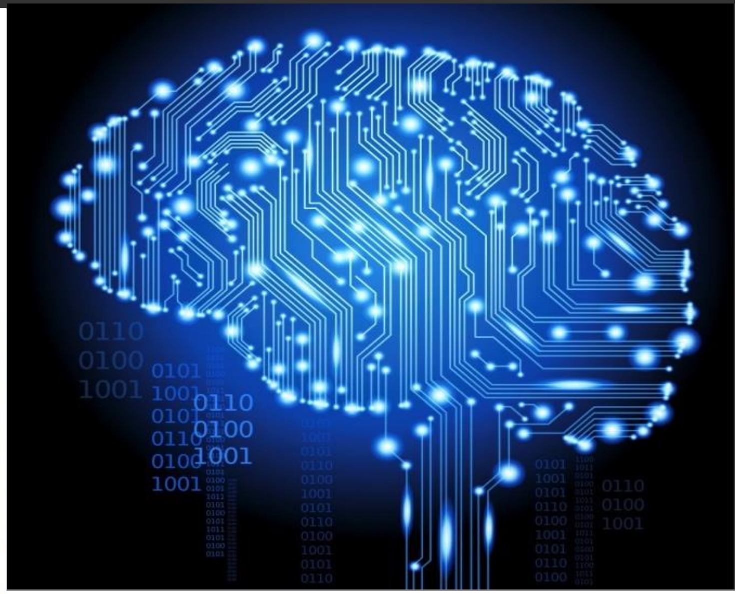
Information Technology

Hash Tables I

Prepared by Maria Garcia de la Banda
Updated by Brendon Taylor

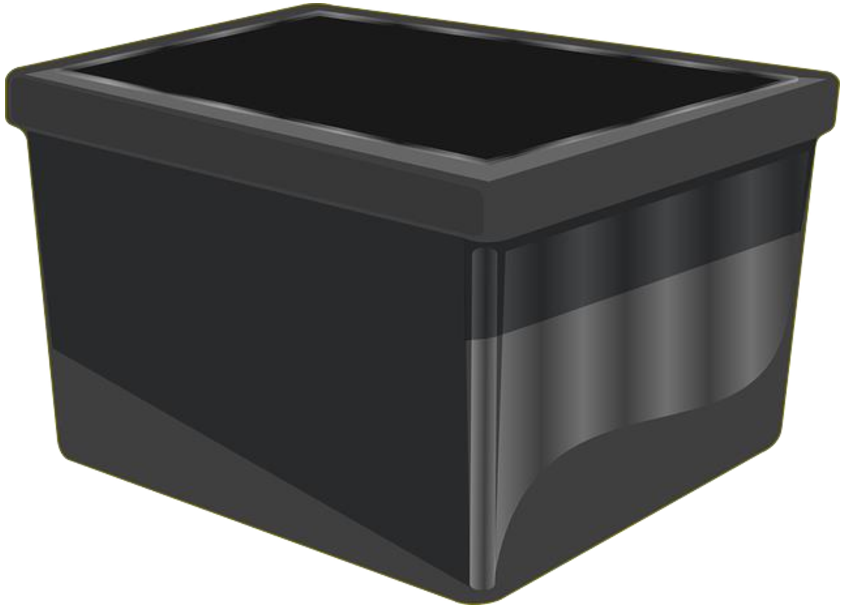


Objectives for this lesson

- **To understand what is expected from a Hash Table**
- **To understand**
  - What is a hash function
  - The properties of a good hash function
- **To be able to implement simple hash functions**

Container ADTs

- **Store and remove items independent of contents**
- **Examples include:**
  - List ADT
  - Stack ADT
  - Queue ADT
- **Core operations:**
  - Add
  - Delete
  - Search (for lists)



Dictionary ADT

Dictionary ADT

Aren't elements in array containers identified by their cell position?

They are accessed by the position, but it does not really "identify" them; which is why we can shuffle, swap, etc.

