Martin is getting the projector to work with his laptop.

Hashed Collections

You Can (and sometimes should) Build Your Own

Hashed Collections

It slices, it dices...

Martin McClure

Andrés Valloud


Andrés
Dale
James
Norm

Linear search O(n)

Linear search O(n)
Binary search O(lg n)

Linear search O(n)

Binary search O(lg n)

O(1) ?

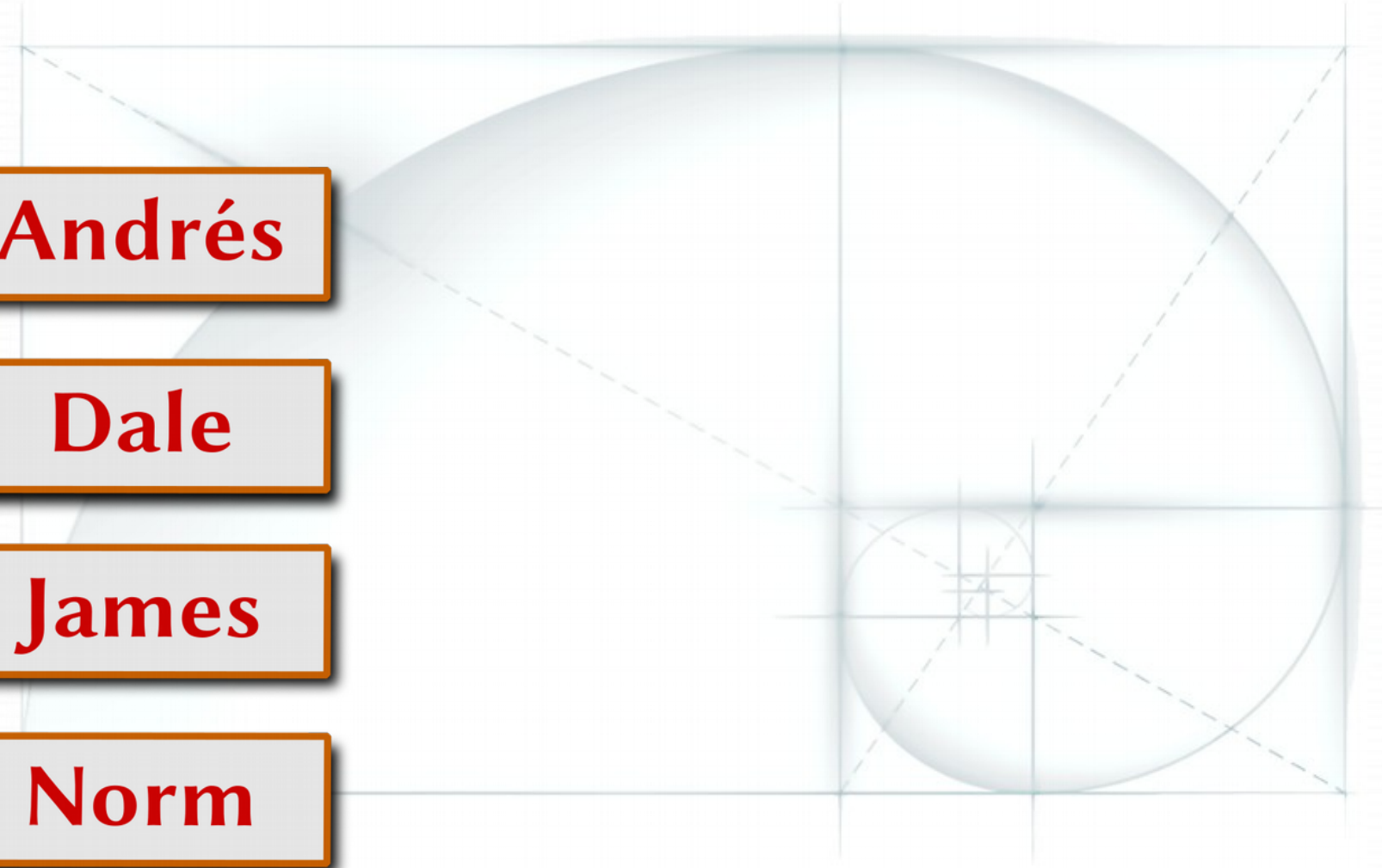
What is a Hash Table?

