Lecture 29: Hash tables

Doina Precup
With many thanks to Prakash Panagaden and Mathieu Blanchette

March 19, 2014

1 Dictionaries

In many applications, we need to maintain a collection of data in which elements have *unique identifiers or keys*. For example, maybe we need to maintain information about all McGill students; each student has a unique student ID. A collection of pairs of the form *(key, information)* is often called a **dictionary** (by analogy with word dictionaries, in which the word is the key and the associated meaning is the information). Dictionaries have to support the following operations:

- Object find(key) - retrieve the information corresponding to a given key
- void insert(key, information) - insert a pair in the dictionary
- void delete(key) - remove the information associated with the given key

Optionally, we can also modify the information associated with a given key (but of course, this could be implemented by a delete and insert pair of operations). We will assume that the set of possible keys is ordered and finite (e.g. integers, strings of a finite length, etc).

So far, we have discussed several methods that are useful for maintaining dictionaries: arrays, linked lists, and binary search trees (heaps are used for priority queues, and as such, do not support efficient search of arbitrary keys). All have their advantages and disadvantages, in terms of the time it takes to search information, insert and delete new elements.

For example, as you may recall, in a linked list, finding or removing an element will take $O(n)$ in the worst case, where n is the number of elements (insertion can be done in $O(1)$, if we always insert at one end of the list). In a sorted array, search takes $O(\log n)$, but insertion and removal will take $O(n)$ in the worst case (as we may have to shift the content of the array). In a binary search tree, all the operations take $O(k)$ in the worst case, where k is the depth of the tree. If we balance the tree, this will be $O(\log n)$, but the balancing requires a bit more memory and trickier code.

Today we discuss a new data structure, called a **hash table**, which is often the preferred solution in practice. As we will see, if we are a bit careful, a hash table will allow us to do most operations in $O(1)$ (even though the worst case will not be as good).

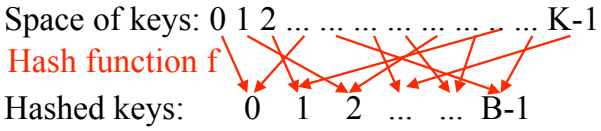
2 Hash tables

Suppose that we know that we will get entries with integer keys between 0 and $K-1$. Then we could use an array of size K to store the information associated with the keys. The Java code would look roughly as follows:

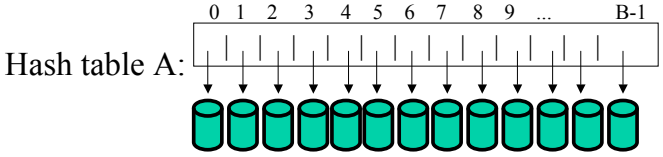
```
public class Dictionary {
        private Object a[] = new Object(k); //array to hold the information
        //...
        public Object find (int key) {return a[key];}
        public void insert (int key, Object info) {a[key] = info;}
        public void delete(int key) {a[key]=null; }
}
```

Note that all these operations are $O(1)$! This is really nice, but there is a catch: if K can be very large, then we will need a huge array to store the information, so we may not have enough memory to do this. Moreover, in many practical applications, not all keys need to be stored at the same time; instead, at any given time, the real number of elements that we need is often much smaller than K. For example, McGill's ID numbers are 9 digits long, so the maximum number of keys is $10^9 = 1,000,000,000$. But only about $30,000$ students are enrolled at any given time. If we used an array of size 10^9, it would be a huge waste of resources, as the array would be very sparsely populated. We would like to use a data structure which takes advantage of the fact that only a small fraction of entries are populated. This is precisely the goal of a hash table.

In a hash table, we will use an array of size $B \ll K$ to hold the dictionary. The keys will be mapped to indices in this array by a function $f : \{0 \dots K-1\} \to \{0, \dots B-1\}$, called a **hashing function**. This is illustrated below:



Of course, now there is no guarantee that every element will be mapped to a different location in the array: f may map elements with different keys to the same table entry either because the table is too small, or just because of the way f works. This is called a **collision**. How do we handle it? The basic idea is to have a **bucket** in each array entry, instead of a simple entry:



A bucket can be a linked list, another array, a binary search tree, or even another hash table. If we use linked lists for each bucket, the Java code above will be modified as follows:

```
public class Dictionary {
```

```
private LinkedList a[k]; //array to hold the information
//...
private int hash(int n) {//... };
public Object find (int key) {return a[hash(key)].getFirst();}
public void insert (int key, Object info)
        {a[hash(key)].insert(key, info);}
public delete(int key) {a[hash(key)].delete(); }
}
```

Now, the complexity of these operations will be determined by the data structure used for the buckets, and by how well the hash function distributes the data in the buckets. In Java, the HashMap class provides a useful hash table implementation, and we will use it later in the course.

