The following are considered to be pretty good hash functions for integer keys (assuming that the keys are roughly uniformly distributed):

```
hash( key ) = key % N;
```

```
double goldenRatio = 0.6180339887;
```

$\text{hash(key)} = (\text{int})(\text{floor}(\text{N} * (\text{key} * \text{goldenRatio} - \text{floor}(\text{key} * \text{goldenRatio}))))$

If we are starting with a String object as the key, we need to convert the characters into an integer value first, and then apply an integer hash function (one of the ones above, for example). To get an integer from a String, we need to use ALL of the characters and their position information to obtain the best results. Given a String S, some examples of such conversion are:

```
int num = 0;
for (int i = 0; i < S.length(); i++ )
{
    num = 37*num + (int) S.charAt(i);
}
```

```
int num = 0;
int primePow = 37;
for (int i = 0; i < S.length(); i++ )
{
    num +=  primePow * (int) S.charAt(i);
    primePow = primePow*37;
}
```

Joanna Klukowska
joannakl@cs.nyu.edu

3.3 Collision Resolution ...

... or what to do when two keys are hashed to the same location in the hash table.

There are different ways of handling situations in which two keys map to the same location (called **collision**):

- **open addressing** (also known as closed hashing) - finds an alternative location for the `(key, value)` pair, if the first location is occupied,
- **closed addressing** (also known as open hashing) - allows multiple `(key, value)` pairs to be stored in a single array location.

Each of these two have multiple ways of being implemented. Here are examples of how they can be handled.

Open addressing:

- linear probing - put the `(key, value)` pair in the next available location, if that is occupied, try the next one and so on,
- quadratic probing - the the `(key, value)` pair in the next available location, if that is occupied, the the one four spaces away, then nine spaces away, and so on (use a square of how many times we tried to put it in the next location).

In both of these, we need to handle retrieving elements from the hash table in a special way - they may not be in the locations computed by the hash value of the key. If the `(key, value)` pair is not found at the location computed by the hash function, we have to check all the other places in which the collision resolution method might have place them.

Closed addressing:

- separate chaining - put each `(key, value)` pair that maps to the same array location in a linked list that starts at that array location. If implemented well, the linked lists remain always very short.

3.4 Rehashing

To guarantee good performance, the load factor (number of elements divided by the size of the array) should be less than 1 (often it is expected to be below 0.75). When array becomes too full (i.e. the load factor exceeds some predefined value), all the values stored in the table need to be rehashed to a larger table.