CS61B Summer 2006
Instructor: Erin Korber
Lecture 23, 3 Aug.

Hashtables

We've already looked at several ways to implement dictionaries. Today we'll look at one more, which allows us to access the value associated with any key very quickly - in constant time.

We know that to get constant-time access to a piece of data, we can use an array. But to decide what indexes to store items at, we need a way to map keys (which might be strings, or very large integers, or anything else) to array indices (integers starting at 0), and we don't want to waste a lot of space by making an array much larger than we need. The function that we need from keys to array indices will be called a *hashing* function, and its output is a *hashcode*. The hashcode of a key will tell us which "bucket" (array index) to use for the key-value pair that we are trying to insert (or where to find the key we're looking up).

1 Basic hash table structure

Suppose, for example, that we are implementing a dictionary of English words and their definitions. The keys will be the words, and the values their definitions.

Suppose n is the number of keys (words) whose definitions we want to store, and suppose we use a table of N buckets, where N is perhaps a bit larger than n, but much smaller than the number of *possible* keys. The hashing function that we use must map the entire set of possible keys to the set of integers less than N.

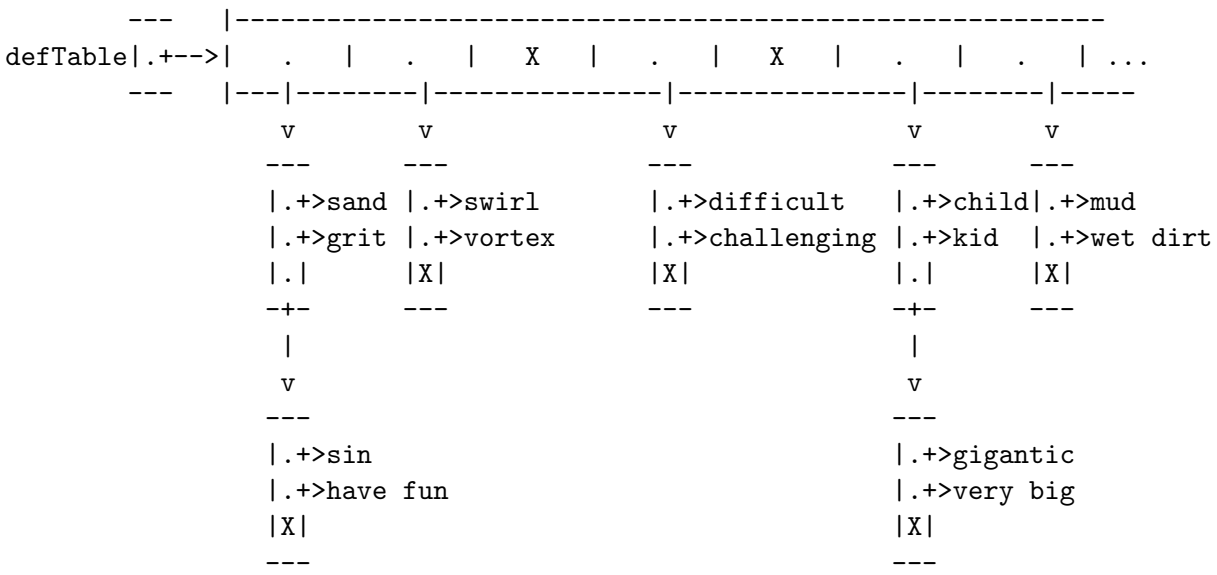
By doing this, no matter how long and variegated the keys are, we can map them into a table whose size is not much greater than the actual number of entries we want to store. However, we've created a problem: several keys could hashed to the same bucket in the table - since the space of possible keys is much larger than the number of buckets, many keys will have the same hashcode. The circumstance that arises when we have multiple keys to put in the table that have the same hashcode is called a *collision*.

How do we handle collisions without losing entries? There are several options.

2 Chaining

One is to use a simple idea called *chaining*. Instead of having each bucket in the table reference one entry, we have it reference a linked list of entries, called a *chain*. If several keys are mapped to the same bucket, their definitions all reside in that bucket's linked list. So each cell of the array contains a reference to a linked list of key-value pairs. To look up the definition of a word, we compute the hash code of the word, go to that place in the table, and search in the linked list at that spot until we find the word we want.

