Lecture 17: Implementing HashTables
10:00 AM, Mar 5, 2018

Contents

1 Implementing Hashtables
   1.1 Understanding Collisions ........................................ 2
   1.2 The Underlying Array, Take 1 .................................. 3
   1.3 The Underlying Array, Take 2 .................................. 3

Objectives

By the end of these notes, you will know:

- How hashtables use chaining to manage collisions

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

- Implement the chaining method for collision management in hashtables

1 Implementing Hashtables

Let’s talk in a bit more detail about how to implement hashtables. We gave a very fast overview of this in the last lecture (so you could look at the homework this weekend), but now we want to set this up a bit more carefully.

First, let’s articulate the interface that we want our hash table implementation to satisfy.

```java
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 recap from last lecture: 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).

```java
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 `hashCode`es, 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 `Person`s in the hash table; this means the hash table must store a list of values at each index, not just one.

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

```java
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) {
        int index = key.hashCode() % size;
        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.

```java
public V lookup(K key, V value) {
    int index = key.hashCode() % size;
    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.

```java
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:

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

```java
/**
 * 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 KVPair<K, V>(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:

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

Writing this method is one of the tasks on this week’s 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.

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: http://cs.brown.edu/courses/cs018/feedback