Andrés

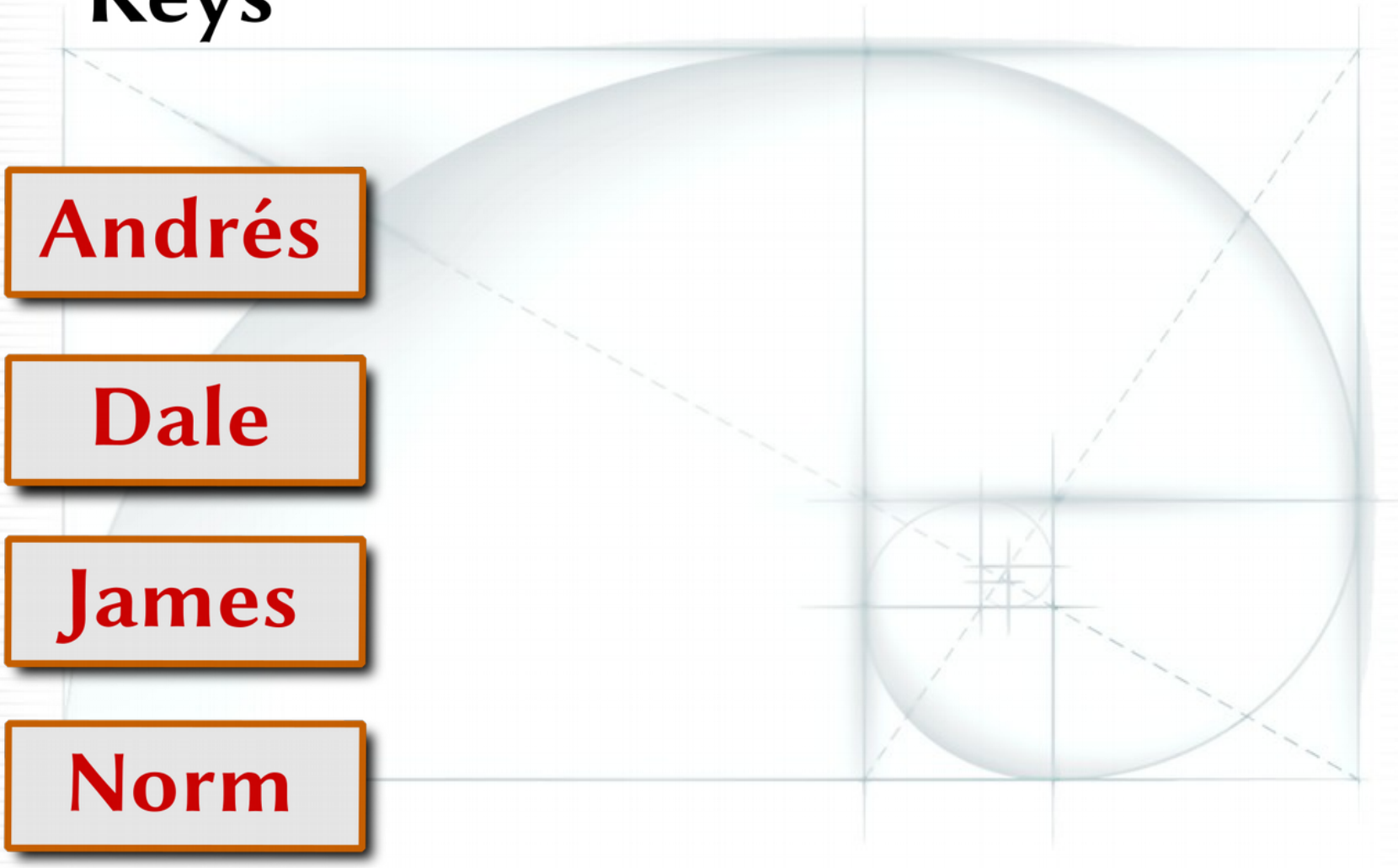
Dale

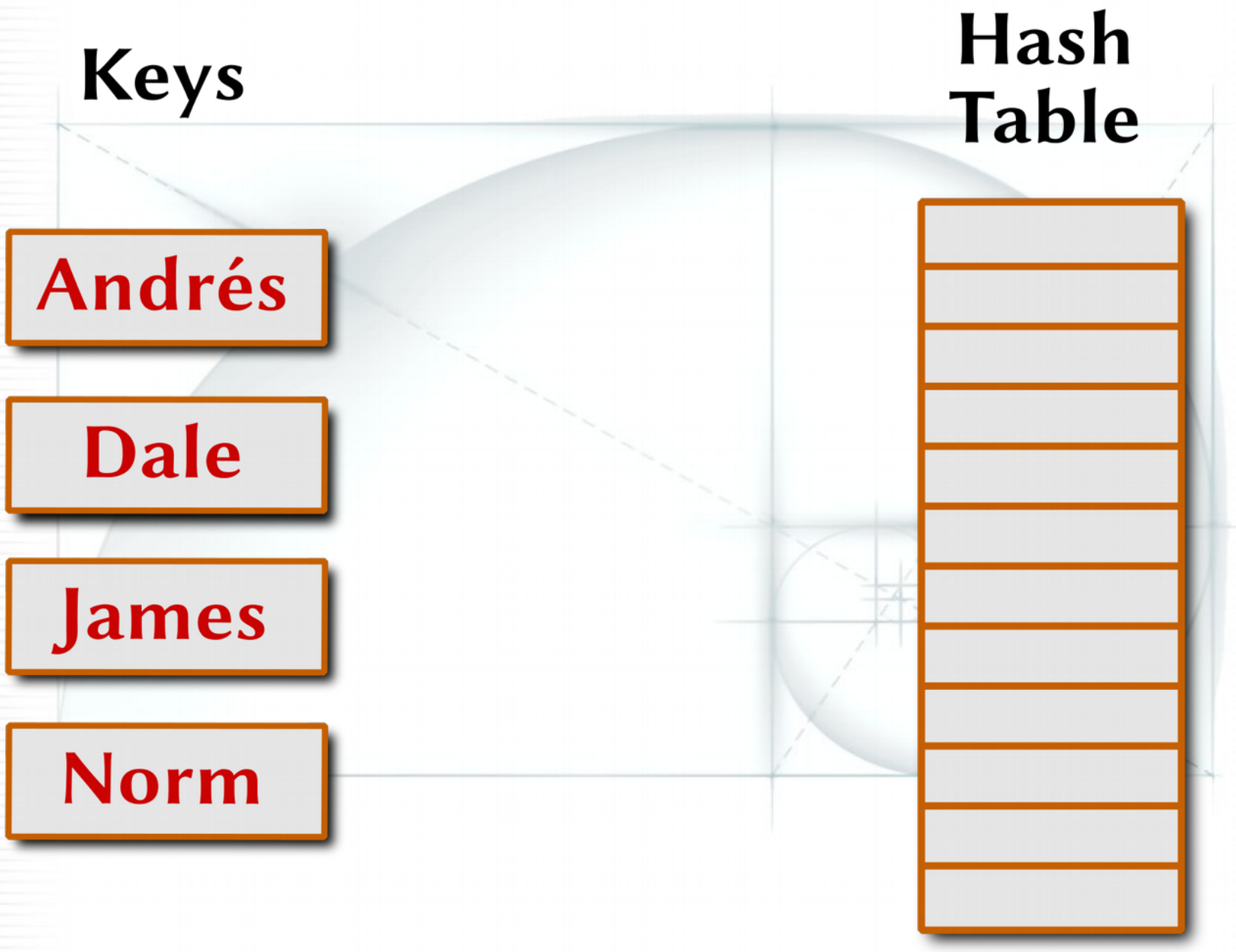
James

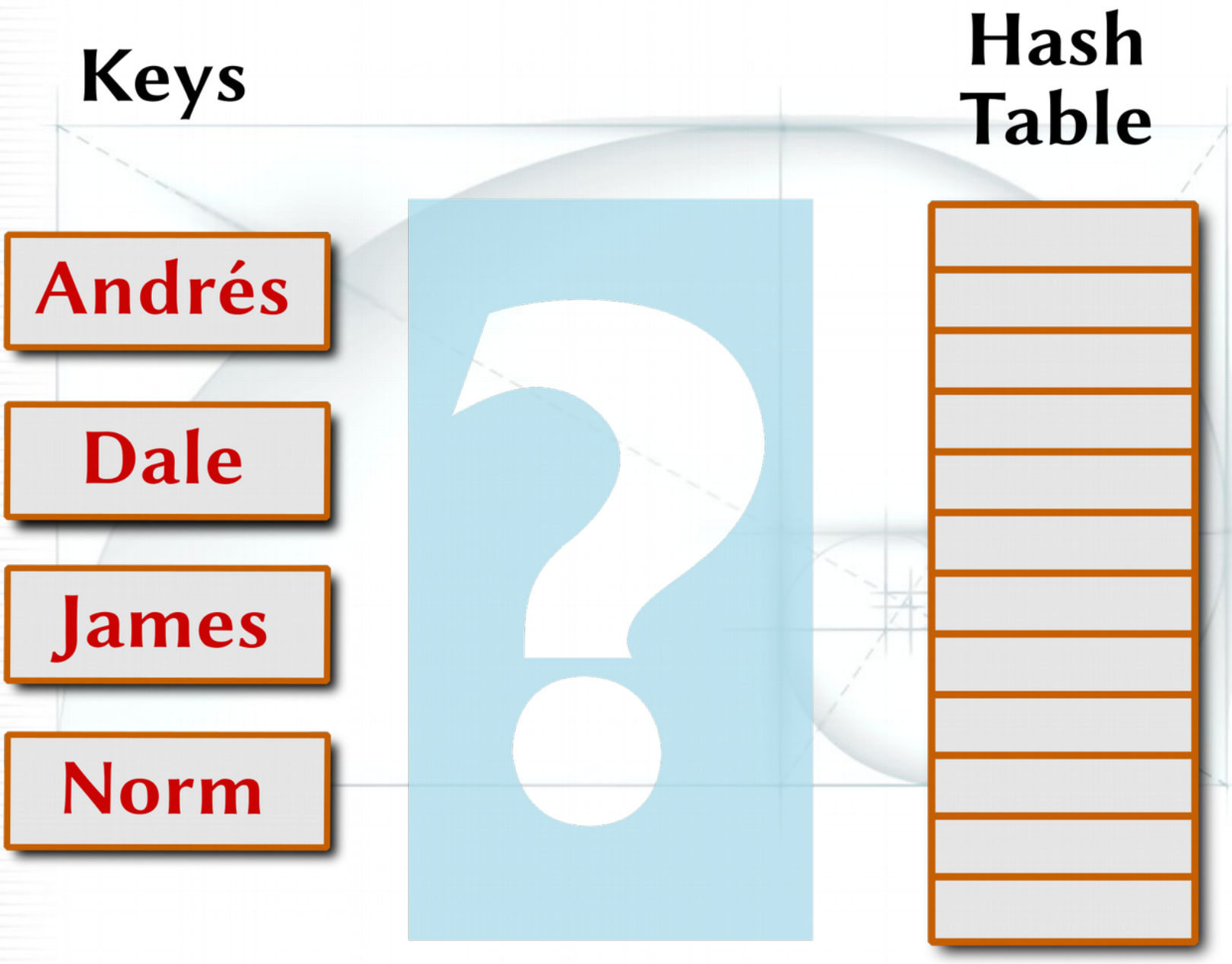
Norm

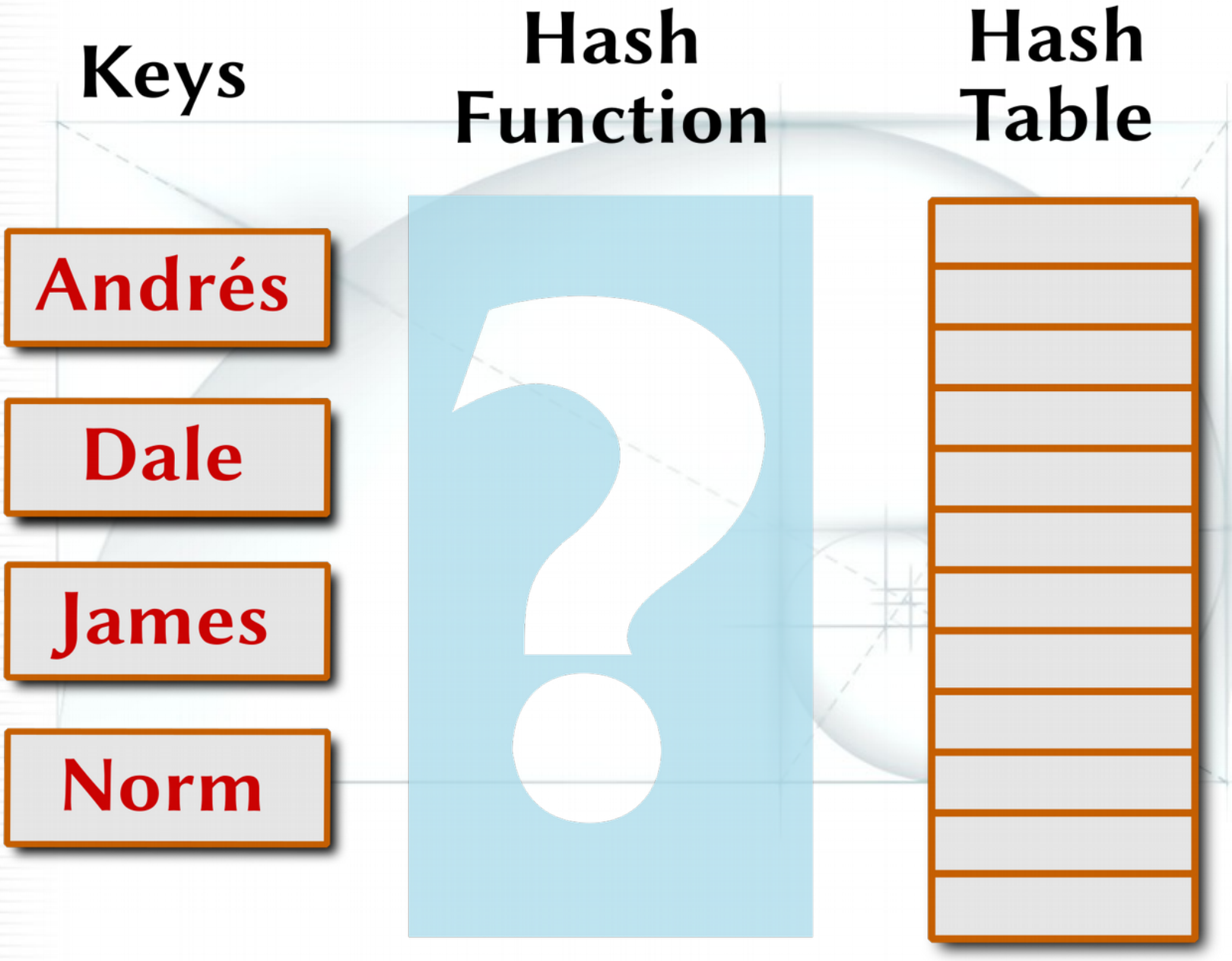


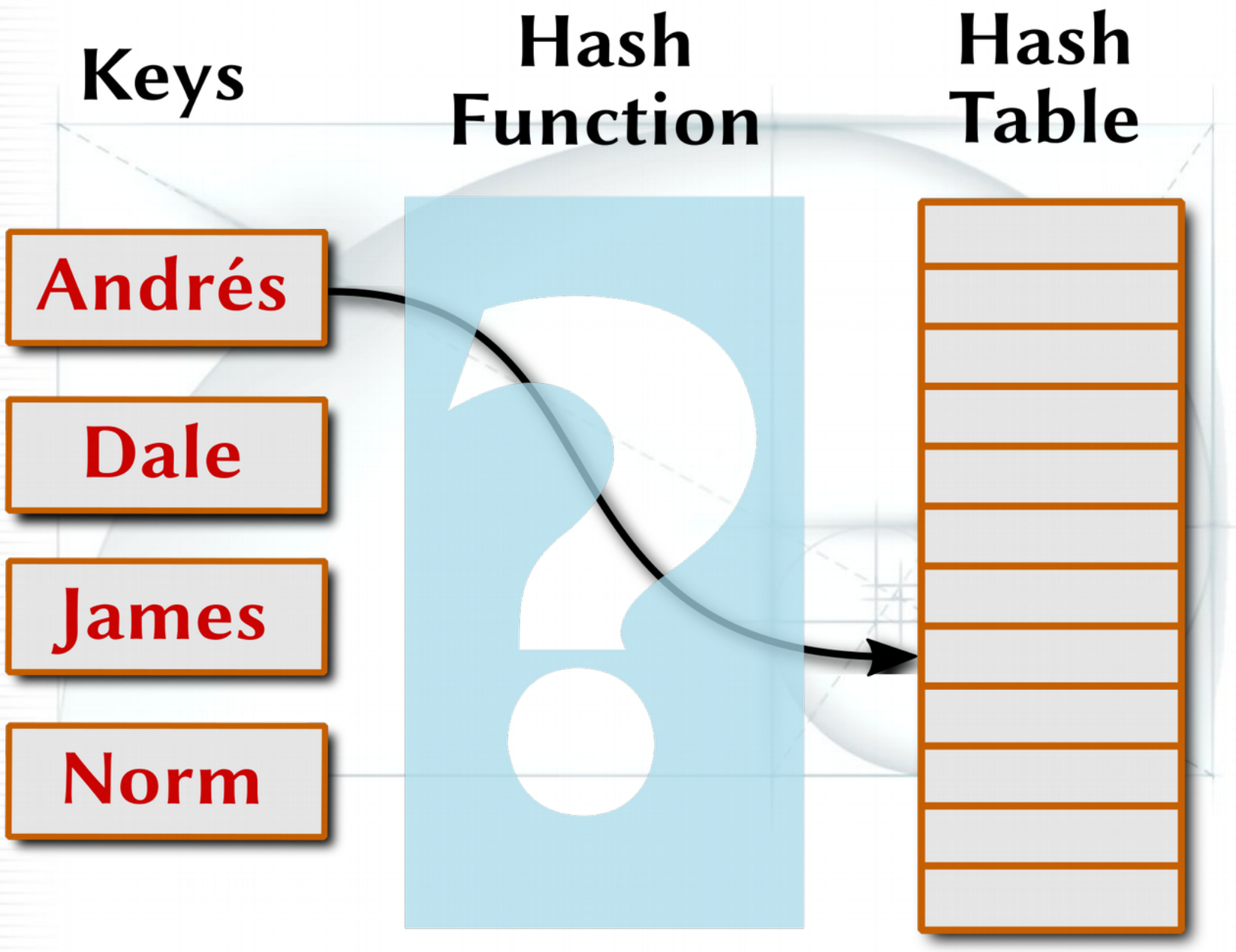
Keys

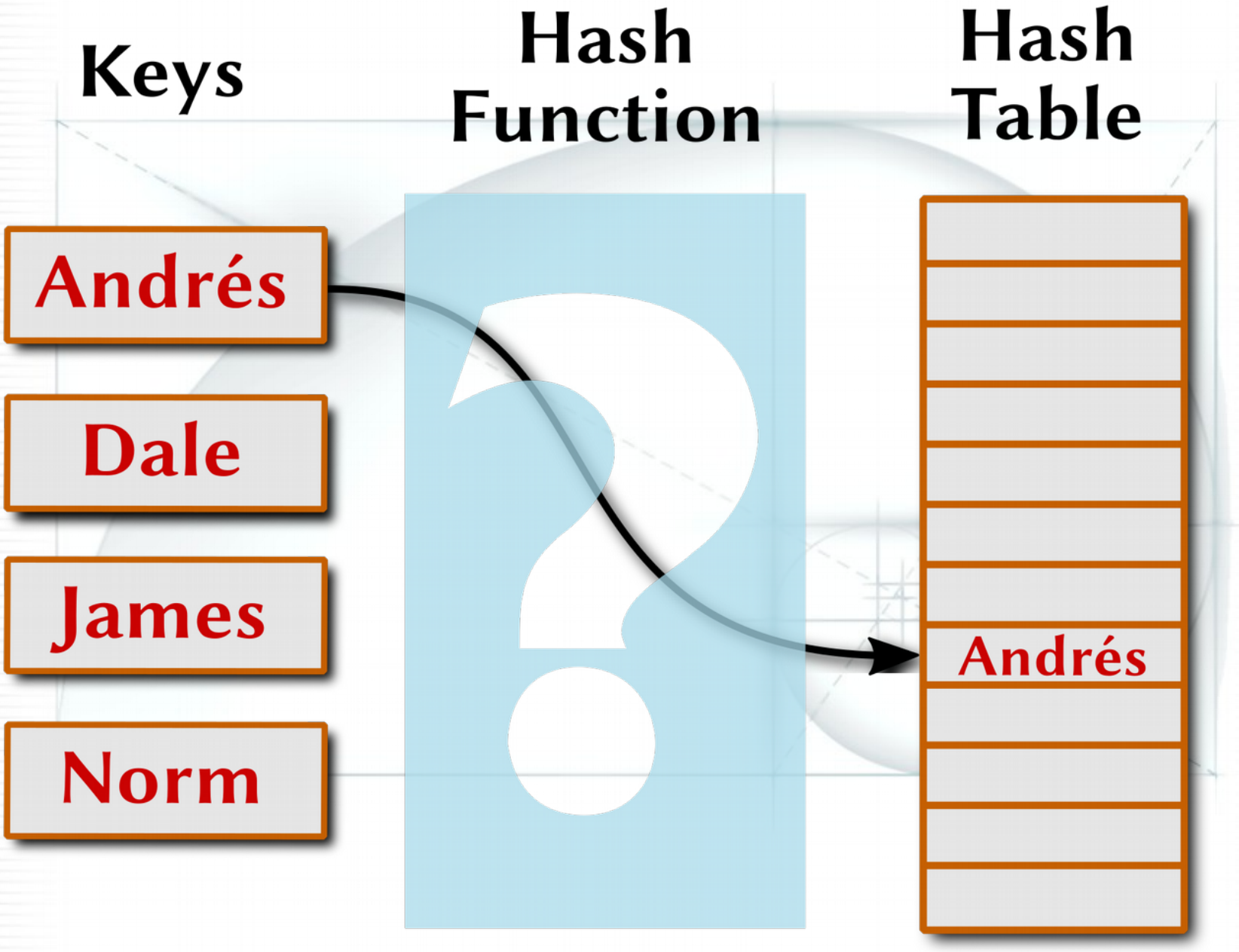


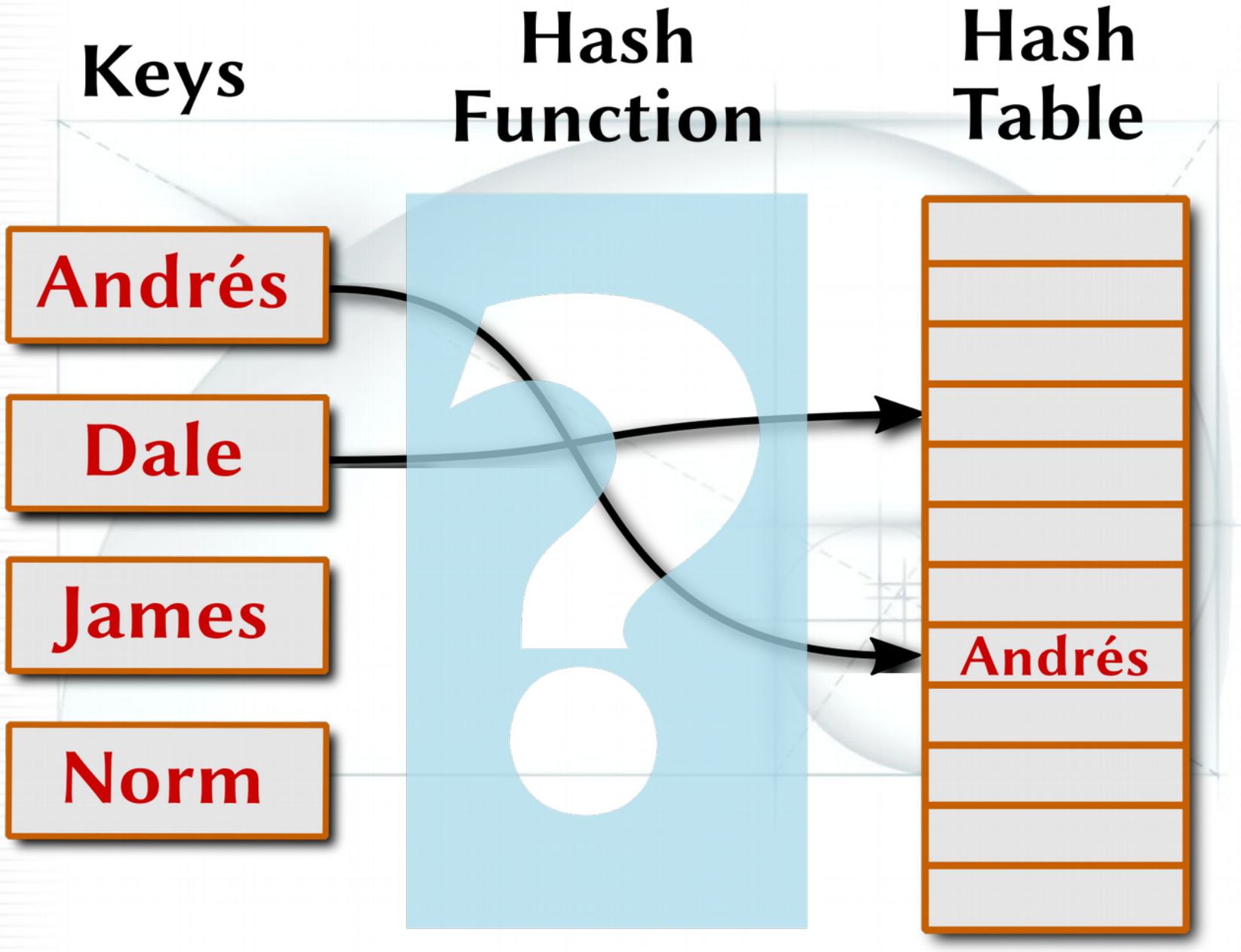