Here is a diagram of how such a dictionary might look - each cell references a list of key-value pairs, so each list node has a key (word), a value (definition), and a next pointer (to the next thing in that linked list).



Hash tables support the standard dictionary operations - insert, find, and remove.

- `public void insert(KeyValPair p)` Compute the key's hash code to determine the entry's bucket. Insert the key-val pair into that bucket's list.
- `public KeyValPair find(Key k)` Hash the key to determine its bucket. Search the list for an entry with the given key. If found, return the entry; otherwise, return null.
- `public KeyValPair remove(Key k)` Hash the key to determine its bucket. Search the list for an entry with the given key. Remove it

from the list if found. Return the key-val pair, if it was there, or null otherwise.

What if two entries with the same key are inserted? There are three possible approaches. We can insert both, and have `find` or `remove` arbitrarily return/remove one. In this case, it makes sense to have `findAll` and `removeAll` methods to find or remove all the entries with a given key. Another approach is to replace the old value with the new one, so only one entry with a given key exists in the table. A third is to give some kind of error (an exception, perhaps) when someone tries to insert a duplicate key. Which approach is best? It depends on the application.

WARNING: When an object is stored in a hash table, an application should never change the object in a way that will change its hash code. If you do so, the object will thenceforth be in the wrong bucket. Generally, you want hashing functions for objects to use only properties of the object that won't change over the object's lifetime.

The *load factor* of a hash table is n/N, where n is the number of keys in the table and N is the number of buckets. If the load factor stays below one (or a small constant), and the hashing function is "good," and there are no duplicate keys, then the linked lists are all short, and each operation takes O(1) time. However, if the load factor is too large (n much larger than N), performance is dominated by linked list operations and degenerates to O(n) time (albeit with a much smaller constant factor than if you replaced the hash table with one singly-linked list). On Monday, we'll analyze this more precisely with aa more sophisticated form of analysis.

3 Open Addressing

An alternative to chaining is *open addressing*, where we store only one item in each bucket, and when one bucket is full, we just use another. Note that this requires that the load factor always be < 1. There are a few ways to do this.

- Linear probing - if there is already something at hash(K), store it at hash(K) + 1; if that's full, try hash(K) + 2, etc. When finding an item, follow the same pattern (look at hash(K), if it's not there, look one slot over, etc.) Wrap around to the beginning if you fall off the end of the table.

- Quadratic probing - choose a constant m. Then if there's a collision at hash(K), try hash(K) + m, then hash(K) + m^2, then hash(K) +

m^3, etc.

- Double hashing - come up with a second hashing function hash', then if there's a collision at hash(K), try hash(K) + hash'(K), then hash(K) + 2*hash'(K), etc.

Open addressing hash tables are slightly more space efficient than chaining ones, but their operations run slower, and they are more complex to implement. The only time the memory savings might be significant would be if the data we were storing in the hashtable was just a primitive, in which case we could use open addressing to avoid creating any objects to put in linked lists. However, generally we're storing key-value pairs, and furthermore, the value itself is often a fairly complex object, so we already have a lot of objects around.

4 Hashing functions

Hashing functions are a bit of a black art. The ideal hashing function would map each key to a uniformly distributed random bucket from zero to N-1. (By "random", I don't mean that the function is different each time; a given key always hashes to the same bucket. I mean that two different keys, however similar, will hash to independently chosen integers, so the probability they'll collide is 1/N.) This ideal is tricky to obtain.

In practice, it's easy to mess up and create far more collisions than necessary. Let's consider bad functions first. Suppose the keys are integers, and each integer's hash code is just itself mod N, so hashCode(i) = i mod N. Suppose that our table size is 10,000.

Now, suppose for some reason that our application only ever generates keys that are divisible by 4. A number divisible by 4 mod 10,000 is still a number divisible by 4, so three-quarters of the buckets are never used! We have at least four times as many collisions as necessary.

