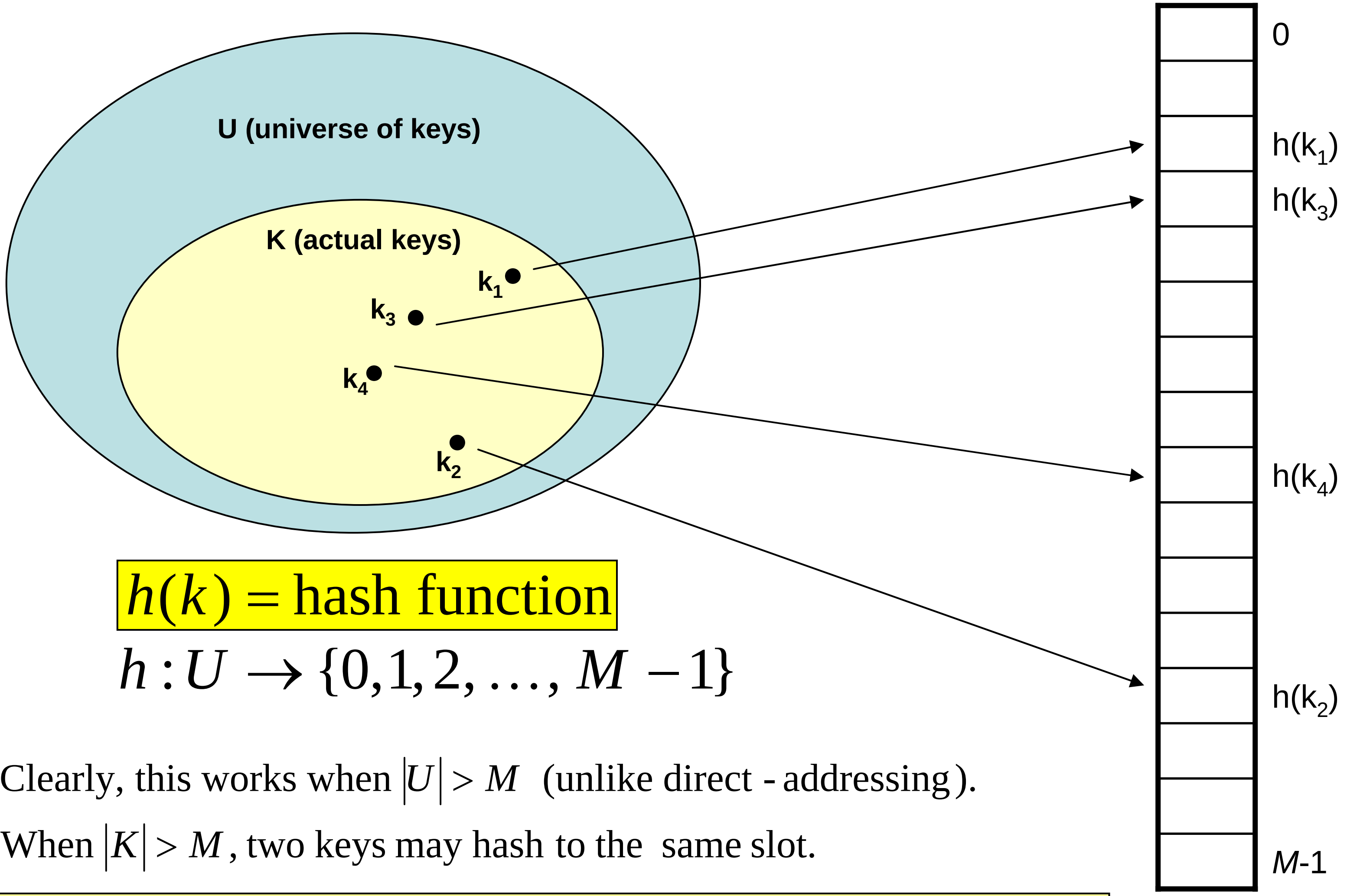
Bucknell University
Computer Science

CSCI 311 - Data Structures

Hash Functions

Last Time: Hash Table



$$\boxed{h(k) = \text{hash function}}$$

$$h : U \rightarrow \{0, 1, 2, \ldots, M-1\}$$

Clearly, this works when $|U| > M$ (unlike direct-addressing).