Keys
Hash Table
Andrés
Dale
James
Norm

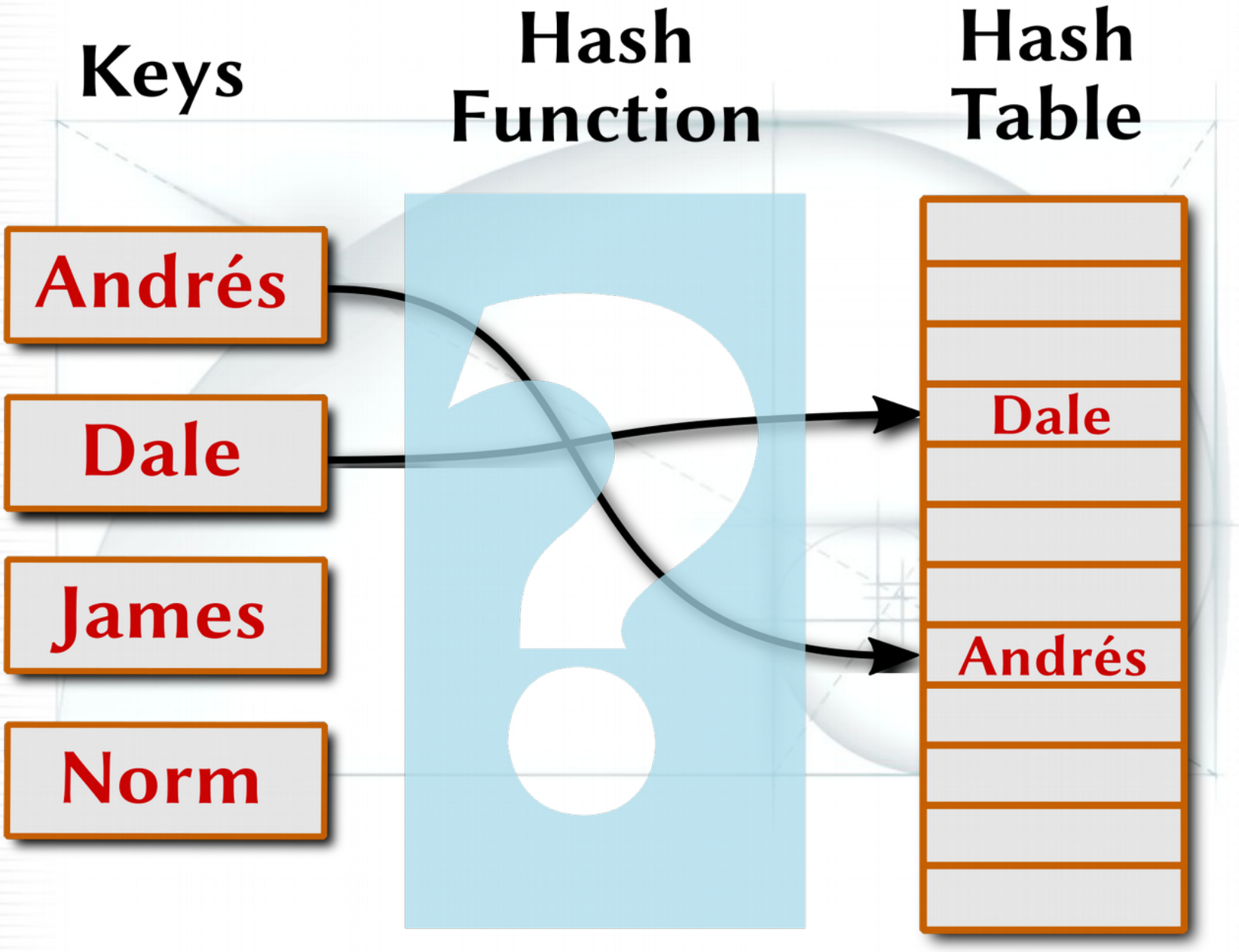

Keys
Hash Table
Andrés
Dale
James
Norm

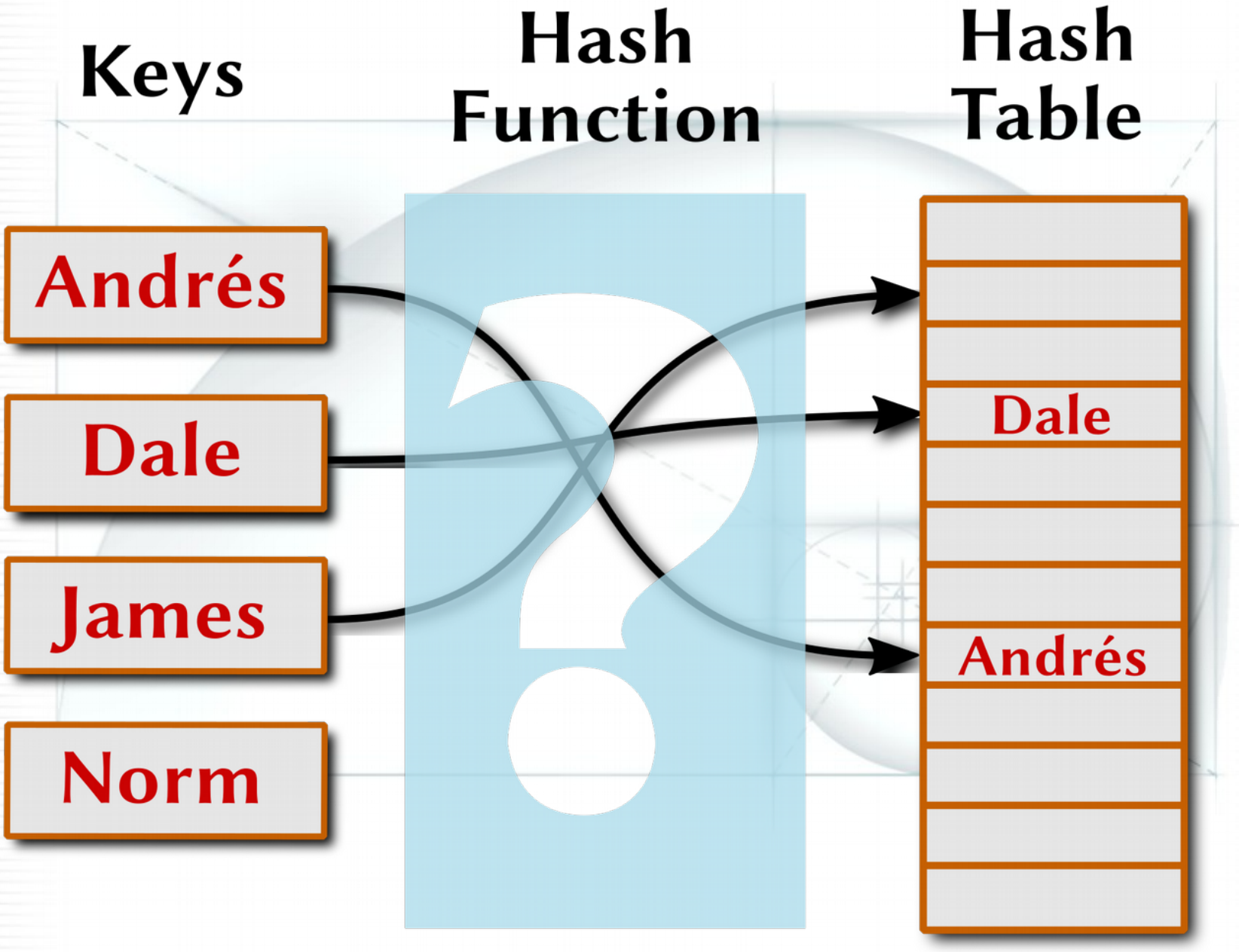

Keys
Hash Function
Hash Table
Andrés
Dale
James
Norm

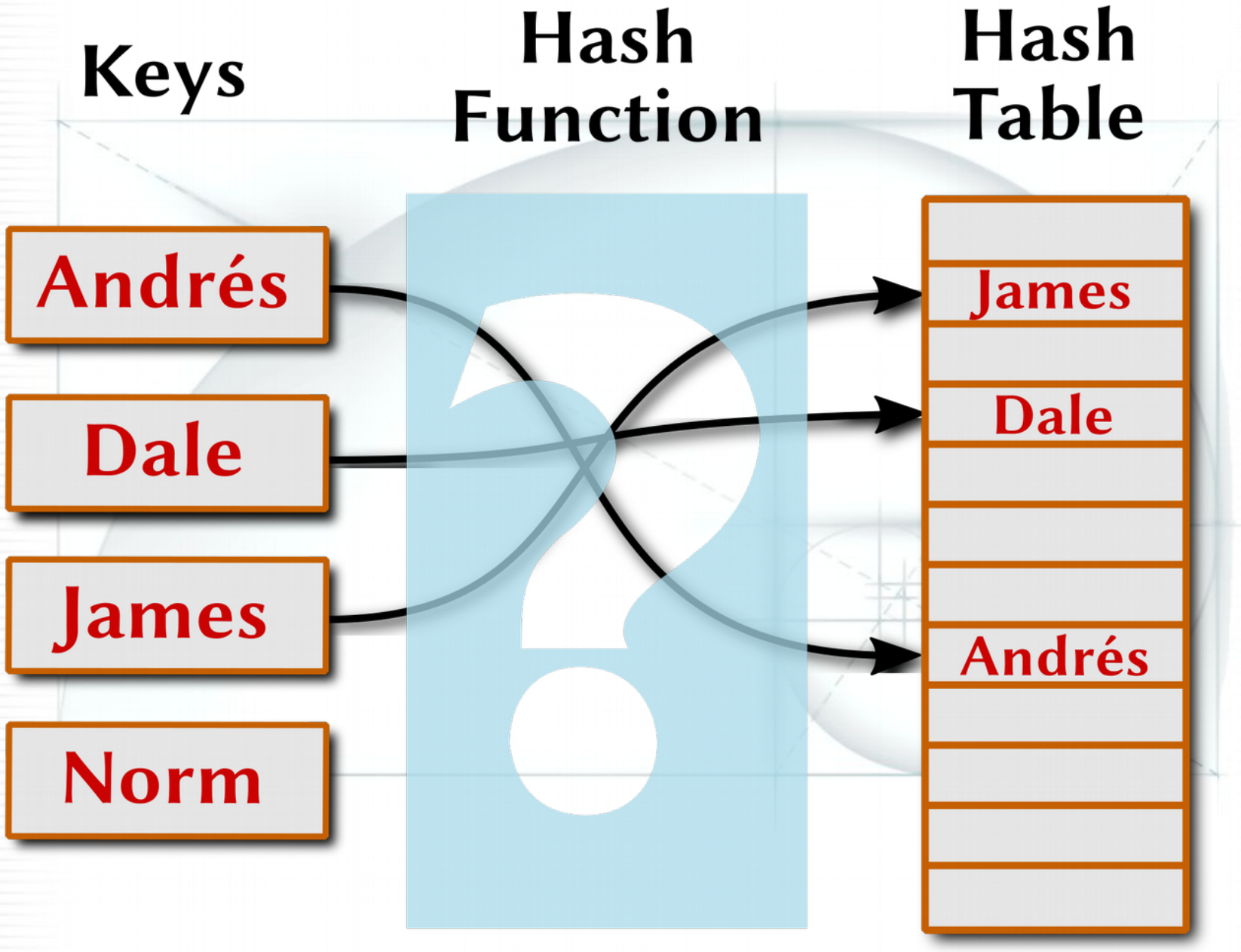

Keys
Hash Function
Hash Table
Andrés
