Chapter 43: Hash Functions

Section 43.1: Hash codes for common types in C#

The hash codes produced by `GetHashCode()` method for built-in and common C# types from the `System` namespace are shown below.

Boolean

1 if value is true, 0 otherwise.

Byte, UInt16, Int32, UInt32, Single

Value (if necessary casted to Int32).

SByte

```
((int)m_value ^ (int)m_value << 8);
```

Char

```
(int)m_value ^ ((int)m_value << 16);
```

Int16

```
((int)((ushort)m_value) ^ (((int)m_value) << 16));
```

Int64, Double

Xor between lower and upper 32 bits of 64 bit number

```
(unchecked((int)((long)m_value)) ^ (int)(m_value >> 32));
```

UInt64, DateTime, TimeSpan

```
((int)m_value) ^ (int)(m_value >> 32);
```

Decimal

```
((((int *)&dbl)[0]) & 0xFFFFFFF0) ^ ((int *)&dbl)[1];
```

Object

```
RuntimeHelpers.GetHashCode(this);
```

The default implementation is used sync block index.

String

Hash code computation depends on the platform type (Win32 or Win64), feature of using randomized string hashing, Debug / Release mode. In case of Win64 platform:

```
int hash1 = 5381;
int hash2 = hash1;
int c;
char *s = src;
while ((c = s[0]) != 0) {
    hash1 = ((hash1 << 5) + hash1) ^ c;
    c = s[1];
    if (c == 0)
        break;
    hash2 = ((hash2 << 5) + hash2) ^ c;
    s += 2;
}
```

```
return hash1 + (hash2 * 1566083941);
```

ValueType

The first non-static field is look for and get it's hashcode. If the type has no non-static fields, the hashcode of the type returns. The hashcode of a static member can't be taken because if that member is of the same type as the original type, the calculating ends up in an infinite loop.

Nullable<T>

```
return hasValue ? value.GetHashCode() : 0;
```

Array

```
int ret = 0;
for (int i = (Length >= 8 ? Length - 8 : 0); i < Length; i++)
{
    ret = ((ret << 5) + ret) ^ comparer.GetHashCode(GetValue(i));
}
```

References

- GitHub .Net Core CLR

Section 43.2: Introduction to hash functions

Hash function `h()` is an arbitrary function which mapped data $x \in X$ of arbitrary size to value $y \in Y$ of fixed size: $y = h(x)$. Good hash functions have follows restrictions:

- hash functions behave likes uniform distribution
- hash functions is deterministic. `h(x)` should always return the same value for a given `x`
- fast calculating (has runtime O(1))

In general case size of hash function less then size of input data: $|y| < |x|$. Hash functions are not reversible or in other words it may be collision: $\exists\ x1, x2 \in X, x1 \neq x2: h(x1) = h(x2)$. X may be finite or infinite set and Y is finite set.

Hash functions are used in a lot of parts of computer science, for example in software engineering, cryptography, databases, networks, machine learning and so on. There are many different types of hash functions, with differing domain specific properties.

Often hash is an integer value. There are special methods in programmning languages for hash calculating. For example, in `C#` `GetHashCode()` method for all types returns `Int32` value (32 bit integer number). In `Java` every class provides `hashCode()` method which return **`int`**. Each data type has own or user defined implementations.

Hash methods

There are several approaches for determinig hash function. Without loss of generality, lets $x \in X = \{z \in \mathbb{Z}: z \geq 0\}$ are positive integer numbers. Often `m` is prime (not too close to an exact power of 2).

Method	Hash function
Division method	$h(x) = x \bmod m$
Multiplication method	$h(x) = \lfloor m\ (xA \bmod 1) \rfloor,\ A \in \{z \in \mathbb{R}: 0 < z < 1\}$

Hash table

Hash functions used in hash tables for computing index into an array of slots. Hash table is data structure for implementing dictionaries (key-value structure). Good implemented hash tables have O(1) time for the next operations: insert, search and delete data by key. More than one keys may hash to the same slot. There are two ways for resolving collision:

1. Chaining: linked list is used for storing elements with the same hash value in slot
2. Open addressing: zero or one element is stored in each slot

The next methods are used to compute the probe sequences required for open addressing

Method	Formula
Linear probing	`h(x, i) = (h'(x) + i) mod m`
Quadratic probing	`h(x, i) = (h'(x) + c1*i + c2*i^2) mod m`
Double hashing	`h(x, i) = (h1(x) + i*h2(x)) mod m`

Where `i ∊ {0, 1, ..., m-1}`, `h'(x), h1(x), h2(x)` are auxiliary hash functions, `c1, c2` are positive auxiliary constants.

Examples

Lets `x ∊ U{1, 1000}, h = x mod m`. The next table shows the hash values in case of not prime and prime. Bolded text indicates the same hash values.

x	m = 100 (not prime)	m = 101 (prime)
723	23	16
103	3	2
738	38	31
292	92	90
61	61	61
87	87	87
995	95	86
549	49	44
991	91	82
757	**57**	50
920	20	11
626	26	20
557	**57**	52
831	31	23
619	19	13

Links

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms.
- Overview of Hash Tables
- Wolfram MathWorld - Hash Function