- **Yet another kind of container type to store objects**
- **Main difference: objects are uniquely identified by a label or key**
- **Defined in Python either using curly brackets or calling `dict()`**
- **Accessed with the usual square brackets (as lists)**

key

value

```
>>> x={"Name":"Peter", "Age":5}
>>> x["Name"]
'Peter'
>>> x["Age"]
5
>>> x
{'Name': 'Peter', 'Age': 5}
>>>
```

These are search operations

Due to its definition of `__str__`

Operations for Dictionary ADTs

- **Operations:**
  - Search (already seen with `x["Name"]`)
  - Add
  - Delete
  - Update

```
>>> x["Age"]= 6
>>> x["Age"]
6
>>> x["Class"]= "1A"
>>> x
{'Name': 'Peter', 'Age': 6, 'Class': '1A'}
>>> del x["Name"]
>>> x
{'Age': 6, 'Class': '1A'}
```

Update, since `x` was `{"Name":"Peter", "Age":5}`

Add, since it is a new key

Delete

Updates and additions look very similar. How do I know when it is one or the other?

Remember, keys are unique. So, if the key already appears in the dictionary: update. If not, add.

Another dictionary example

```
>>> x = dict()
>>> x[1152]="Maria"
>>> x[4563]="Julian"
>>> x[1324]="Pierre"
>>> x
{1152: 'Maria', 4563: 'Julian', 1324: 'Pierre'}
>>> x[132]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 132
>>> x[1324]
'Pierre'
>>>
```

Add

search

- **Python dictionaries are implemented using hash tables**

Motivation behind Hash Tables

Hash Tables: Motivation

- **Assume we want to store a very significant amount of data (a big N)**
- **Assume we will need to perform the following operations relatively often:**
  - Search for an item
  - Add a new item (or update its data)
  - You might also want to delete an item (optional)
- **But we do not need to traverse them in a particular order or sort them (at least not often)**
  - Traversing all is fine as long as the order is irrelevant
  - If you often need to traverse in a particular order, use a different ADT
- **What data types have we seen suitable for this?**

Data types we have seen in depth:

- **Stacks: follow LIFO**
  – Therefore, not suitable for searching/deleting
- **Queues: follow FIFO**
  – Therefore, not suitable for searching/deleting
- **Unsorted Lists (N = len(list)):**
  – Search: $O(N * Comp_{eq})$ worst in linked list and array
  – Add: $O(1)$ worst in linked list (first element) and arrays (last element)
  – Delete: $O(N * Comp_{eq})$ worst in linked lists and arrays
- **Sorted Lists (N = len(list)):**
  – Search: $O(N * Comp_{>eq})$ worst in linked lists; $O(\log N)$ in array
  – Add: $O(N * Comp_{>eq})$ worst in linked lists and arrays
  – Delete: $O(N * Comp_{>eq})$ worst in linked lists and arrays

We do not talk about best here because in this case it is more the exception than the rule