Dale
James
Norm

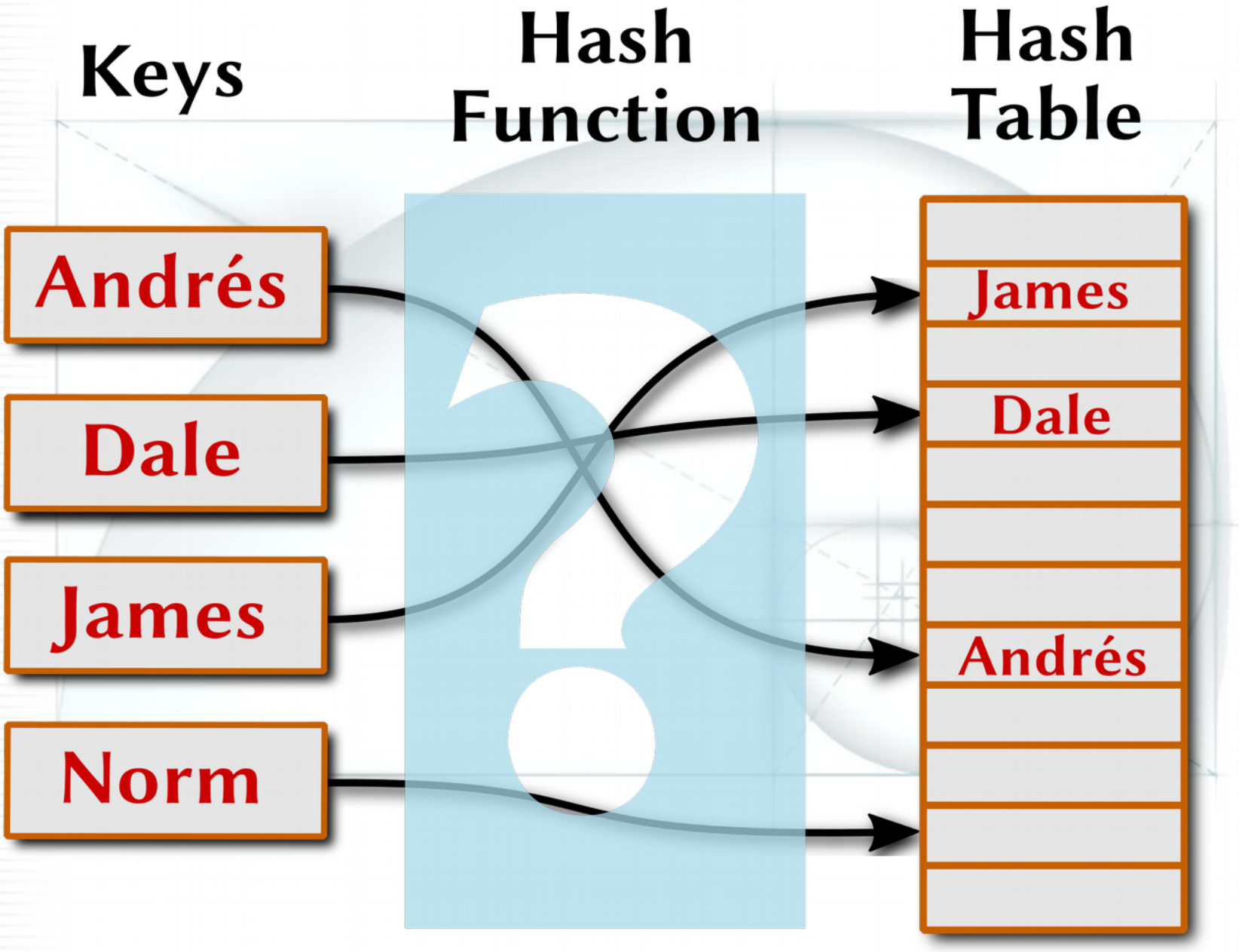

Keys
Hash Function
Hash Table
Andrés
Dale
James
Norm
Andrés

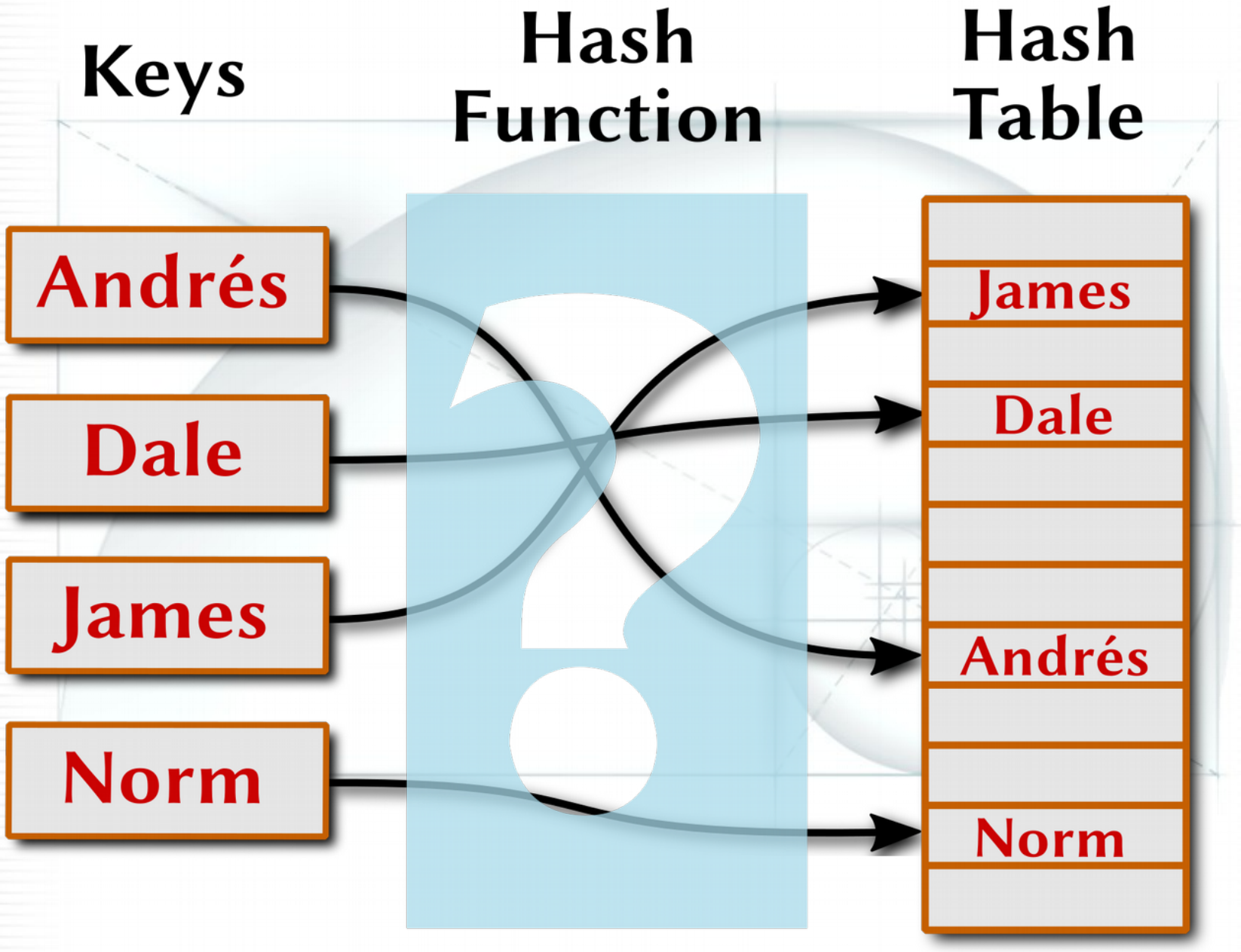

Keys
Hash Function
Hash Table
Andrés
Dale
James
Norm
Andrés

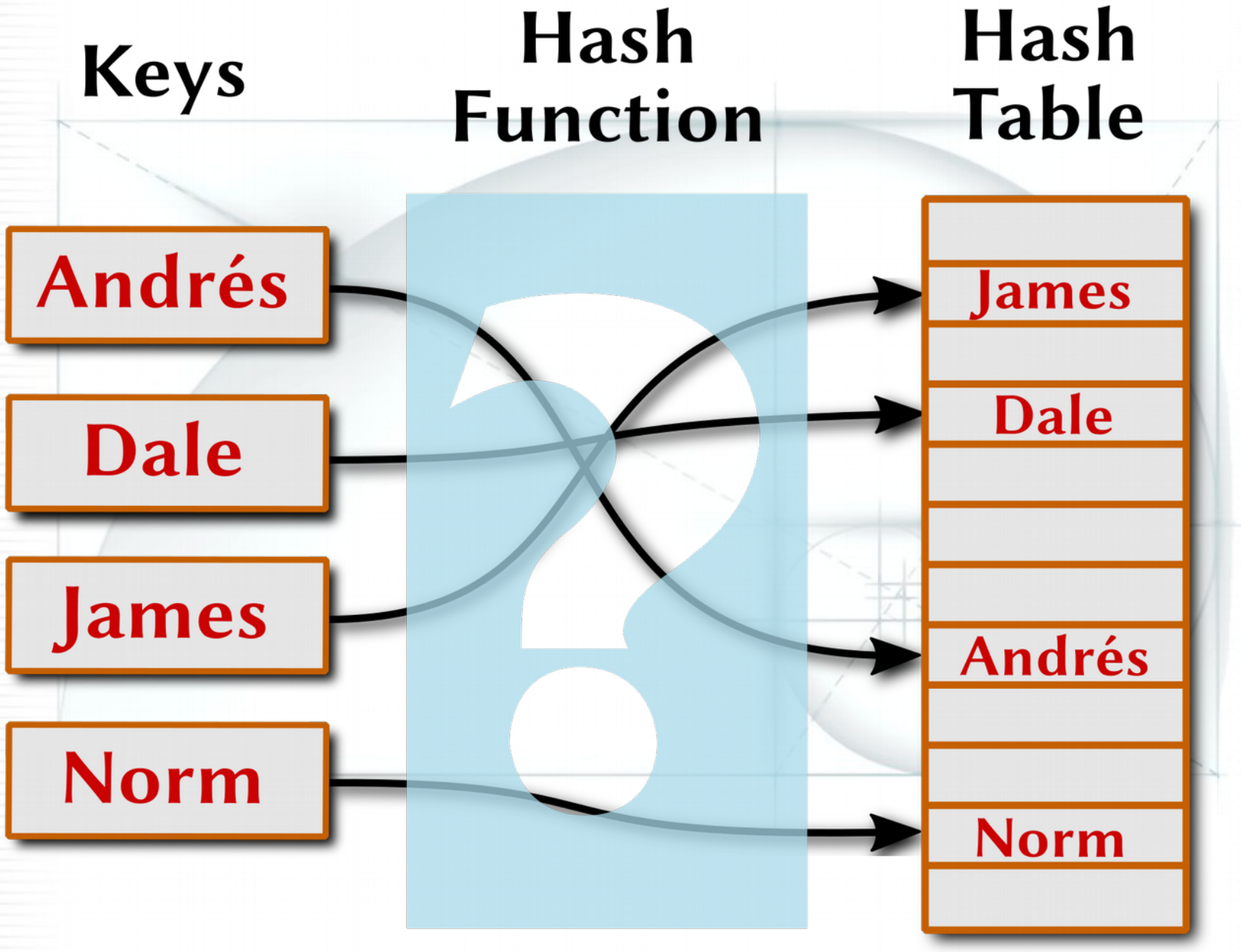

Keys
Hash Function
Hash Table
Andrés
Dale
James
