Hash Functions and Hash Tables

hash function a function that can take a key value and compute an integer value (or an index in a table) from it

For example, student records for a class could be stored in an array `C` of dimension 10000 by truncating the student's ID number to its last four digits:

H(IDNum) = IDNum % 10000

Given an ID number `X`, the corresponding record would be inserted at `C[H(X)]`.

This would be easy to implement, and cheap to execute. Whether it's actually a very good hash function is another matter…

Hash Functions

Suppose we have N records, and a table of M slots, where $N \leq M$.

- there are M^N different ways to map the records into the table, if we don't worry about mapping two records to the same slot
- the number of different *perfect* mappings of the records into different slots in the table would be

$$P(M, N) = \frac{M!}{(M-N)!}$$

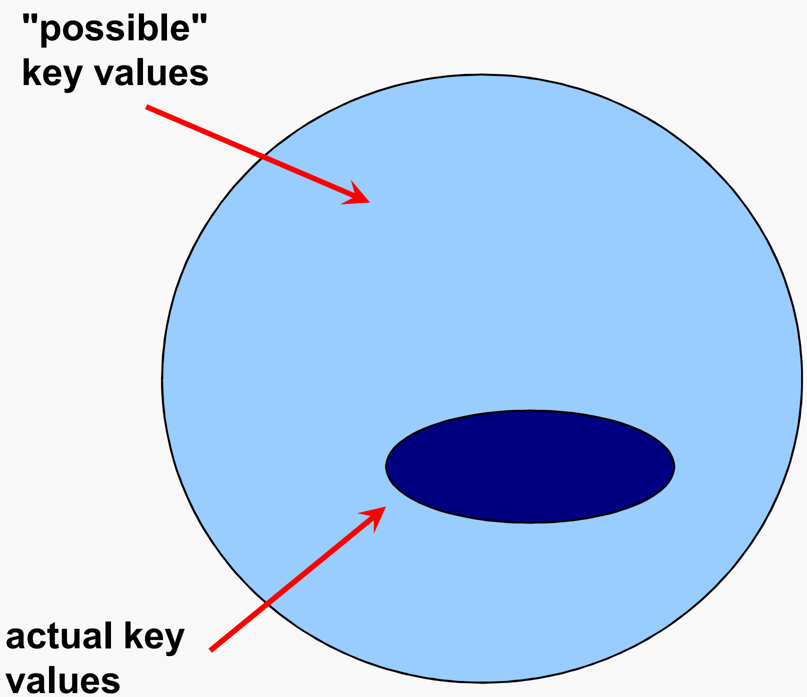
- for instance, if N = 50 and M = 100, there are 10^{100} different possible hash mappings, "only" 10^{94} of which are perfect (1 in 1,000,000)
- so, there is no shortage of potential perfect hash functions (in theory)
- however, we need one that is effectively computable, that is, it must be possible to compute it (so we need a formula for it) and it must be efficiently computable
- there are a number of common approaches, but the design of good, practical hash functions must still be considered a topic of research and experiment

Hash Function Domain Issues

The set of logically possible key values may be very large.

- set of possible Java identifiers of length 10 or less (xxx)

The set of key values we actually encounter when compiling a program will be much smaller, but we don't know which values we'll actually see until we see them…

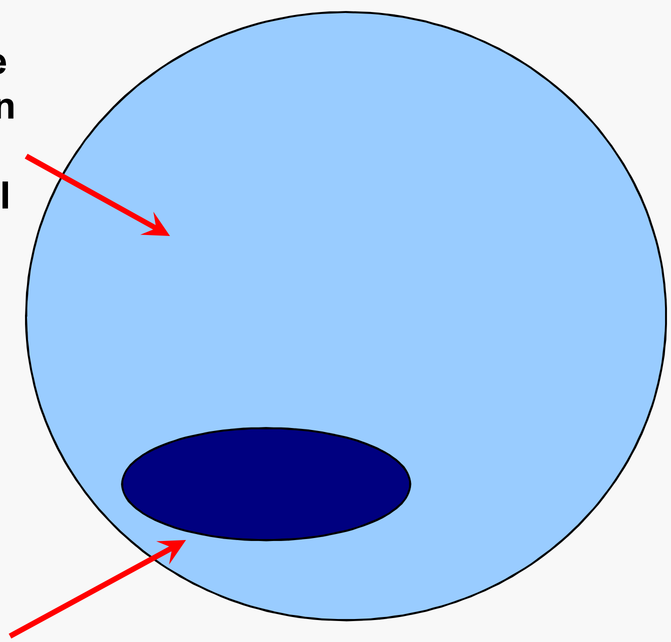


Hash Function Domain Issues

The ideal is a one-to-one hash function… good luck with that:

- take a reasonable table size for hashing the identifiers in a Java program
- consider the number of possible Java identifiers
- both sets are finite and the second is much, much larger

F() may be uniform on the whole theoretical domain…



…but not at all uniform on the small subset of it that we actually get.