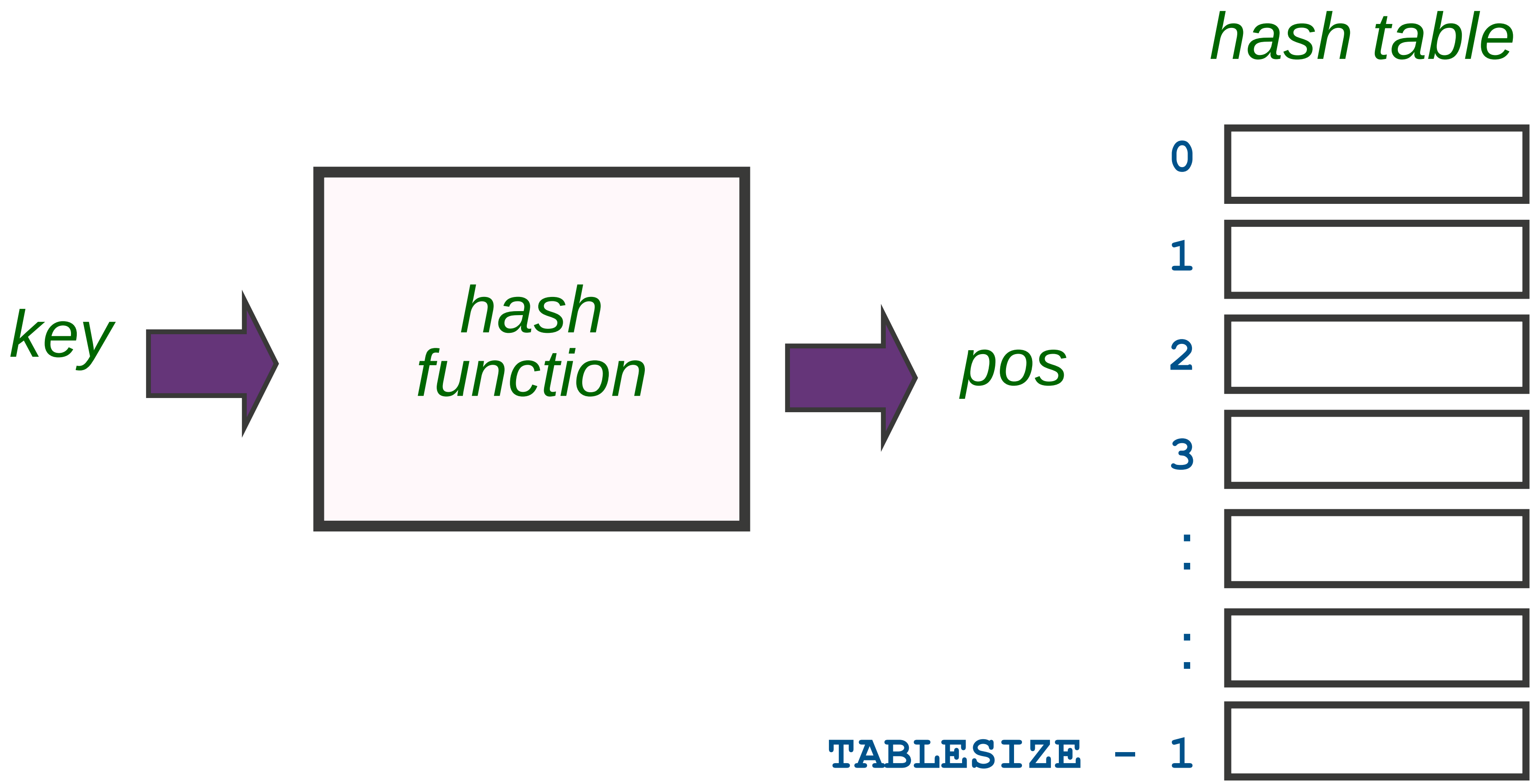
Hash Tables: aim

- **Can we go further? Can we have O(1) for adding, searching and deleting?**
- **This is what Hash Tables promise:**
  - Constant time operations (O(1) is the expected;
  - Worst case can still O(N) if not careful – need to construct the hash table well
- **How?**
  - Using arrays: constant time access to a given position
  - But this means, each item must have an assigned position

