CSE 326 Autumn 2006
Section 7 Notes

1 Randomized Hashing

Note that you are not responsible for knowing the randomized hashing strategy presented in section. It is presented here merely for your own information. While this scheme gives good performance, it is much more complicated to implement than the probing strategies. With that warning, if you wish to use randomized hashing in Project 3, or in any future project, you'll need to know a few more implementation details.

In section, we saw that with a good hashing function, you can guarantee a constant number of collisions in each hash table position. I stated without proof that for any fixed hashing function, you can always find a worst case sequence of items which will hash to the same position. Therefore, an optimal hashing strategy must involve randomness, but in a careful way. If we hash every item to a truly random location, we encounter the "Where did I put it?" problem. For a good reference, the interested reader is referred to Kleinberg and Tardos, Section 13.6, pages 734-741.

Since almost no standard programming library uses randomized hashing, you'll need to write your own. In that case, you can make the table size a prime number, and we will assume this for the remainder of the notes. Also, the original model I presented assumed you were hashing numbers from a small universe, which is not very useful in practice. A natural question is "Can we generalize randomized hashing for strings or arbitrary objects?"

In these notes, I'll first present the randomized hashing model, then I'll give algorithms and pseudocode implementation for efficiently hashing strings and generic objects and finding prime numbers.

2 Dictionary ADT and Hashing Model

The dictionary abstract data type has an interface which is a subset of search tree methods. It does not maintain order, and instead focuses on fast insertion, deletion, and lookup for a dynamic collection. As with search trees, the *values* which are manipulated by dictionaries have an associated *key* which is usually much smaller and serves as a kind of "name tag" for the value.

Initialize(size): Create an empty dictionary and optionally preallocate space for the given number of entries.

Insert(key, value): Insert the given value by its associated key.

Lookup(key): Return the value associated with the given key if it exists in the dictionary.

Remove(key): Removes the given key and its associated value from the dictionary.

In general, the key is a separate object from the value. Since a dictionary should be able to store any type(s) of values generically, we can only impose constraints on the keys.

We assume that all keys are strings of exactly r symbols, where each r is taken from an alphabet Σ. In the simple case of integer keys in binary form, $\Sigma = \{0, 1\}$, and r is the maximum number of bits needed to store any number. Shorter numbers are simply padded with 0's in their higher-order bits. For an arbitrary type, you will need to pad shorter keys with a special non-key symbol up to the maximum length of r symbols.

We will call the *universe* of all possible keys U, which is usually very large $|U| = O(|\Sigma|^r)$. Usually we only want to store a small subset $S \subseteq U$ of keys, where we denote the size of this subset $n = |S|$. At first, we treat n as a constant that we know beforehand, but in reality, we may not know how many keys we will need to store over time. Therefore, we will add a special case to deal with *resizing* a dictionary if it gets too full.

Most importantly, how can we implement this ADT efficiently?

2.1 Bit Array

Imagine that we store an array that has $|U|$ locations to represent all possible keys. In each location, we store a 1 if that key is in the dictionary, otherwise we store a 0. Insertion, lookup, and removal are all $O(1)$-time operations, but we use $\Omega(2^n)$ space.

2.2 Linked List

To improve on the space, we store only the keys that are in the dictionary in a linked list, one node per key. This only uses $O(n)$ space, but now all operations are $O(n)$-time in the worst case.

2.3 Hash Table

It is easy to see that a dictionary will always need $\Omega(n)$ space, but can we achieve this along with something better than linear time? The answer is yes, using the most popular implementation of the dictionary ADT, a hash table. Our goal is to get the $O(1)$-time operations of a bit array with the $O(n)$ space of the linked list.

A hash table is an array of p locations, which is greater than or equal to n but of the same order, $p = \Theta(n)$. A *hash function* maps a key from the universe to a hash table location efficiently (in $O(1)$-time):

$$h : U \to \{0, \ldots, p-1\} \tag{1}$$

In an optimal hash table, the probability of collision is $\frac{1}{p}$, and without going into too much math detail, this gives you $O(1)$ expected collisions. Therefore, if your hash function is good enough, you can just use separate chaining or preallocate an array for colliding keys at each hash table location (since the average number of collisions is constant). No probing is needed. Note that randomized hashing could have worst case $O(n)$ running time, just like probing, but it is extremely unlikely because of the randomized strategy.

So what makes a good hash function, and how do we implement it?

3 Randomized Hashing for General Objects

Using a result from number theory without proof, taking products modulo a prime number gives a good, "even" distribution which will help us avoid hash table collisions. Therefore, we take p to be a prime greater than n, for example, $n \leq p \leq 2n$. Technically, p must also be greater than the size of your alphabet $|\Sigma|$, but the alphabet is usually small (e.g. Roman letters) compared to the number of keys you wish to store (e.g. English words). In section I called the size of the hashtable m which may be different from p, but in practice you should set $m = p$.

3.1 Randomized Hash Functions

Randomized hash functions need to be initialized with random values which then remain fixed until rehashing and are stored with the function so that we can lookup or remove keys after inserting them.

