CMPSCI 187: Programming With Data Structures

Lecture #36: Hashing
David Mix Barrington
7 December 2012

Hashing

- The Hashing Idea
- Hashing Assumptions
- The `Hashable` Interface
- Dealing With Collisions: Linear Probing
- Clustering and Rehashing
- Buckets and Chaining
- Hashing in Java

The Hashing Idea

- We would like to maintain a collection of items, each with a **key**, in such a way that we can add, get, or remove any item in O(1) time given the key. We saw last time that if we can afford to dedicate a memory location for each *possible* key, we can do this easily. But key spaces are normally very large, much larger than the amount of space we'd like to devote to the collection.

- The basic idea is simple -- we define a **hash function** that maps keys to **hash values**. A hash value is an index into a **hash table**, the array in which we will actually store the items. Our hope is that there will be few or no **collisions**, meaning few or no pairs of different keys in use at the same time that have the same hash value. If the number of different possible keys is greater than the size of the hash table, though, we can't avoid collisions in the worst case. We'll see later how to deal with them.

- We need the hash function to be easy to compute. We also require that it have nothing in particular to do with the meaning of the keys. DJW give an example where there is a pattern in the keys leading to many collisions.

Hashing Assumptions

- Although we can't avoid the problem of collisions, we'll make some other assumptions that are fairly realistic but will simplify our discussion considerably.
- We'll assume that our hash table never gets full. That is, no matter how large the key space is, we will only use a number of keys less than the size of the hash table. This just requires us to budget enough size for the keys we use.
- We won't worry about there being two distinct items with the same key. (This is why we have keys like SSN's and student ID's.) A concern in real databases is two records with different keys representing the same real-world thing.
- We will assume that the user will only call the `get` or `remove` commands for keys that are actually in the database. (We can make them use `contains` first.)

The `Hashable` Interface

- We will write our HashTable class to store items that are Hashable, meaning that we can compute a hash value for each object. Here the Employee class is just an example of how we implement the interface. If the keys have no particular pattern relative to the hash function, they should be fairly evenly distributed among the hash values.

```
public interface Hashable {
   int hash( );}

public class Employee implements Hashable {
   protected String name;
   protected int idNum;
   protected int yearsOfService;
   protected final int MAX_ELEMENTS = 50;
   public Employee (String name, int id, int years) {
      this.name = name; idNum = id; yearsOfService = years;}
   public int hash( ) {
      return (idNum % MAX_ELEMENTS);}
```

Dealing With Collisions: Linear Probing

- Here's a simple way to resolve collisions. If the slot where the hash function tells you to add is full, try the next, then the next, and so on until you find an empty one. To get an element, try the hash function's place first, then the succeeding places until you find it. On average, if the table is not very full, you shouldn't have to look long.

```
public static void add (Hashable element) {
   int location = element.hash( );
   while (list[location] != null) {
      location = (location + 1) % list.length;
   list[location] = element;
   numElements++;}

public static Hashable get (Hashable element) {
   int location = element.hash( );
   while (!list.location.equals(element))
      location = (location + 1) % list.length;
   return (Hashable) list[location];
```

Clustering and Rehashing

- We are assuming that gets and removes are only ever for elements that are in that table. Without that assumption, linear probing has a problem -- we will search the entire table if the item isn't there. If we *never remove*, though, we are better off -- we can give up our search when we reach a null array entry.

- Linear probing has a problem called **clustering**. If a particular area of the table gets lots of hash values, they tend to block out an expanding portion of the array, and new values only make this portion bigger. This means more repeated probes for more elements, and thus more time.

- We can avoid clustering by using a different method for reprobing after a miss. Our **rehash function** above was to just add 1, whatever the hash value. But if we use some arithmetic function of the original value and the number of rehashes (as in **quadratic probing**, which you'll probably see in CMPSCI 311) two values that hash to the same place once probably won't hash to the same place again.

Dealing With Collisions: Buckets and Chaining

- Another way to deal with collisions is to replace our array of elements with an **array of linked lists of elements**. The method is called **chaining** (for the linked lists) and the individual lists are called **buckets**.

- To add a new element with key k, for example, we go to the bucket numbered h(k) and add the element to the linked list we find there. To get the element with key k, we do a linear search of the linked list in bucket h(k). We can handle unsuccessful searches and removal of elements easily now, using the simple methods for an unsorted linked list.

- There's probably no advantage in sorting the linked list, because it is likely to be *short*. The **load factor** of the table is the number of keys stored divided by the number of buckets -- this is the *average* number of keys in a bucket. (The maximum size of a bucket is around log n if the hash function is good.) The load factor may go above 1 (unlike the linear probing case) but as long as it is small most of our linear searches will be very fast.

Hashing in Java

- The `Hashable` interface we saw earlier was a DJW creation to demonstrate the concept. In Java every object has a hash code because the class `Object` has a `hashCode` method. But like `Object`'s `toString` method, you probably don't want to use it because it just returns something derived from the object's location in memory. This means you can't count on two "equal" objects having the same hash code. So if you want to make a hash table of objects from a class of your own design, you need to write a `hashCode` method. The standard classes like `String` and `Integer` have sensible methods.

- The `Hashtable` class (which DJW misspell) keeps a collection of key-value pairs, storing the values in an array of buckets with a given start size and resizing (and rehashing) as necessary to keep the load factor around 0.75. The `Hashtable` object has a variety of methods to add and remove pairs, find values for given keys, and so forth. Variants of `Hashtable` exist that implement the useful `Set` and `Map` interfaces, which we haven't done here.