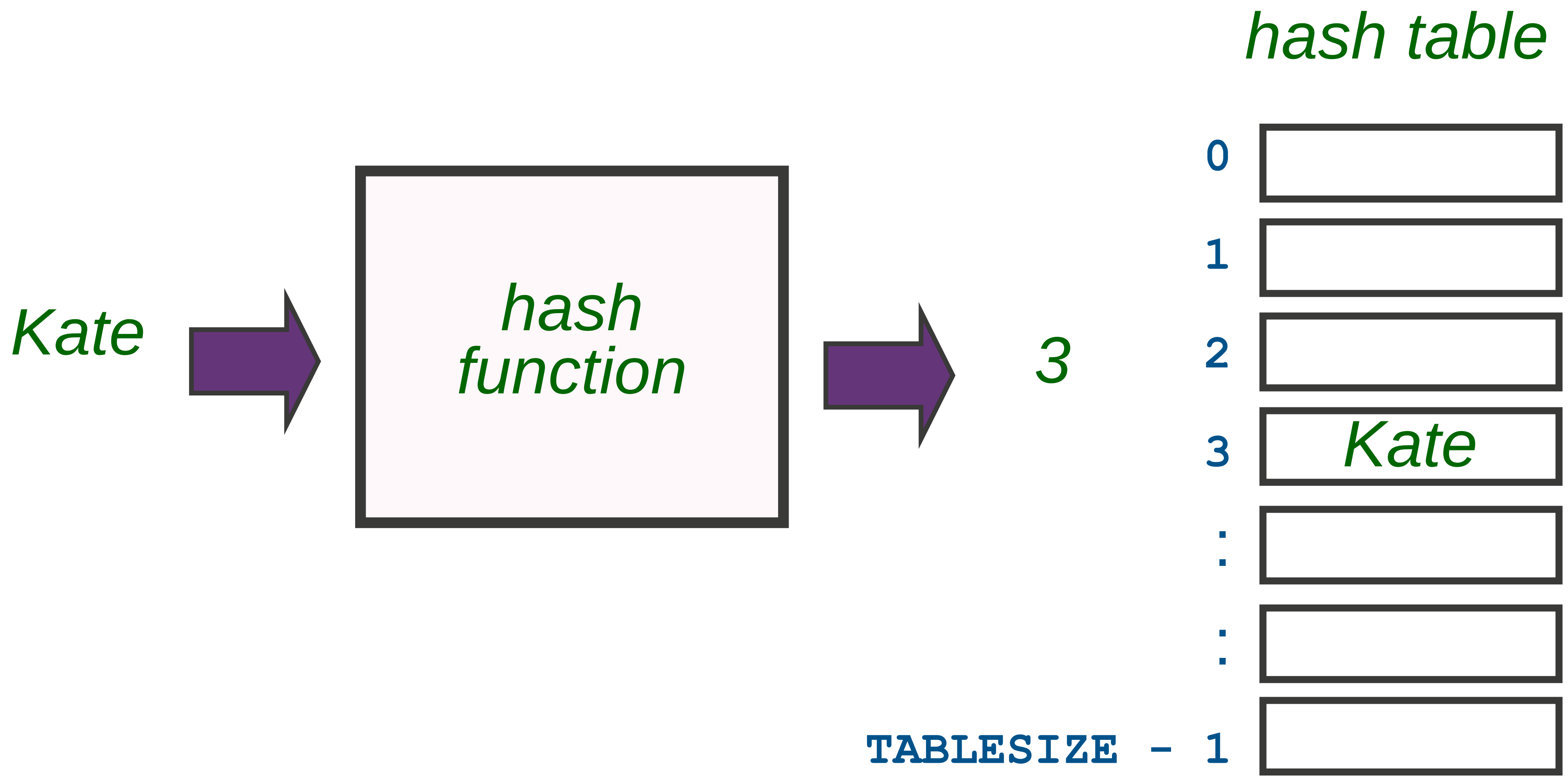
Hash Table Data Type

- **Data :**
  - Items to be stored
  - Each item must have a unique key (symbolic like “Kate”, numeric, etc)
- **Data Structure to implement the Hash Table:**
  - Large array (also referred to as the Hash Table)
- **Basic operations:**
  - Hash Function: maps a unique key to an array position
  - Add
  - Search
  - Delete

Overview



Overview: example



Hash Function

Hash Function's properties

- **Basic properties:**
  - Type dependent: depends on the type of the item's key
  - Return value within array's range (0 .. TABLESIZE-1)
- **Desirable:**
  - Fast, a slow hash function will degrade performance
    - So, should not have too many arithmetic operations
  - Minimises collisions (two keys mapped to same position)
    - Maps every key into a different array position (Perfect hash)
- **Perfect hash functions are rare:**
  - Rely on very particular properties of the keys
- **Good functions approximate random functions**
  - Chance of a collision is 1/TABLESIZE (Universal hash)

How to define Hash Functions?

- **If the key is an integer randomly distributed:**
  - Position = key % TABLESIZE is random and fast
- **Often it is not, and then what?**
- **Consider the key** 033-400-03-94-530 **where:**
  - 033: Supplier number (1..999, currently up to 70)
  - 400: Category code (100,150,200, 250, up to 850)
  - 03: Month of introduction (1..12)
  - 94: Year of introduction (00 to 99)
  - 530: Checksum (sum of other fields module 100)
- **First observation: don't use non-data**
  - Modify the key until all bits count:
    - Checksum should not be considered & category codes should be changed to 0..15

How to define Hash Functions?