Norm
Dale
Andrés


Keys
Hash Function
Hash Table
Andrés
Dale
James
Norm
Dale
Andrés

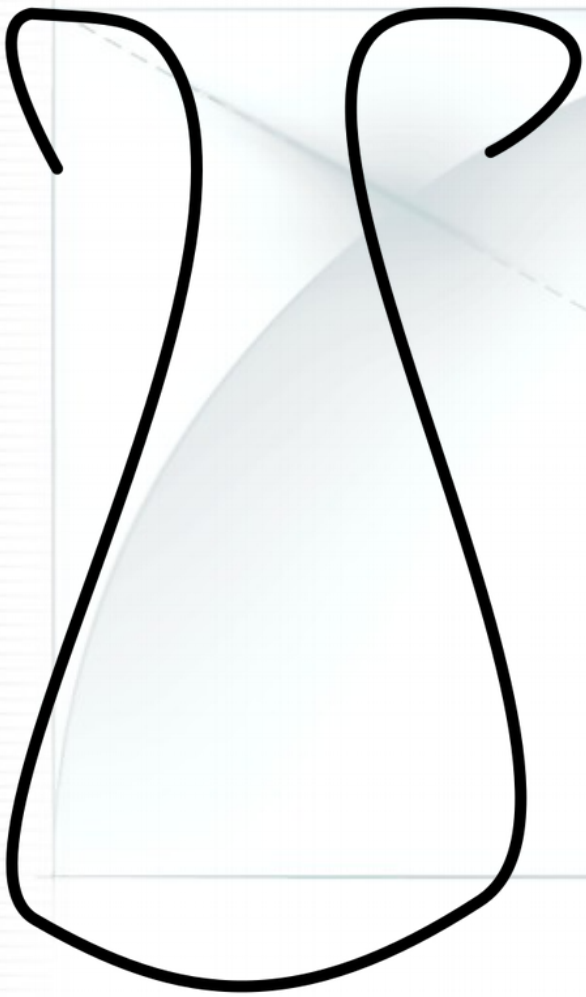

Keys
Hash Function
Hash Table
Andrés
Dale
James
Norm
James
Dale
Andrés


Keys
Hash Function
Hash Table
Andrés
Dale
James
Norm
James
Dale
Andrés


Keys
Hash Function
Hash Table
Andrés
