CmpSci 187: Programming with Data Structures
Spring 2015

Lecture #24, More on Selection

John Ridgway

April 28, 2015

The Selection Problem Revisited

- The selection problem can be restated in this way: given a **key**, we wish to find some value associated with it.
- Earlier in the course we frequently had the key and the value being the same object, but that is not necessary.
- There is a standard interface in `java.util` that expresses an appropriate interface.
- A (simplified) version of that interface is shown below.

The (Simplified) `Map` Interface

```
public interface Map<K,V> {
  void clear();
  boolean containsKey(K key);
  V get(K key);
  boolean isEmpty();
  V put(K key, V value);
  V remove(K key);
  int size();
}
```

Review of Searching Methods

- To solve the problem we need some way of searching for a given key.
- In order to get $O(\log n)$ time searching where we can also easily insert and delete, we looked at binary search trees. These allow insertion, deletion, and finding in $O(\log n)$ time each if they are balanced. (There are self-balancing BST's, which you will see in CmpSci 311.)
- Can we do better? Is there a data structure in which we can insert, delete, and find all in $O(1)$ time?

$O(1)$ **Searching: Dedicated Slots**

- DJW give the example of a small company where the employee ID's range from 0 to 99 and the HR department knows everyone's ID without having to look it up.
- They can keep the employee record objects in an array, and find it by using the ID itself as an index.
- The general problem of finding an object from a key is simplified if the key space is small, as in this instance.
- If we can afford an array with one index for each key, all our operations become easy.
- We test whether a key is in use by checking that key's slot for a `null` entry. We add a key by replacing a `null` entry with a real one. We delete a key by replacing that entry with a `null` one.
- The problem is, of course, that our key space is usually uncomfortably large. UMass keeps student records by eight-digit ID's, so using this system would require an array with 100,000,000 entries.
- Only a small fraction of these entries would ever be used, since fewer than a million students have ever attended UMass.
- US Social Security numbers have a key space of size 10^9, a sizable fraction of which is in use.

Introduction to Hashing

- Hashing is a technique to simulate the dedicated-slot method in the general situation where only a small fraction of the possible keys are used.
- In the general hashing situation we have a large set of keys and a smaller set of indices for our array. We choose a hash function, which takes any key and produces an index. The simplest hash function uses the integer remainder operator `%`, if we have m different indices, then for any key k the hash function produces the index `k % m`, which is in the range from 0 through $m - 1$.
- Suppose for a moment that on the subset of the keys that we use, our hash function is what CmpSci 250 will call a one-to-one function. No two different keys in use are ever mapped to the same index.
- In that case, we can use the array just as we used the dedicated-slot array, inserting, deleting, and finding elements in $O(1)$ time with an array of size m instead of the size of the key space.
- In the worst case, there is no way to avoid the possibility that two different keys, both in use, will be mapped to the same index.
- Such a failure of the hash function to be one-to-one is called a collision.

Clicker Question #1

Suppose I have the following list of strings: {`"when", in", "the", "course", "of", "human", "events"`}, and I want to store them in a hash table indexed by the letters `{a,...,z}`. Which hash function will have no collisions?

A. $a(w) =$ **first letter of** w

B. $b(w) =$ second letter of w

C. $c(w) =$ last letter of w

D. all three will have collisions

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 earlier 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.
- We talked a little bit about hashing functions in yesterday's discussion.
- If the number of different possible keys is greater than the size of the hash table, though, we probably cannot avoid collisions. 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 that leads to many collisions.

Clicker Question #2

Suppose we hashed the 220 students in this class as follows. For student s, we compute $h(s)$ by adding the ASCII values of the letters in the student's last name to get a number n, and then have $h(s) = n \ \% \ 257$. Which of these statements is true?

A. $220 < 257$, so there are no collisions

B. **if s and t have the same last name,** $h(s) = h(t)$

C. if s and t have different last names, $h(s) \neq h(t)$

D. $257 > 220$, so there must be collisions

Hashing Assumptions

We make some assumptions that are realistic but that simplify the discussion:

- The 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.
- We won't worry about there being two distinct items with the same key. (This is why we have keys like student IDs.)

Header for `HashTableMap`

```
public class HashTableMap<K,V>
    implements Map<K,V> {
  private Entry<K,V>[] slots;
  private int numElements;

  public HashTableMap(int tableSize) {
    this.slots =
      (Entry<K,V>[])(new Entry[tableSize]);
  }

  public boolean isEmpty() {
    return numElements == 0; }

  public int size() {
    return numElements; }
```

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.
- The trouble is that if you look for a key that's not there you will end up searching the entire table (or worse).

Clustering and Rehashing

- Linear probing has another 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. In the worst case you end up looking through (almost) the entire table.
- 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.
- 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 Cmp-Sci 311), then two values that hash to the same place once probably won't hash to the same place again.

Buckets and Chaining

- Another way to deal with collisions is to replace our array of key-value pairs with an array of linked lists of key-value pairs.
- The method is called **chaining** (because of the linked lists) and the individual lists are called **buckets**.
- The basic idea is that all the elements that map to a given hash address are kept in the bucket for that address.
- 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.

Code for `put`

```
public V put(K key, V value) {
  int location = key.hashCode();
  for (Entry<K,V> entry = slots[location];
       entry != null; entry = entry.getNext()) {
    if (entry.getKey().equals(key)) {
      V oldValue = entry.getValue();
      entry.setValue(value);
      return oldValue;
    }
  }
```

```
    slots[location] = new Entry<K,V>(key, value,
                                     slots[location]);
    numElements += 1;
    return null;
  }
```

Code for `get`

```
  public V get(K key) {
    int location = key.hashCode();
    Entry<K,V> entry = slots[location];
    while (entry != null) {
      if (entry.getKey().equals(key)) {
        return entry.getValue();
      }
      entry = entry.getNext();
    }
    return null;
  }
```

- 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, the average number of keys in a bucket. (The maximum size of a bucket is $O(\log n)$ 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.

Clicker Question #3

What is the worst-case time needed to add, remove, or find an element in a bucket/chain hash table of size n with $n/2$ keys in use?

A. $O(1)$

B. $O(\log n)$

C. $O(n)$

D. $O(n^2)$

Hashing in Java

- The `Map` interface we saw earlier was a somewhat simplified version of the `Map` interface in the `java.util` package.
- Java provides a `HashMap` class that implements the `Map` interface.
- It is worth mentioning that there is also a `TreeMap` class that implements the `Map` interface using a Red-Black tree, a form of self-balancing binary search tree that you will see in CmpSci 311.

- The `HashMap` class 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 below a specified value (the default is 0.75).
- The `HashMap` object has a variety of methods to add and remove pairs, find values for given keys, and so forth.
- The class depends on the `hashCode` method, and that the correspondence between `equals` and `hashCode` is maintained.
- Remember that the default implementation of `hashCode` returns a value dependent on the object's identity.
- This works with the default implementation of `equals`; two objects are equal only if they are the same object (i.e., have the same identity).
- If you change `equals` you must change `hashCode` correspondingly.
- The standard classes like `String` and `Integer` have sensible `hashCode` methods, where objects with equal values do have the same hash code.

Clicker Question #4

Suppose someone has designed a new class and has overwritten the method `public int hashCode` for it. I have a hash table of size `m`, and I use the hash function that maps an object `x` to address `x.hashCode( ) % m`. Which of these assumptions do I need for this to work?

A. Different `x`s have different `x.hashCode()`s

B. `x.hashCode()` **is never negative**

C. The `x.hashCode()` values are evenly distributed

D. All of the above