For integer keys, choose two numbers a and b uniformly at random from $\{1, \ldots, p-1\}$. Then your hash function is simply:

$$h_{ab}(x) = (ax + b) \bmod p \tag{2}$$

You must keep the parameters a and b with the hash function, which takes $O(1)$ space.

As stated above, any key can be treated as a string x over an alphabet Σ padded up to r symbols ($x = (x_1, \ldots, x_r)$ and $x_i \in \Sigma$). We can convert this key string to a sequence of numbers $y = (y_1, \ldots, y_r)$ where $y_i \in \{1, \ldots, p-1\}$. This conversion takes the form of a table σ which maps each of $|\Sigma|$ symbols to a random number from $\{1, \ldots, p-1\}$, where no two symbols are mapped to the same number.

$$\sigma : \Sigma \to \{1, \ldots, p-1\} \tag{3}$$
$$y = (\sigma(x_1), \ldots, \sigma(x_r)) \tag{4}$$

Now we must modify our hash function. Instead of choosing random numbers a and b, we choose a random vector $a = (a_1, \ldots, a_r)$, where each a_i is taken uniformly at random from $\{1, \ldots, p\}$. You must store the vector a and the table σ with your hash function, which takes $O(r)$-space. With respect to n, we can treat r as a constant, so this is still $O(1)$ space and takes $O(1)$ time.

$$h_{a,\sigma}(x) = \left[\sum_{i=1}^{r} a_i \sigma(x_i) \right] \mod p \tag{5}$$

Because we must be able to deal with the rare case of collisions, we need to store the original key along with the value in the given hash table location.

Hash Value / Table Index	**Original Key**	**Value**
h(x)	x	...

To make this more concrete, we provided the following example.

3.2 English Example

Suppose you wanted to hash a string containing English words in lowercase. Call the maximum length of any word in Roman letters as r. You will need an alphabet which contains all lowercase Roman letters, as well as any punctuation you might need (like a hyphen or space for compound words) and a special padding symbol that doesn't appear in any word (for example, \$).

$$\Sigma = \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \ldots, \mathrm{z}, -, \$\} \tag{6}$$

Let's assume $|\Sigma| = 30$, that we want to store about $n = 500$ words, and the maximum length of any word is $r = 12$. Let's choose the prime number greater than n to be $p = 521$. For each string, we convert it to a key $y = (y_1, \ldots, y_r)$ where $y_i \in \{1, \ldots, 520\}$. Each y_i can only have one of 30 values out of 520 possibilities, so we choose a random but fixed mapping of string symbols to some non-zero number less than p:

For our example, we can choose something like this:

Key Symbol	**Number**
a	415
b	23
c	127
d	222
e	67
f	444
...	...
z	2
-	386
\$	509

Then the word "deadbeef" is padded up to 12 letters and then mapped to a sequence of 12 numbers:

$$\text{deadbeef} = (222, 67, 415, 222, 23, 67, 67, 444, 509, 509, 509, 509) \tag{7}$$

Let's choose the random vector $a = (179, 273, 236, 487, 363, 376, 441, 431, 165, 250, 384, 26)$. Our hash function becomes:

$$h_{a,\sigma}(x) = 179\sigma(x_1) + 273\sigma(x_2) + 236\sigma(x_3) + 487\sigma(x_4) + 363\sigma(x_5) + 376\sigma(x_6) + \quad (8)$$
$$441\sigma(x_7) + 431\sigma(x_8) + 165\sigma(x_9) + 250\sigma(x_{10}) + 384\sigma(x_{11}) + 26\sigma(x_{12}) \bmod 521 \quad (9)$$

The actual computation of the hash value for the key "deadbeef" is:

$$h_{a,\sigma}(\text{“}deadbeef\text{''}) = 179 \cdot 222 + 273 \cdot 67 + 236 \cdot 415 + 487 \cdot 222 + 363 \cdot 23 + 376 \cdot 67 + \quad (10)$$
$$441 \cdot 67 + 431 \cdot 444 + 165 \cdot 509 + 250 \cdot 509 + 384 \cdot 509 + 26 \cdot 509) \bmod 521 \quad (11)$$

This is equal to 139, so you would store the key "deadbeef" and its associated value in the hash table at location 139. The remaining piece of implementation is finding a prime number efficiently.

4 Finding a Prime Number

You can find a prime number through brute force by testing numbers in sequence from n up to $2n$. For each number i, search all of its factors up to $\sqrt{i}$. This approach takes $O(n^{1.5})$-time. Amortized over $O(n)$ insertions, the expected time between rehashes, this is still an extra $O(n^{0.5})$.

