CI583: Data Structures and Operating Systems
Implementing hashtables – prime numbers and hash functions

A note on prime numbers

Prime numbers, those which no number divides evenly except 1 and the number itself, are an essential part of many algorithms.

We often need to generate a new prime number, and some algorithms call for very large ones. Unfortunately, most methods for testing primality are $O(2^n)$.

A naive way to test primality:

```
1  boolean isPrime(int n) {
2    for(int i=2; i<n; i++) {
3      if(n % i == 0) return false;
4    }
5    return true;
6  }
```

A note on prime numbers

In fact, we don't need to loop all the way to n: no number greater than $n/2$ will divide n.

```
1  boolean isPrime(int n) {
2    for(int i=2; i<n/2; i++) {
3      if(n % i == 0) return false;
4    }
5    return true;
6  }
```

A note on prime numbers

Thinking about it more closely, we only need to loop up to $\sqrt{n}$ since, if we make a list of the factors of a number, the square root will be in the middle:

```
for(int i=2; i*i<n; i++) {
```

Prime numbers

As a last improvement, we should notice that 2 is a weird prime – it is the only one which is even. So, we can test whether 2 divides n then only check odd numbers:

```
1  boolean isPrime(int n) {
2    if(n<= 2) return true;
3    if(n % 2 == 0) return false;
4    for(int i=3; i*i<n; i+=2) {
5      if(n % i == 0) return false;
6    }
7    return true;
8  }
```

This is still exponential but is a big improvement on what we started with.

Implementing a hash function

So, most hash functions generate a large unique number based on the key, and mod that with the size of the array. To hash a string, we might start out with a method like this:

```
1  int hash1(String key) {
2    int hashVal = 0;
3    int pow27 = 1;//1, 27, 27*27, etc
4    for(int i=key.length()-1; i>=0; i--) {
5      int c = key.charAt(i) - 96;
6      hashVal += pow27 * c;
7      pow27 *= 27;
8    }
9    return hashVal % arraySize;
10 }
```

Implementing a hash function

Note the presence of the constants 27 and 96. Why?

```
1  int hash1(String key) {
2    int hashVal = 0;
3    int pow27 = 1;//1, 27, 27*27, etc
4    for(int i=key.length()-1; i>=0; i--) {
5      int c = key.charAt(i) - 96;
6      hashVal += pow27 * c;
7      pow27 *= 27;
8    }
9    return hashVal % arraySize;
10 }
```

Implementing a hash function

We can improve on this. The first insight is to use Horner's Method: see (p41, Cormen).

This formula allows us to transform a *monomial* (polynomial with only one term) expression into a computationally efficient form:

$$vn^4 + wn^3 + xn^2 + yn^1 + zn^0 = (((vn + w)n + x)n + y)n + z.$$

Implementing a hash function

Applying this insight we rewrite the method, this time starting with the leftmost character:

```
1  int hash2(String key) {
2    int hashVal = key.charAt(0) - 96;
3    for(int i=0; i<key.length(); i++) {
4      int c = key.charAt(i) - 96;
5      hashVal = hashVal * 27 + c;
6    }
7    return hashVal % arraySize;
8  }
```

Implementing a hash function

This will overflow a standard numeric type for strings longer than about seven chars.

We can, however, apply the modulus operation inside the loop, keeping the size of `hashVal` down:

```
1  int hash3(String key) {
2    int hashVal = key.charAt(0) - 96;
3    for(int i=0; i<key.length(); i++) {
4      int c = key.charAt(i) - 96;
5      hashVal = (hashVal * 27 + c) % arraySize;
6    }
7    return hashVal;
8  }
```

Implementing a hash function

Some variation on this approach is used to hash strings.

The function needs to be as efficient as it can be, so wherever possible we might use tricks such as bitwise arithmetic (eg $\gg$) instead of %.

To enable this, we might pick a base which is a power of 2, e.g. 32 instead of 27.

Implementing a hash function

We can also manipulate the keys before hashing, compressing them and removing non-data.

For example a set of catalogue numbers might be of the form `nn-nnn-nnnn-nn`, where the last two digits are a *checksum* (add together the rest of the number and mod by 100, say).

In this case we can drop the last two digits before hashing.

Implementing a hash function

If the keys are not randomly distributed (e.g., in our catalogue example, digits 6 to 10 might represent a date) then it's important that the table size is prime.

If many keys happen to share a divisor with the table size, they may tend to have the same hash value and cause clustering.

If the table size is prime, no keys will divide evenly into it.

Summary

If open addressing is used, double hashing is the best approach.

In choosing between open addressing and separate chaining, go for separate chaining *if* you don't know in advance how much data you will need to store.

Table sizes should be prime numbers.

Hash functions should be efficient and designed, as far as possible, to produce a uniform distribution of values.