So, the next best thing would be a hash function that is "uniform".

That is, we'd like to map about the same number of domain values to each slot in the table… good luck with that too…

Simple Hash Example

It is usually desirable to have the entire key value affect the hash result (so simply chopping off the last k digits of an integer key is NOT a good idea in most cases).

Consider the following function to hash a string value into an integer range:

```
public static int sumOfChars(String toHash) {

   int hashValue = 0;
   for (int Pos = 0; Pos < toHash.length(); Pos++) {
      hashValue = hashValue + toHash.charAt(Pos);
   }
   return hashValue;
}
```

```
Hashing: hash
     h: 104
     a:  97
     s: 115
     h: 104
Sum:     420

Mod by table
size to get the
index
```

This takes every element of the string into account… a string hash function that truncated to the last three characters would compute the same integer for "hash", "stash", "mash", "trash“.

Hash Function Techniques

Division

- the first order of business for a hash function is to compute an integer value
- if we expect the hash function to produce a valid index for our chosen table size, that integer will probably be out of range
- that is easily remedied by modding the integer by the table size
- there is some reason to believe that it is better if the table size is a prime, or at least has no small prime factors

Folding

- portions of the key are often recombined, or *folded* together
- *shift folding*: 123-45-6789 → 123 + 456 + 789
- *boundary folding*: 123-45-6789 → 123 + 654 + 789
- can be efficiently performed using bitwise operations
- the characters of a string can be xor'd together, but small numbers result
- "chunks" of characters can be xor'd instead, say in integer-sized chunks

Hash Function Techniques

Mid-square function

- square the key, then use the middle part as the result
- e.g., 3121 → 9740641 → 406 (with a table size of 1000)
- a string would first be transformed into a number, say by folding
- idea is to let all of the key influence the result
- if table size is a power of 2, this can be done efficiently at the bit level:

  3121 → 100101001010000101100001 → 0101000010 (with a table size of 1024)

Extraction

- use only part of the key to compute the result
- motivation may be related to the distribution of the actual key values, e.g., VT student IDs almost all begin with 904, so it would contribute no useful separation

Radix transformation

- change the base-of-representation of the numeric key, mod by table size
- not much of a rationale for it…

Hash Function Design

A good hash function should:

- be easy and quick to compute
- achieve an even distribution of the key values that actually occur across the index range supported by the table
- ideally be mathematically one-to-one on the set of relevant key values

Note: hash functions are NOT random in any sense.

Improving Scattering

A simple hash function is likely to map two or more key values to the same integer value, in at least some cases.

A little bit of design forethought can often reduce this:

```
public static int sumOfShiftedChars(String toHash) {

   int hashValue = 0;
   for (int Pos = 0; Pos < toHash.length(); Pos++) {
      hashValue = (hashValue << 4) + toHash.charAt(Pos);
   }
   return hashValue;
}
```

```
Hashing: hash
      h: 104
      a:  97
      s: 115
      h: 104
Sum:  452760
```

The original version would have hashed both of these strings to the same table index.

Flaw: it didn't take element position into account.

```
Hashing: shah
      s: 115
      h: 104
      a:  97
      h: 104
Sum:  499320
```

A Classic Hash Function for Strings

Consider the following function to hash a string value into an integer:

```
public static long elfHash(String toHash) {

   long hashValue = 0;
   for (int Pos = 0; Pos < toHash.length(); Pos++) {      // use all elements

      hashValue = (hashValue << 4) + toHash.charAt(Pos);  // shift/mix

      long hiBits = hashValue & 0xF000000000000000;       // get high nybble

      if (hiBits != 0)
         hashValue ^= hiBits >> 24;    // xor high nybble with second nybble

      hashValue &= ~hiBits;            // clear high nybble
   }

   return hashValue;
}
```

This was developed originally during the design of the UNIX operating system, for use in building system-level hash tables.

Details

Here's a trace (using 32-bit integers):

```
 Character   hashValue
-----------------------
     d: 64   00000064
     i: 69   000006a9
     s: 73   00006b03
     t: 74   0006b0a4
     r: 72   006b0ab2
     i: 69   06b0ab89
     b: 62   0b0ab892
     u: 75   00ab8925
     t: 74   0ab892c4
     i: 69   0b892c09
     o: 6f   0892c04f
     n: 6e   092c05de

distribution: 15388030
```

```
hashValue      : 06b0ab89
hashValue << 4: 6b0ab890
add 62         : 6b0ab8f2

hiBits         : 60000000
hiBits >> 24   : 00000060

hashValue ^      6b0ab8f2
   hiBits        00000060
               : 6b0ab892

hashValue &
   ~hiBits     : 0b0ab892
```

```
f: 1111
6: 0110
^: 1001
```