- **Consider the key is a words of up to ten letters**
- **One possibility:**
  - Convert each character into a number (0..25)
  - Add the first two characters to obtain the array position
- **Example:**
  - maria → 12 + 0 = 12
  - bernd → 1 + 4 = 5
  - malena → 12 + 0 = 12
- **Not a great hash function: all words starting with the same two characters go to the same array position**
- **Second observation:**
  - The more elements (characters, digits, etc) in the key you use, the better the hash function (in terms of collisions)
  - Careful though: considering all might be too slow

How to define Hash Functions?

- **Consider again the key is a word of up to ten letters**
- **Another possibility:**
  – Convert each character into a number (0..25)
  – Add all of them to obtain the array position
- **Example:**
  – maria → 12 + 0 + 17 + 8 + 0 = 37
  – bernd → 1 + 4 + 17 + 13 + 3 = 38
  – malena → 12 + 0 + 11 + 4 + 13 + 0 = 40
- **Smallest positon: word a → 0 = 0**
- **Biggest: word zzzzzzzzzz → 10*25= 250**
- **But we have about 50,000 words in our dictionary!**
- **Many collisions: each array position is the hash key for 200 words!**
  – Which words?
  – Anagrams since position is disregarded

How to define Hash Functions?

- **Third observation:**
  - Using all elements is not enough to guarantee a good spread of possible indices
  - Need something that uses all elements & takes into account its position in the key
- **Where have we seen this before…?**
  - Think about the relationship between binary strings and decimal numbers

	2^3	2^2	2^1	2^0
	8	4	2	1
9	1	0	0	1
8	1	0	0	0
6	0	1	1	0
5	0	1	0	1

2^p where p is the position

Decimal numbers

Binary numbers

Base 2

Decimal numbers can be seen as the keys for (binary) words, that is, words made of two characters: 1 and 0

How to define Hash Functions?

- **Consider again a key of up to ten letters**
- **Another possibility (base 26):**
  - Convert each character into a number (0..25)
  - Multiply it by 26^p where p is the character position
  - Add them to obtain the array position:

array position

character code

$$h = a_0x^n + a_1x^{n-1} + \ldots + a_{n-1}x + a_n$$

Base

character position

- **Example:**
  - maria → $12*26^4+0*26^3+17*26^2+8*26^1+0*26^0 = 5{,}495{,}412$
  - zzzzzzzzzz is greater than $26^9 > 5{,}000{,}000{,}000{,}000$

Greater than 2^{32}

- **Good discrimination: unique position per word**
- **Might exceed the capability of our table (or overflow our index)**
- **Too big for our 50,000 words: lots of empty positions**

How to define Hash Functions?

- **Fourth observation:**
  - We want something in the range of our TABLESIZE
- **Possible solution:**
  - If the number is too big: use % TABLESIZE
  - If it is too small: convert to 0..1 and * TABLESIZE
  - But careful, we might have overflow/underflow!
    - Not in Python (arbitrary precision)
    - In our example: base 26 number could overflow in Java
- **Possible solution: mod/multiply at each step**

array position

character code

Character position

  - Since the key h = $a_0x^n + \dots + a_{n-3}x^3 + a_{n-2}x^2 + a_{n-1}x + a_n$
  - Equivalent to h = $((\dots(a_0x + a_1)x + \dots + a_{n-3})x + a_{n-2})x + a_{n-1})x + a_n$
  - And then, at each step we mod by TABLESIZE
  - This is Horner's idea

Same as $ax^2 + bx = (ax+b)x$

How to define Hash Functions?

- **Horner's method:**
  - Recall $h = ((\ldots(a_0x + a_1)x+\ldots+a_{n-3})x + a_{n-2})x + a_{n-1})x + a_n$
  - Mods at each step, thus casting out multiples of TABLESIZE
  - Assume 101 is our TABLESIZE (yes, too small, but good to visualise some pitfalls)

```
def hash(word: str) -> int:
    value = 0
    for char in word:
        value = (value*31 + ord(char)) % 101

    return value
```

tablesize

base

`ord('a')` returns the ASCII integer value of character a

  - Why do we use 31 rather than 26? We will see later

How to define Hash Functions?