However, a simple randomized prime finder using the *Fermat primality test* returns a prime number in constant expected time. We use Fermat's little theorem, which states that if p is a prime, then for all numbers i between 1 and p:

$$i^{p-1} = 1 \bmod p \quad (12)$$

Using a constant number of random samples, we can determine if a number is prime with high probability. By looping from n up to $2n$, we can apply this test to each number until we find one that passes. Because the above primality test depends on modular exponentiation, we need a fast algorithm for the general problem of $a^b \bmod n$. The following pseudocode runs in $O(\log^3 n)$ time.

```
FastModExp(a,b,n):
  d = 1
  e = a
  while (b != 0) {
    if ((b & 0x1) == 0x1) {
      d = d*e % n
    }
    e = e*e % n
    b = b >> 1
  }
  return d
```

The following pseudocode uses fast modular exponentiation and the Fermat test to return the next prime number (with high probability) greater than the given size. The number of random samples below is 10, but this number is an arbitrary constant. You can choose any sufficiently large number.

```
GetNextPrime(n)
  current = n
  composite = true
  while (composite) {
    current++
    for (i = 0; i < 10; i++) {
      a = rand() % current
```

```
    b = FastModExp(a, current-1, current)
    if (b != 1) {
      break
    }
    composite = false
  }
}
return current
```

Due to the Prime Number Theorem, we know that there are about $n \log n$ prime numbers less than or equal to any integer n. This means the density of primes is $1/\log n$, or the average number of consecutive integers after n we have to examine before finding a prime. Therefore, on average we can expect the above procedure to return after searching the integers between n and $n + \log n$. Since it loops at most $O(\log n)$ times, and the constant number of modular exponentiation takes at most $O(\log^3(n + \log n)) = O(\log^3 n)$ time, the total running time is $O(\log^4 n)$. Amortized over n insertions, this is $O(\log^4 n/n)$, which is $o(1)$. This means it is sub-constant or goes to zero asymptotically.

5 Final Notes

For the *static* hashing problem, you know all your n items and the number n itself before hashing, and they never change after that. Most of the time, however, you want to solve the *dynamic* hashing problem where n can grow over time unpredictably. Like the probing hashing schemes, we have to handle the case when the hash table becomes full.

Also, there are some details specific to Java that are helpful to know with respect to using the built-in hashing and also writing your own.

5.1 Rehashing

The randomized hashing function $h_{a,\sigma}(x)$ given above has a collision probability of $1/n$, which is the best that any hashing function can do. Over $n = p$ insertions (the size of the hash table), you will expect to get no collisions. However, if the table becomes more than full ($p \leq n \leq kp, k > 1$), you will begin to have $O(k)$ expected collisions. This in itself is not a problem since you can just preallocated $O(k)$ spaces at each hash table location and search through them as in separate chaining. You can choose k as a fixed constant in your implementation, usually something small like 2 or 3.

However, if n keeps increasing over time it will eventually exceed kp, which will increase your chances of getting $k + 1$ collisions. In this case, you will need to rehash. This is similar to creating an empty hash table: you must find a prime number, allocate space, randomly choose a hash and mapping function, and store these with the new table. However, when rehashing a table you should probably double the number of keys you want to store ($n' \geq 2n$) and you also need to rehash all the old items with the new hash function.

5.2 Java Details

In Java, all objects have an inherited `hashCode()` method from the `Object` class. This default method computes a hash based on the object's address in the virtual machine heap. The details are not public, and there is no guarantee that the hash function gives a good distribution over all address locations.[1]

In general, you should override the `hashCode()` method in any object that will be a hashtable key. In Java, keys can be objects of any type, usually separate from the value objects. Also in Java 1.5, HashMaps now have generic support so you can enforce a particular type for all keys and all values. Anytime you override `hashCode()`, you should also override `equals` to maintain the contract: `a.equals(b)` implies `a.hashCode()`

[1] Although Sun Microsystems recently opened the source of Java under the GNU Public License, so now we will finally know!

`== b.hashCode`. However, the converse is not true; two objects with the same hash code do not necessarily have to be equal.

Using the built-in `HashMap` class rounds your hashtable size to the nearest power-of-two and calls its `hashCode` method. The `Hashtable` class lets you specify arbitrary sizes (for example, a prime number). The problem with using these classes directly is that you cannot store extra information about the hashing function parameters. Also, you don't want your classes to depend on a specific hashing functions, but each class must define a mapping of itself to a sequence of integers. One possible design for hashing is as follows:

1. **Create a global parent for all of your other classes:** Create two methods that returns a sequence of integers as a hash value: `int[] hashNumbers(int p)` and `int[] recomputeHash(int p)`. The first method will be called by your hash table class, which will pass in its prime number size. The global parent class should keep track of the last `p` that it has seen and also cache its sequence of hash numbers once instead of recomputing it every time. If it receives a different `p`, it should recompute the hash numbers by calling `recomputeHash`, which should be defined in all implementing subclasses. Be sure `recomputeHash` satisfies `equals` contract above.

2. **Create a custom hash table class:** You will want to implement all the methods in the `Map` interface, but you shouldn't subclass `Hashtable` or `HashMap` because it does not give you enough control of the internal representation. Create an initialization function that chooses a random vector a and finds the next prime number given an initial size. This should be called from the constructor and any time you want to rehash. Write your `put()` method to call `hashNumbers` on key class, passing in the prime number size. The returned sequence of numbers should be multiplied with your random vector a modulo p and then stored in the corresponding internal `Hashtable` with an integer key. Add a check of the number of items; if it is greater than the prime number size times your chosen constant, rehash.

3. **Use your custom hash table for your custom classes**.

This completes our construction of a randomized hashing scheme.