Dale
James
Norm
James
Dale
Andrés
Norm


Keys
Hash Function
Hash Table
Andrés
Dale
James
Norm
James
Dale
Andrés
Norm

Hash Tables in Smalltalk

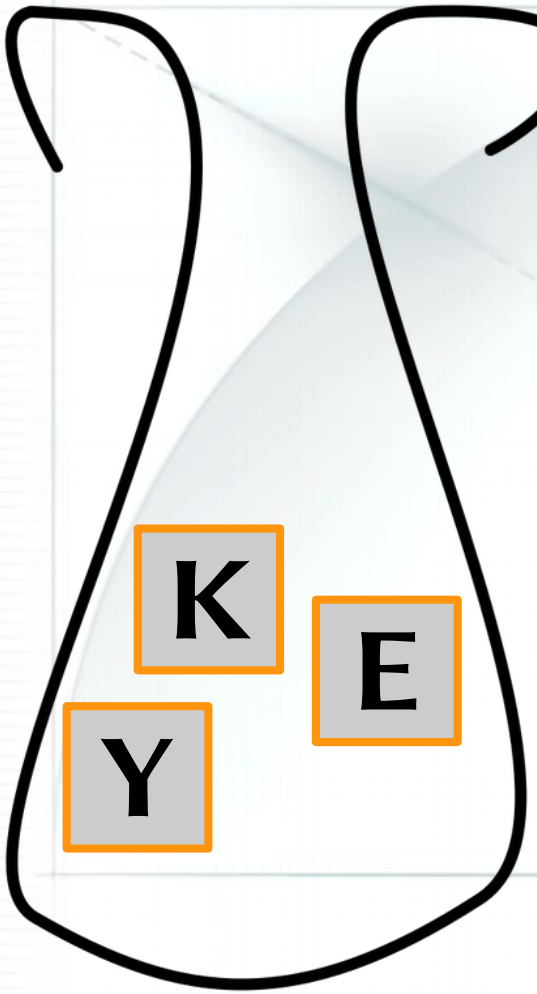


K

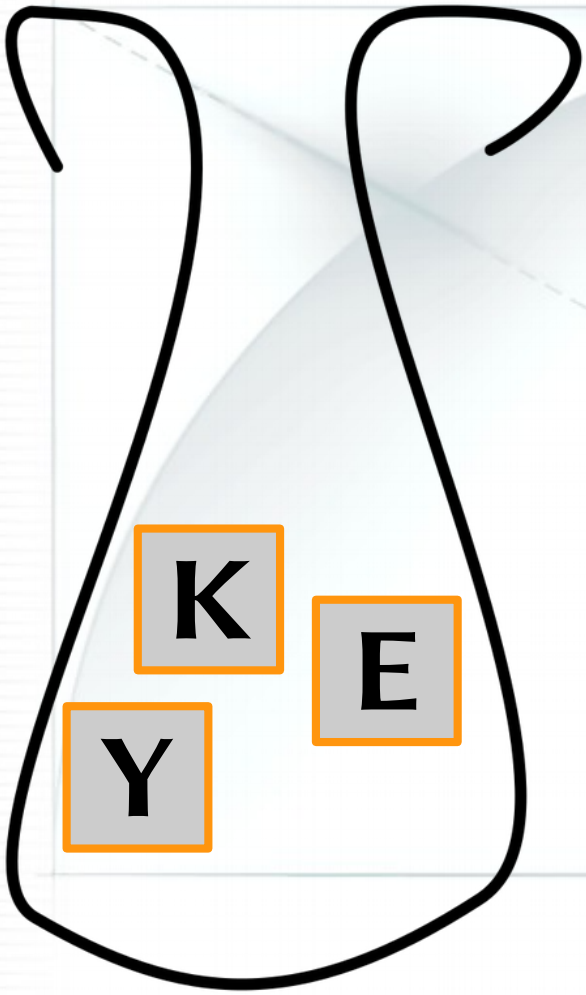
E

Y

Set



Set

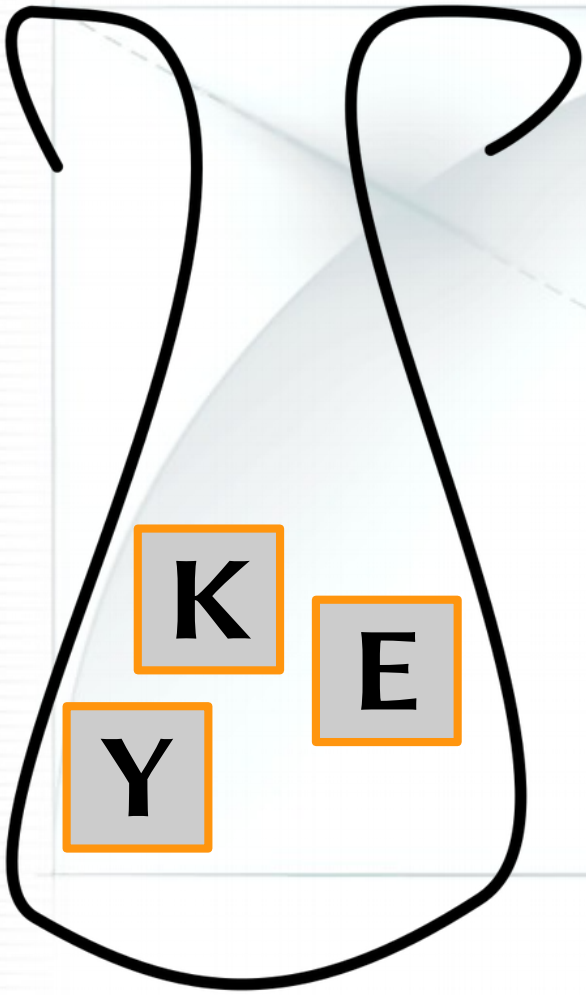


Set

#includes: behavior

