Lecture 16: Implementing HashTables

11:00 AM, Feb 26, 2021

Contents

Objectives

By the end of these notes, you will know:

- How hashtables use chaining to manage collisions
- Worst-case and average-case run-time performance for hash tables
- Java's defaults for hash table size and collision management

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

- Implement the chaining method for collision management in hashtables
- choose reasonable array sizes and hashcode policies for hashtables in general programming contexts

1 Implementing Hashtables

Let's talk about how to implement hashtables. First, let's articulate the interface that we want our hashtable implementation to satisfy.

```
interface IDictionary<K,V> {
   // Looks up a value in the dictionary, given its key.
   // throws KeyNotFoundException if the key is not found
   public V lookup(K key) throws KeyNotFoundException;

   // Updates the value associated with the given key.
   // throws KeyNotFoundException if the key is not found
   public V update(K key, V value) throws KeyNotFoundException;

   // Inserts a key-value pair into the dictionary.
   // throws KeyAlreadyExistsException if the key already exists
   public void insert(K key, V value) throws KeyAlreadyExistsException;

   // Deletes a key-value pair from the dictionary.
   // throws KeyNotFoundException if the key is not found
   public V delete(K key) throws KeyNotFoundException;
}
```

Note that this interface has different methods from the `put` and `get` methods that are in Java's `HashMap` class. In part, we want our methods to behave slightly differently (such as throwing exceptions instead of returning `null`), so we give them different names. In part, we want to avoid confusion with the built-in operators.

Let's review: what does a hashtable implementation need?

1. Information about the types of keys and values to store in the hashtable
2. An array to hold the values
3. A hash function to map keys to (possibly large) integers
4. A compression function to turn the hash function results into indices
5. A way to manage collisions (when two keys map to the same index in the array)

We can rely on Java's `hashCode` method for the hash function, and we have already talked about computing the modulus of the hashCode by the array length for compression. The types of the key and value can just be type parameters to our hashtable class (as they are for Java's `HashMap`). The interesting parts then are the array and collisions.

1.1 Understanding Collisions

Let's take a simple (and contrived) example that will illustrate the problem of collisions, both practically and in implementation. Let's map people (by name) to their childhood home state. Here's a class for people, with a ridiculous (but allowable, since it type checks) choice of `hashCode` function (that maps short strings to 0 and longer strings to 1).

```
class Person {
  String name;

  @Override
```

```
  public int hashCode() {
    if (name.length() < 6) {
      return 0;
    } else {
      return 1;
    }
  }
}
```

If we created two `Person` objects with similar name lengths, then computed their `hashCodes`, we'd find they mapped to the same integer, which would put them in the same position within the underlying array. But we want to be able to look up either of these `Persons` in the hash table; this means the hash table must store a list of values at each index, not just one.

```
    Person kPerson = new Person("Kathi");
    Person tPerson = new Person("Tim");
    System.out.println(kPerson.hashCode());
    System.out.println(tPerson.hashCode());
  }
}
```

1.2 The Underlying Array, Take 1

Here's a basic class with a first attempt at implementing a hashtable (this isn't what you'll implement in homework – we'll get to that by the end of the lecture). We're intentionally leaving off the `implements IDictionary` annotation, so that we can focus on only a couple of the operations. We'll also ignore the exceptions for the moment.

```
public class HashTable<K,V> {
  int size;
  LinkedList<V>[] contents;

  HashTable(int size) {
    this.size = size;
    this.contents = (LinkedList<V>[]) new LinkedList[size];
  }

  /**
     * insert the value into the array under the given key
     */
  public void insert(K key, V value) {
    // hash the key and apply compression
    int index = key.hashCode() % size;
    // store the value under the key's index
    this.contents[index].addFirst(value);
  }
}
```

The behavior of `insert` is straightforward: we compute the index that corresponds to the key, and add the value to the list.

Now, let's consider how we extract an item from the hashtable. For our implementation, the method for that is called `lookup`: given a key, `lookup` should return the value associated with that key.