When $|K| > M$, two keys may hash to the same slot.

Ideally, *h* scrambles the key values well enough so that each slot is equally likely. When more than one key hashes to the same slot, we have ***collisions***.

Hash Functions

The data type of the key determines the hash function one needs.

Example: The keys are strings of characters. Assume characters use 7-bit encoding. We treat the key as a base 128 number, that is, each character in the key will correspond to one base-128 digit.

$$\text{"now"} = x = 110 \cdot 128^2 + 111 \cdot 128^1 + 119 \cdot 128^0$$

where 110, 111, and 119, respectively, are the ASCII encodings of “n”, “o”, and “w”.

How do we now hash this number x? If we say $\boxed{h(k) = x \bmod M,}$ where M=64, the slot in the table will be determined only by the last 6 bits in x. **A good hash function should consider all the bits in the key, especially when the keys are strings of characters.**

Hash Functions

	ASCII 21-bit decimal	M=64	M=31
now	1816567	55	29
tip	1914096	50	20
ilk	1734251	43	18
dim	1651949	45	21
tag	1913063	39	22
nob	1816546	34	8
sob	1898466	34	6
hut	1719028	52	16
ace	1602021	37	3
bet	1618676	52	11
jot	1751028	52	24
egg	1668071	39	23
gig	1701095	39	1
men	1798894	46	26
cab	1634530	34	24

Hashing Strings

When dealing with keys that take more than a 32-bit word, there are two issues to deal with: First, we want to compute the hash function in a reasonable amount of time (linear in the string length). Second, we must deal with large numbers that might cause overflow.

To deal with the first issue, we apply Horner’s method. The string:

$$c_k c_{k-1} c_{k-2} \ldots c_2 c_1 c_0$$

is equal to the integer:

$$c_k \times 128^k + c_{k-1} \times 128^{k-1} + c_{k-2} \times 128^{k-2} + \ldots + c_2 \times 128^2 + c_1 \times 128 + c_0$$

This can also be written in the following form, which allows the value to be computed using one multiply and one addition for each character:

$$((\ldots((c_k \times 128 + c_{k-1}) \times 128 + c_{k-2}) \times 128 + \ldots + c_2) \times 128 + c_1) \times 128 + c_0$$

Hashing Strings

Note that if we try to compute the value

$$[((\ldots(c_k\times 128 + c_{k-1})\times 128 + \ \ldots + c_2)\times 128 + c_1)\times 128 + c_0] \bmod M$$

we will run into overflow problems. However, using properties of the mod function, we can compute this value correctly without overflow as follows:

$$[([([(\ldots[(c_k\times 128 + c_{k-1}) \bmod M]\times 128 + \ldots + c_2) \bmod M]$$
$$\times 128 + c_1) \bmod M]\times 128 + c_0)] \bmod M$$

Improving String Hashing

- The hash function must produce an integer less than *M* and should consider all the bits in the key value.
- We choose *M* to be prime so as spread the hashed key values evenly in the range [0, M-1].
- When we look at keys as numbers in a **base-*R*** system, if the table size *M* and *R* have common factors (or sometimes even when they don't), the distribution of hashed values may be far from even.
- What we need is to pick *M* and *R* values that are **relatively prime**.

```
int hash(String v, int M) {
  int h = 0, a = 127;
  for (int i = 0; i < length[v]; i++)
    h = (a*h + v[i]) % M;
  return h;
}
```

Note that we picked *R*=127, which is prime and will therefore be relatively prime to the chosen table size *M* if it is also prime.