$h = ((\ \dots(a_0x + a_1)x + \dots + a_{n-3})x + a_{n-2})x + a_{n-1})x + a_n$

- **Consider the word "Aho"**

```
value = (31*value + ord(char)) % 101
```

'A' = 65   'h' = 104   'o' = 111

```
value = 0
value = (31 * 0 + 65) % 101 = 65
value = (31 * 65 + 104) % 101 = 99
value = (31 * 99 + 111) % 101 = 49
```

$65*(31^2) + 104*(31^1) + 111 = 65800$

$65800 \% 101 = 49$

same result as modding once

How to define Hash Functions?

5*3 = 15 -> 5
5*4 = 20 -> 0
5*5 = 25 -> 5
5*6 = 30 -> 0

- **We said, if the key is randomly distributed:**
  - Position = key % TABLESIZE is random and fast
- **If the key is not random: use a prime table size (from a pre-computed table – closest to the actual size you need)**
  - If many values and TABLESIZE share common factors they will hash to the same position. Consider TABLESIZE=10:
    - Extreme example: if all keys finish in 0, then all are hashed to 0.
    - Even if not all finish in 0: any key X = 5*Y with the same value for Y%2 is hashed to the same position (only discriminates even/odd).
  - Primes avoid this: this is the reason to chose 101 as TABLESIZE
- **If you are multiplying by another constant and moding:**
  - Make sure they are relatively prime/co-prime (no common factors)
  - This is the reason to choose 31 as the base (rather than 26)

Prime table size **and** prime base will lead to spread out values…

How to define Hash Functions?

```
value = (1024*value + ord(char)) % 128
```

Key	*Hash Value*
Aho	111
Kruse	101
Standish	104
Horowitz	122
Langsam	109
Sedgewick	107
Knuth	104

Why do both Standish and Knuth map to 104?

Having common factors is likely to result in keys with close related values (clustering)

Even with the same value (collisions)

Effect of common factors

- **Let's see in detail how it works for the word "Aho"**

```
value = (1024*value + ord(char)) % 128
```

'A' = 65 'h' = 104 'o' = 111

```
value = 0
value = (1024 *  0 +  65) % 128 = 65
value = (1024 * 65 + 104) % 128 = 104
value = (1024 * 104+ 111) % 128 = 111
```

Since 1024 = 8*128, anything multiplied by 1024 is cast out when we mod by 128. This means only the last character is kept at each mod step.

Standish and Knuth have the same last character.

How to define Hash Functions?

```
value = (31*value + ord(char)) % 101
```

Key	Hash Value
Aho	49
Kruse	95
Standish	60
Horowitz	28
Langsam	21
Sedgewick	24
Knuth	44

since 31 and 101 are prime, they result in a “sparse” table

How to define Hash Functions?

```
value = (3*value + ord(char)) % 7
```

Key	*Hash Value*
Aho	0
Kruse	5
Standish	1
Horowitz	5
Langsam	5
Sedgewick	2
Knuth	1

A small TABLESIZE also leads to collisions

How to define Hash Functions?

- **Even more effective than selecting a single coefficient, like 31:**
  - Choose your coefficients (pseudo) randomly
  - Use a different coefficient for each position
- **For our string hash, a universal hash function is:**

```
def hash(word: str, TABLESIZE: int) -> int:
    value = 0
    a = 31415
    b = 27183
    for char in word:
        value = (ord(char) + a*value) % TABLESIZE
        a = a * b % (TABLESIZE-1)

    return value
```

Base changes for each position pseudo randomly

Hash Functions properties (recap)

- **Type dependent**
- **Must return value within array' s range**
- **Should minimise collisions (each position equally likely)**
  - Don't use non-data
  - Use all elements (or a reasonable subset – odd/even positions)
  - Use the position of each element
  - Avoid common factors if moding
- **Should be fast:**
  - So, should not have too many arithmetic operations
  - Still, it will be linear in the length of the element in the key
- **And of course, it must be a function!**
  - Always return the same value for the same input

Summary

- **Motivation: what is a hash table data type and why is it needed**
- **Hash Functions**
  - Definition
  - Properties
  - How to define them
- **Perfect hash functions**
- **Universal hash functions**