Under this scheme, `lookup` would simply apply the hash function and compression, then return the value stored in the array.

```
public V lookup(K key, V value) {
   // hash the key and apply compression
   int index = key.hashCode() % size;
   // retrieve the value under the key's index
   return this.contents[index];
 }
```

This code doesn't even compile! We want to get back a single value, but the contents array stores a list. So we need to find the specific value within the list that corresponds to the key. But if we look at the array contents (say by printing them out), we don't have that information. We stored the values, but not the keys. To deal with collisions, then, we need a slightly more sophisticated design.

1.3 The Underlying Array, Take 2

Instead of just storing values in the array, we will store pairs containing the keys and the values. Here's a class for such pairs, where K and V are the same types as before for the Key and Value.

```
protected static class KVPair<K, V> {
    public K key;
    public V value;

    /**
     * @param key - key in the pair
     * @param value - value in the pair
     */
    public KVPair(K key, V value) {
      this.key = key;
      this.value = value;
    }
 }
```

Now, we edit the contents variable to create an array of these pairs:

```
  LinkedList<KeyValuePair<K,V>>[] contents;
  // then, in the constructor ...
  this.contents = (LinkedList<KeyValuePair<K,V>>[]) new LinkedList[size];
```

```
 /**
    * insert the value into the array associated with the given key
    */
 public void insert(K key, V value) {
   // hash the key and apply compression
   int index = key.hashCode() % size;
   // store a key-value pair under the key's index
   this.contents[index].addFirst(new KeyValuePair(key, value));
 }
```

Now, the array has the information that `lookup` needs: `lookup` can search the list stored at an index for the right pair, then return the value from that pair.

As you work through implementing the operations in the interface, you will find you often need to look for a key-value pair corresponding to a particular key. It therefore makes sense to create a dedicated helper method for this purpose:

```
protected method KVPair<K, V> findKVPair(K key) {
  // return the pair for the given key
}
```

Writing this method is one of the tasks on the hashtables homework.

But wait, what if we have two `KVPair` for the same key in the hashtable? Which one should we return? Trick question – hashtables store *at most one value per key*, so this can't happen. It is important to understand that collisions are not about duplicate values for the same key. There is at most one value per key; collisions are when our hash function maps two different keys to the same index.

2 Hashtables: Runtime Performance

2.1 Worst-Case Runtime

Remember that we started looking at hashtables as an alternative to storing data in lists: hashtables promised to leverage constant-time array access to improve the runtime cost of looking up/finding items in a set. Due to collisions, we don't necessarily get constant-time lookup. In the worst case, all of the keys would hash to the same index, meaning that we would have a list of all key-value pairs in one index in the array. Searching that list for a specific key-value pair would thus be linear in the number of keys in the hashtable.

What about the other operations?

- Inserting a key-value pair into the hash table takes constant time (assuming that computing the hashcode is constant time), since we can add the new pair to the front of the corresponding LinkedList.
- Deleting the value associated with a key could take linear time in the number of keys, if we have to search an entire linear list to find the key to delete.
- Update depends on how you implement it. If you search for the existing key-value pair and update the value, you could need time linear in the number of keys. You could, however, just add the most recent key-value pair to the front of the list and not delete the old one (searching from the front of the list so you always find the most recent value) – what would that do to the worst-case runtime?

Overall, most hashtable operations have worst-case time that is linear in the number of keys. This seems to miss the entire benefit we wanted to get from using an array underneath. Is there any way to do better?

2.2 Average-Case Runtime

One way to do better is to focus not on the *worst* case running time, but instead to think about the *average* case running time. After all, worst-case often happens only in rare cases. If the average or typical running time is better than the worst case, our data structure may work out fine after all.

In the worst case, all of our keys mapped to the same index. Put differently, the compressed keys distribute badly within the set of possible indices. The best scenario would have the keys be evenly distributed across the indices. If we have k keys and m slots in the array, an even distribution would put k/m keys in each slot. If we had only k/m keys per slot, searching for a key-value pair could take at most k/m time.