K

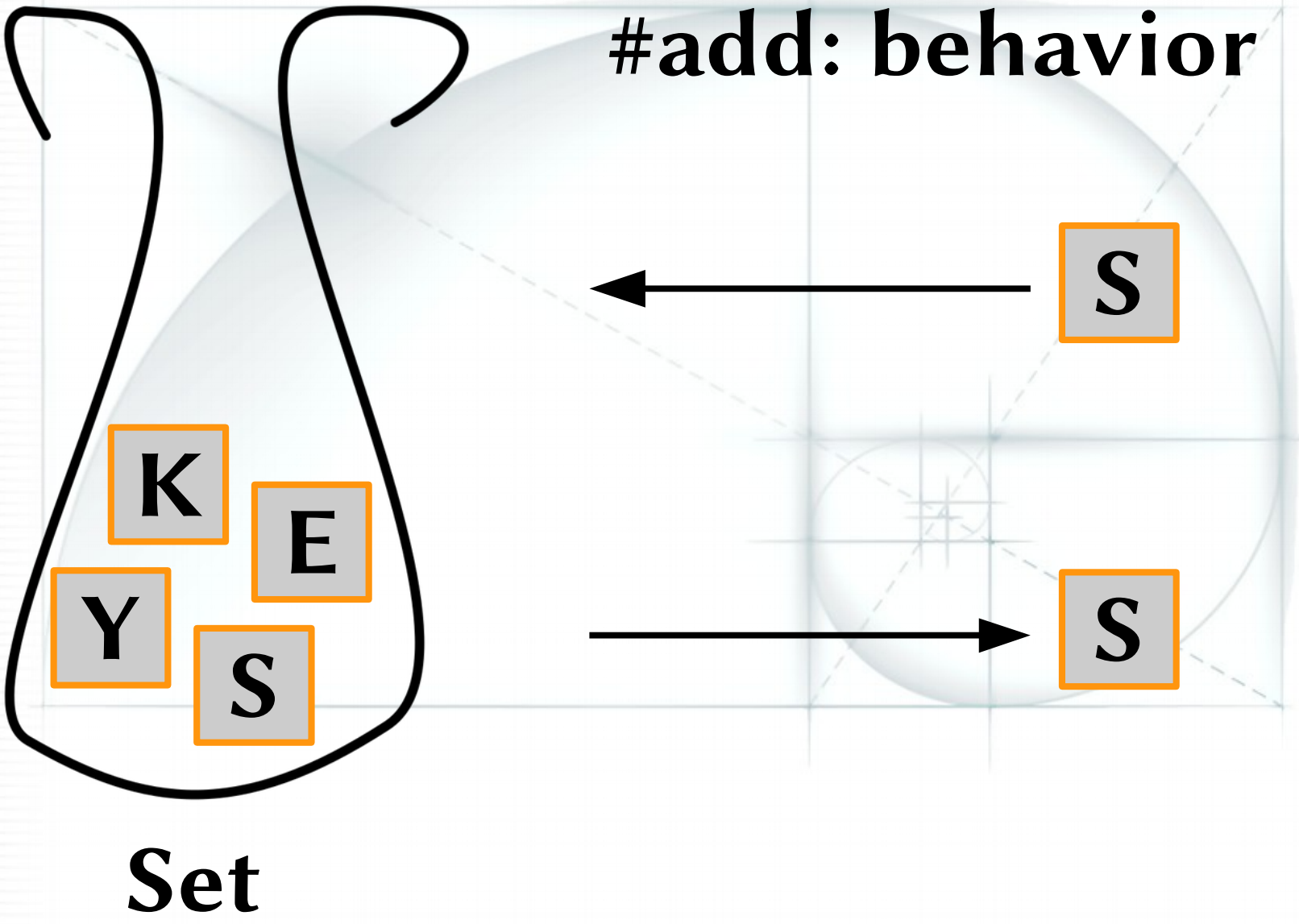
true

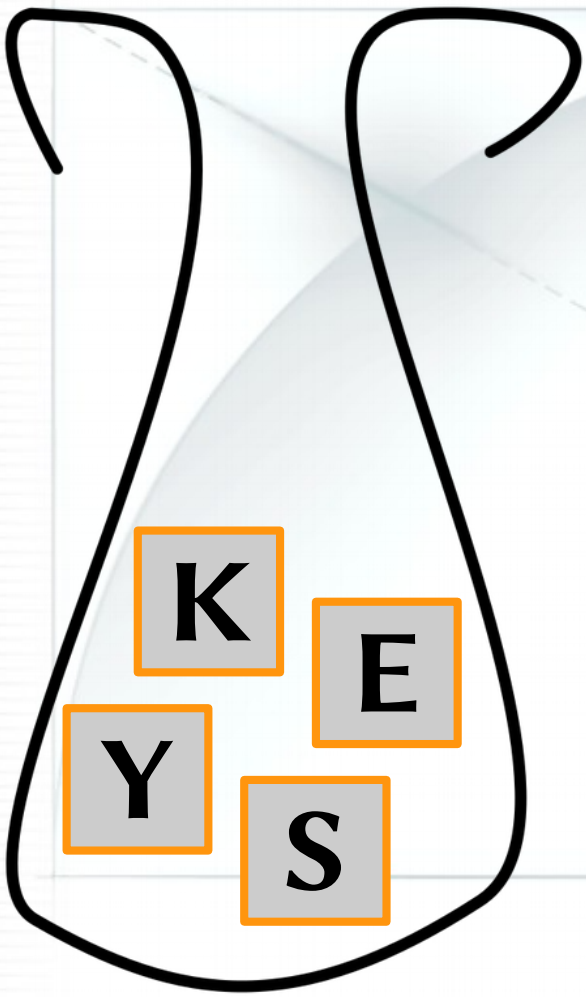


Set

#includes: behavior

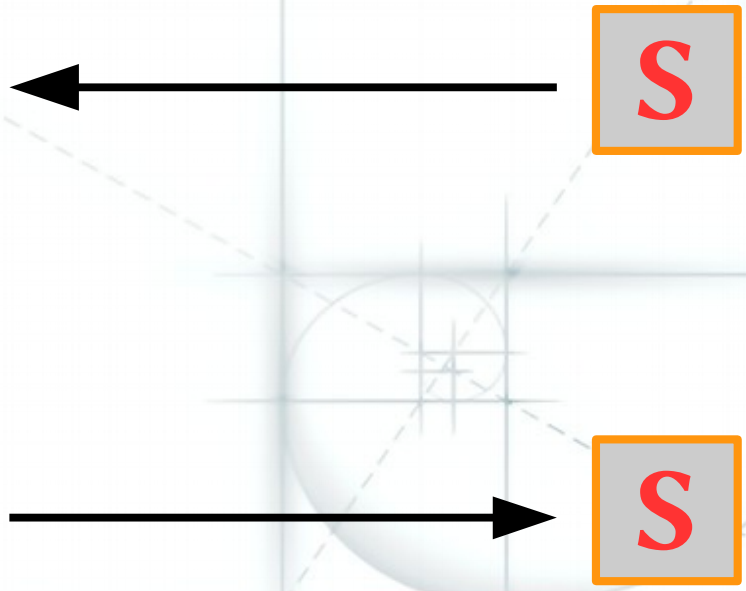
true


#add: behavior
S
S
K
E
Y
S
Set

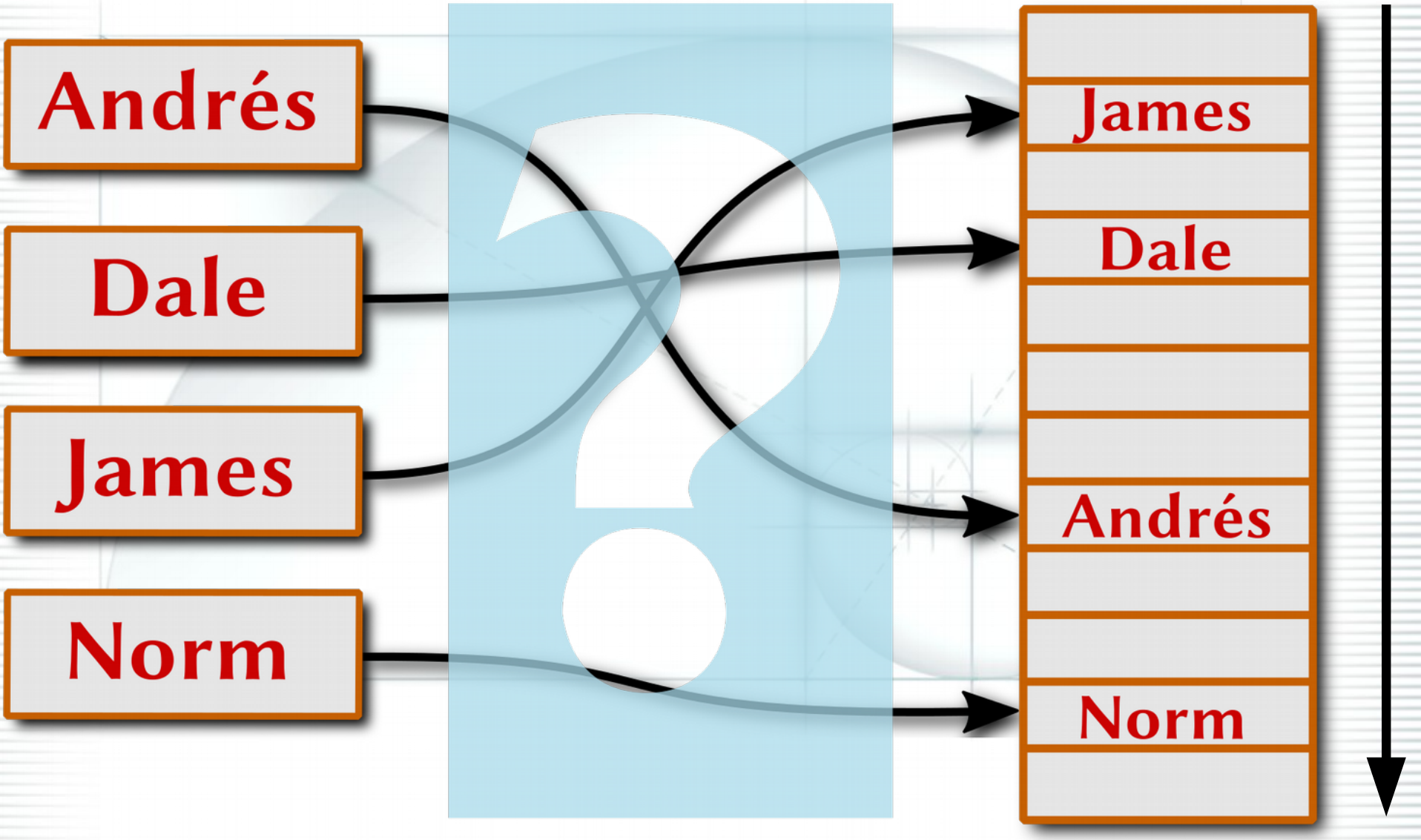


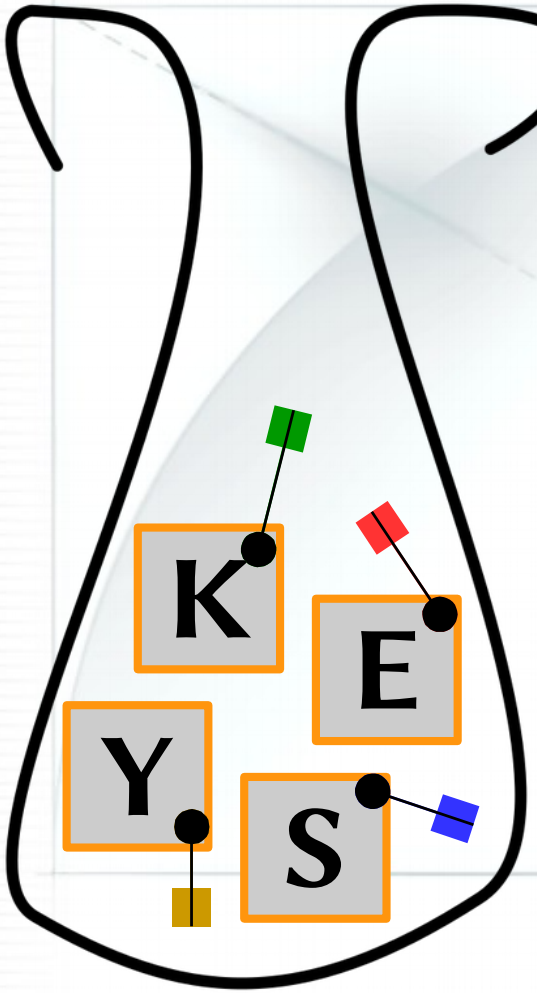
Set

#add: behavior