The same compression function is much better if N is prime. With N prime, even if the hash codes are always divisible by 4, numbers larger than N often hash to buckets not divisible by 4, so all the buckets can be used. Modding by primes means that we will still get a good distribution of codes, even if the input data happens to have weird divisibility properties (like all of the keys being even, or divisible by 5, or some such thing).

For reasons I won't fully explain (look it up if you're interested),
h(i) = ((a * i + b) mod p) mod N

is a yet better hashing function for integers. Here, a, b, and p are positive integers, p is a large prime, and p is much greater than N.

Since hash codes often need to be designed specially for each new object, you're left to your own wits. Here is an example of a good hash code for Strings.

```
private static int hash(String key) {
  int hashVal = 0;
  for (int i = 0; i < key.length(); i++) {
    hashVal = (127 * hashVal + key.charAt(i)) % 16908799;
  }
  return hashVal;
}
```

By multiplying the hash code by 127 before adding in each new character, we make sure that each character has a different effect on the final result. The mod operator with a prime number tends to "mix up the bits" of the hash code. The prime is chosen to be large, but not so large that 127 * `hashVal` + `key.charAt(i)` will ever exceed the maximum possible value of an int.

As you can see, coming up with good hash codes is fairly mysterious. The best way to understand good hash codes is to understand why bad hash codes are bad. Here are some examples of bad hash codes on words.

1. Sum up the ASCII values of the characters. Unfortunately, the sum will rarely exceed 500 or so, and most of the entries will be bunched up in a few hundred buckets. Moreover, anagrams like "pat," "tap," and "apt" will collide.

2. Use the first three letters of a word, in a table with 26^3 buckets. Unfortunately, words beginning with "pre" are much more common than words beginning with "xzq", and the former will be bunched up in one long list. This does not approach our uniformly distributed ideal.

3. Consider the "good" hash function written out above. Suppose the prime modulus is 127 instead of 16908799. Then the return value is just the last character of the word, because (127 * hashVal) That's why 127 and 16908799 were chosen to have no common factors.

Why is the `hash` function presented above good? Because we can find no obvious flaws, and it seems to work well in practice. (A black art indeed.)

5 Resizing Hash Tables

Sometimes we can't predict in advance how many entries we'll need to store. If the load factor n/N (entries per bucket) gets too large, we are in danger of losing constant-time performance.

One option is to enlarge the hash table when the load factor becomes too large (typically larger than 0.75). Allocate a new array (typically at least twice as long as the old), then walk through all the entries in the old array and *rehash* them into the new.

Take note: you CANNOT just copy the linked lists to the same buckets in the new array, because then you are not even going to use the new buckets you created! You will want to adjust the hashing function to properly accomodat the new size, so you have to rehash each entry individually.

You can also shrink hash tables (e.g., when $n/N < 0.25$) to free memory, if you think memory will benefit something else. (In practice, it's only sometimes worth the effort.)

Obviously, an operation that causes a hash table to resize itself will take more than O(1) time; nevertheless, the average over the long run is still O(1) time per operation (as we'll see later).

6 Why not hash tables?

Although hash tables excel at finding a particular key, they are terrible for finding the closest key to a given one, or all the keys in a particular range, or the largest or smallest key in a table, since the hash of a key does not tell us anything about the key's size or the hash of other keys closer to it. Therefore, to find, e.g. the minimum key in a hash table, we simply have to scan the entire thing once, keeping track of the lowest found so far. Similarly, to find all keys in a particular range, we have to scan the whole table, adding each key that we find that falls in the range to the set that we output.

7 Comparing Search Structures

Here's a table of the speeds of the operations on the various search structures that we've looked at . In the table below, n is the number of items and k is the number of answers to a range query. The search tree and hash table are assumed to be "good".

Function	Unord. List	Sorted Array	Search Tree	Hashtable	Heap
find	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(n)$
insert	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$
range query	$\Theta(n)$	$\Theta(k + \log n)$	$\Theta(k + \log n)$	$\Theta(n)$	$\Theta(n)$
find min	$\Theta(n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(1)$
remove min	$\Theta(n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(\log n)$