Consider a concrete example: assume we expected to have to store roughly k keys. If we created a hash table with an array size of $k/2$, then our runtime would be constant $O(1)$ (this would hold true for any array size that's a fraction of k). Thus, there seems to be a benefit to having the size of the array be based on the number of keys. But what if we don't know upfront how many keys we might have?

In practice, hash tables have a *load factor*, which is the ratio of keys to array slots. Once more keys have been inserted than the load-factor allows, hashtables are expanded automatically. For Java HashMaps, the default initial array size is 16 and the default load factor is .75. Once $16 * .75$ keys have been inserted, Java doubles the size of the array to keep the runtime of finding key-value pairs constant time in practice. (Yes, this means that Java targets more indices than keys by default; you are welcome to set a different load factor if your context can tolerate some collisions).

2.3 Choosing Array Sizes and Hash Functions

We've seen that distribution of keys is important to getting good performance from a hash table. While you can't necessarily predict what data you will need to store (and thus what a good array size might be), some choices are likely much worse than others.

Imagine that we did NOT override `hashCode`, and instead let Java compute hash codes based on memory addresses. Memory addresses are usually multiples of 8 (due to the default sizes of space that get allocated in memory – for various reasons, a fixed default size makes computer design much easier). We decide to be more generous than Java and allocate an initial array size of 32. What's likely to happen?

Well, since addresses are multiples of 8 and there are 32 indices, we'll end up mapping to only 4 of the indices – pretty poor use of the buckets in the array. If you're going to use addresses as hashcodes, you are much better off creating an array of size 31 – it's close to 32 in capacity, but 31 is relatively prime to 8, meaning you'll get a better distribution of keys to buckets.

If instead you are willing to write your own hashcode function, then you can embed the primes in the hashcode computation (rather than in the array size). Hence our suggestion from last week that multiplying each field by a unique prime is a good way to create a hashcode. (As an aside, using primes to compute hashcodes also gives you invertible hashcodes, but that's another possibility for another day ...)

2.4 Another Proposal to Improve Worst-Case Runtime

The worst-case runtime arose because we have a list inside of each hashtable bucket. What if we instead put ... another hashtable in each bucket? We could hash on the key with the built-in hashcode to find the outer bucket, then use a different hash function to map the key to a value in an inner hashtable. Wouldn't this improve our performance?

It certainly could, but now you are trading time for space. The hashtables in each array bucket will take some pre-determined amount of space, where a list uses space proportional to the number of elements. You could have many unused array buckets in the nested hash table model. Whether or not this is a problem depends on your context: sometimes time matters more, sometimes space matters more. You pick your data structure based on your needs.

3 Hashcodes in perspective

We've seen hash functions in one context: mapping objects to integers to enable array-based access to data. But hash functions have a much larger role in CS in general, based on a general principle:

> *Hash Functions turn an unordered collection (set) into an ordered one.*

Why is this useful? Because ordered elements lets us use data structures and algorithms that offer good run-time performance. Consider binary search trees (which you covered in CS17). How would you create a BST of `Person` or `Customer` objects from our recent lectures? On their own, these objects have no natural ordering. But if we have a hash function that maps every item to a unique integer, then we can use hashcodes to order the elements and enable searching for them quickly.

Note that the integers ascribed to objects don't carry any particular meaning relative to the objects. They are simply tags that we can use for finding objects. What matters is that tags are ordered relative to each other, and that we can compute the same tag for an object through some function (the hash function).

But what about all that stuff about collision? Wasn't the whole point of collisions that they are unavoidable?

Collisions are unavoidable when we force hashcodes into fewer array indices than we have keys. But for ordering items, there is no limited resource such as array indices. We just need every item to map to a unique integer. Java's default `hashCode` method, which uses memory addresses, meets this guarantee.

Hash functions and hashcodes are thus useful for many applications in CS, when we want to ascribe some order to a collection for use in a data structure or algorithm. Keep this idea in your toolkit as you go forward in CS.

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`.