3 Hash functions

Having a good hash function is really key to a good hash table. For example, assume that each bucket is a linked list. Then in the worst case, the hash function will map all elements into one bucket, which will be of size n. On the other hand, assuming that all entries are equally likely, a good hash function will make roughly equal lists, each of size n/B, where B is the number of entries in the hash table.

If not all entries have to be accessed with similar frequency, we would like the hashing function to map entries which are accessed a lot by themselves, while entries that are accessed little can be mapped together. This will give a good *average access cost*.

There is a whole literature on "good" hashing functions, because what is good depends on the domain. For example, consider again the case of the McGill student IDs. Using something like the first 4 digits of the student ID would be really bad, because all numbers start with 260, followed by 2 digits that indicate the year when the student started. So, if we used this function, all students would be mapped to just a few entries! On the other hand, using the last 5 digits is perhaps better, because the last part of the ID is just a counter. So intuitively, the last 4 digits should give us a more uniform spread of the students into buckets. In general, we can try to get a more uniform spread by performing some operations on the digits of the number, e.g. adding them, computing a polynomial etc.

Usually, no matter what processing we do, in the end we will map the result into the m buckets of the table by using the **modulo** operator. Recall that $p\%B$, or p **mod** B is the remainder of the integer division of p to B. For example, $13\%10 = 3$, $10\%6 = 4$. So what is a good value for B? Often people will use a *prime* B. Recall that a **prime number** is only divisible by 1 and itself. The intuition behind using primes is that they will help "even out" the result, in the event that all our work still leaves us with a skewed distribution.

For example, suppose that as a result of our previous operations, we were left with a lot of numbers that are divisible by 5. If we use a table size which is a multiple of 5, all these keys will be clustered together in the same buckets. For example, if we use a table of size 100, we will get a lot of entries in buckets 0, 5, 10, 15, ... 95, and very few entries in any other buckets. Even if we use something like 125, there will still be a lot of collisions. For example, elements with keys

155, 255 and 355 will all be grouped in buckets with indices divisible by 5 (the indices are 30, 5 and 105 respectively). So we still get very large buckets for divisors of 5 and very small buckets otherwise. What happens if we use a prime number, e.g. 101? In our example, 155 will be mapped to bucket 54, 255 to bucket 53, 355 to bucket 52. So the distribution appears a lot more even, and in particular buckets that are multiples of 5 are not overflowing anymore. Since we may not know in advance *which* kind of flaw our distribution will have, a prime size B is a "vanilla" solution: we will only be unlucky if somehow the distribution favors produces a lot of numbers that are multiples of B. If we choose a table of non-prime size, $p_1 \cdot p_2 \cdot \ldots \cdot p_k$, we will be unlucky with distributions in which we see more numbers divisible by *any* of the p_i factors.

To see why this is, suppose we have a key k and a table of size B divisible by p, i.e. $B = p * n$. Suppose k is also divisible by p, i.e. $k = p * m$. Then we have:

$$\begin{aligned} k\%B &= k - B * l, \text{ where } l \text{ is some natural number} \\ &= p * m - p * n * l \\ &= p * (m - n * l) \end{aligned}$$

So $k\%B$ is also divisible by p, which means that k will fall in a bucket of index divisible by p. To get an intuitive feel for this phenomenon, you can use the HashFn.java code from the web page. E.g., try it out with hash table sizes 100, 101, 102, 103, 104, 105, and see what happens.

If the key is a string, we can map it to an integer by taking the sum of the ASCII codes of the characters. However, this may also lead to collisions, as the distribution of letters in a language is not uniform, so the ASCII code sums will give a skewed distribution. It is better to compute a polynomial of a prime number (in this case, it could be a small number), where the ASCII codes of the characters are coefficients. From then on, we can use a hashing function like for integers. In general, any object is represented in memory as a string of bits, which of course can be interpreted as an integer, and so it can be given as input to a hashing function.

It is worthwhile, when using a hash table, to adopt good hashing functions from the computer science literature, based on knowledge of the application at hand.

4 Notes on Java implementation

In Java, hash tables are implemented through the HashMap class, which implements the Map interface. Maps are mappings from keys to values (the same concept as dictionaries). In older versions of Java, there used to be a Dictionary abstract class instead, but the use of an interface is more convenient.

The HashMap class constructor lets you set (if you want) two parameters: the initial desired capacity and a "load factor". When the HashMap contains a number of objects equal to the capacity times the load factor, it will automatically re-size to a larger capacity. This operation obviously is expensive, but it ensures that the access operations are roughly $O(1)$ (assuming a "good" hash function).

The hashing is implemented using the hashCode() method that is associated with the Object class. If you want to use the HashMap for your particular data, and you want it to work well, you can re-define the hashCode() function to something that is appropriate to the data. If two objects are equal according to the equals() method, they must have the same hash code.