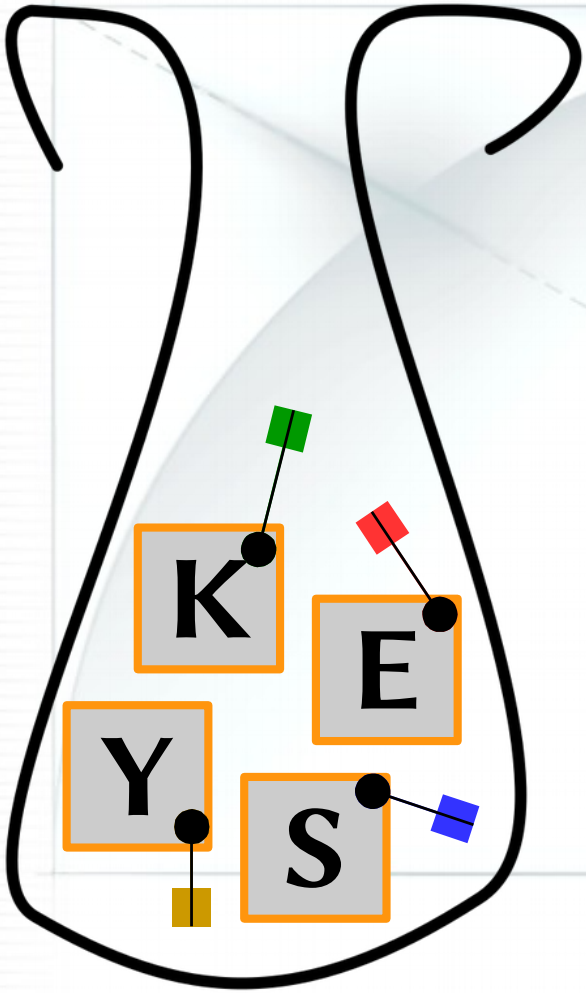


#do: aBlock

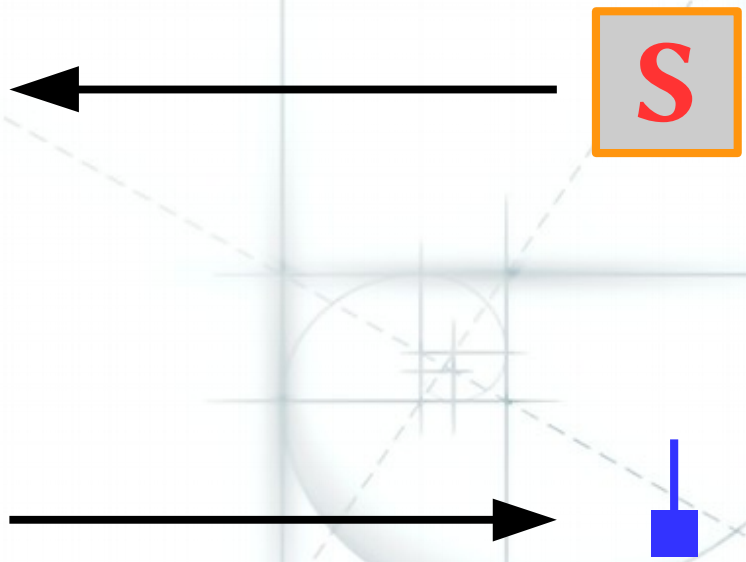




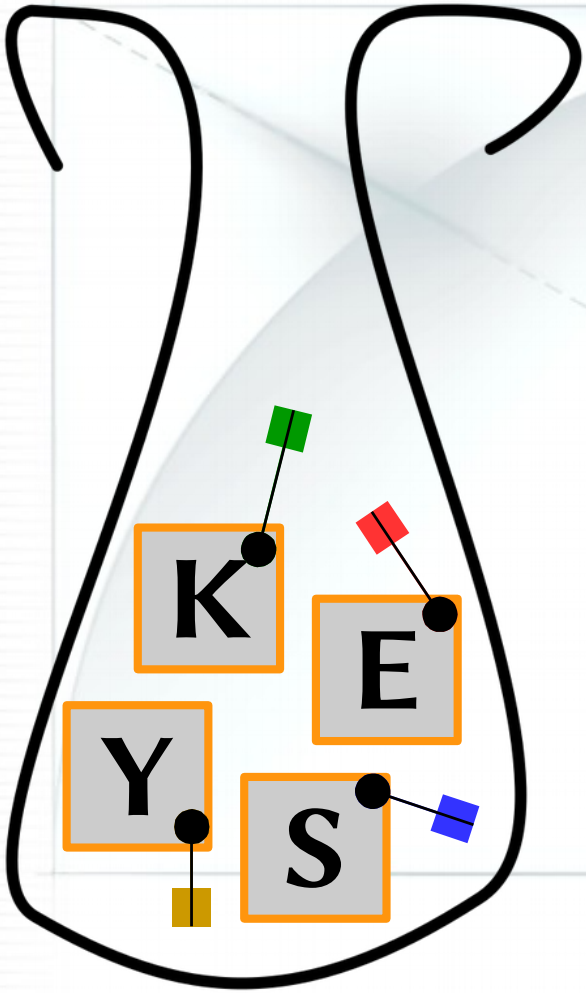
Dictionary



#at: behavior

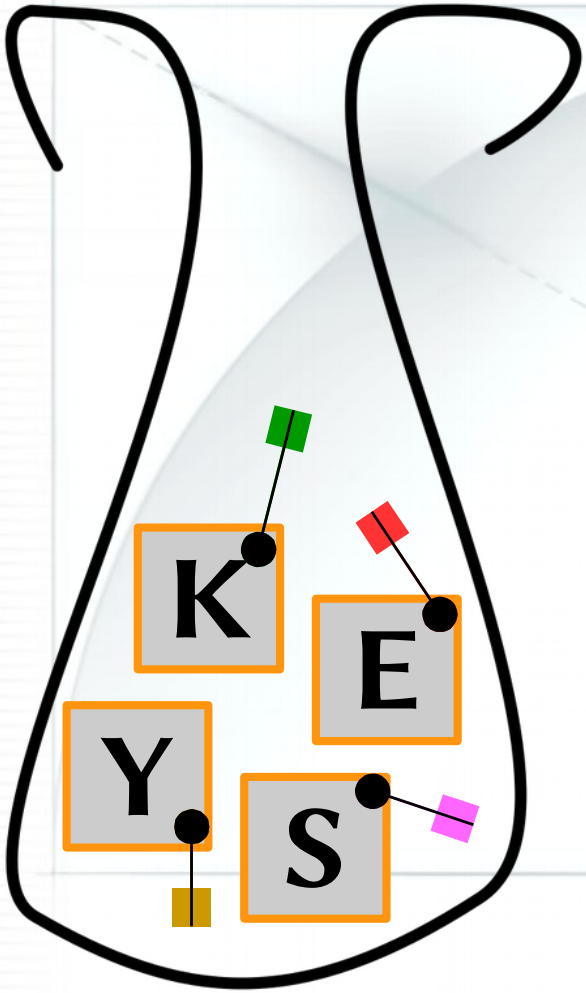


Dictionary



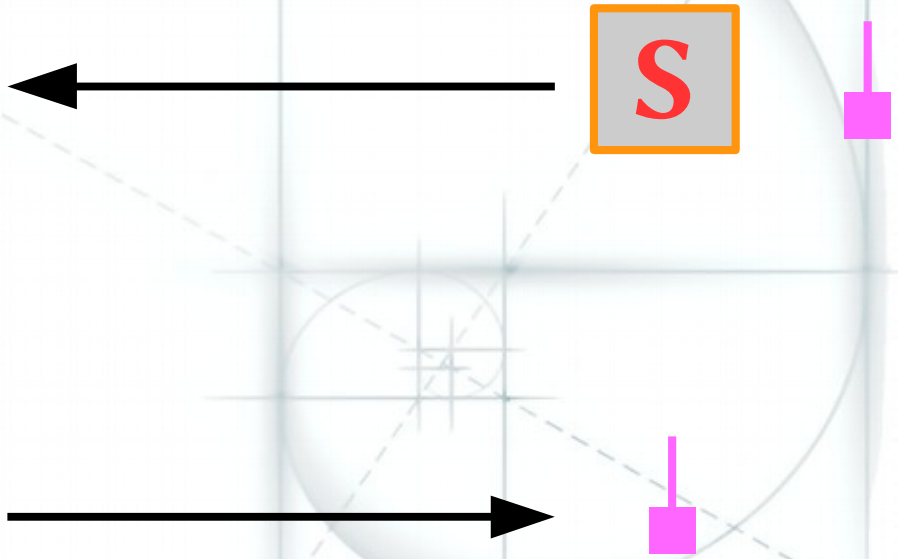
Dictionary

#at:put: behavior



Dictionary

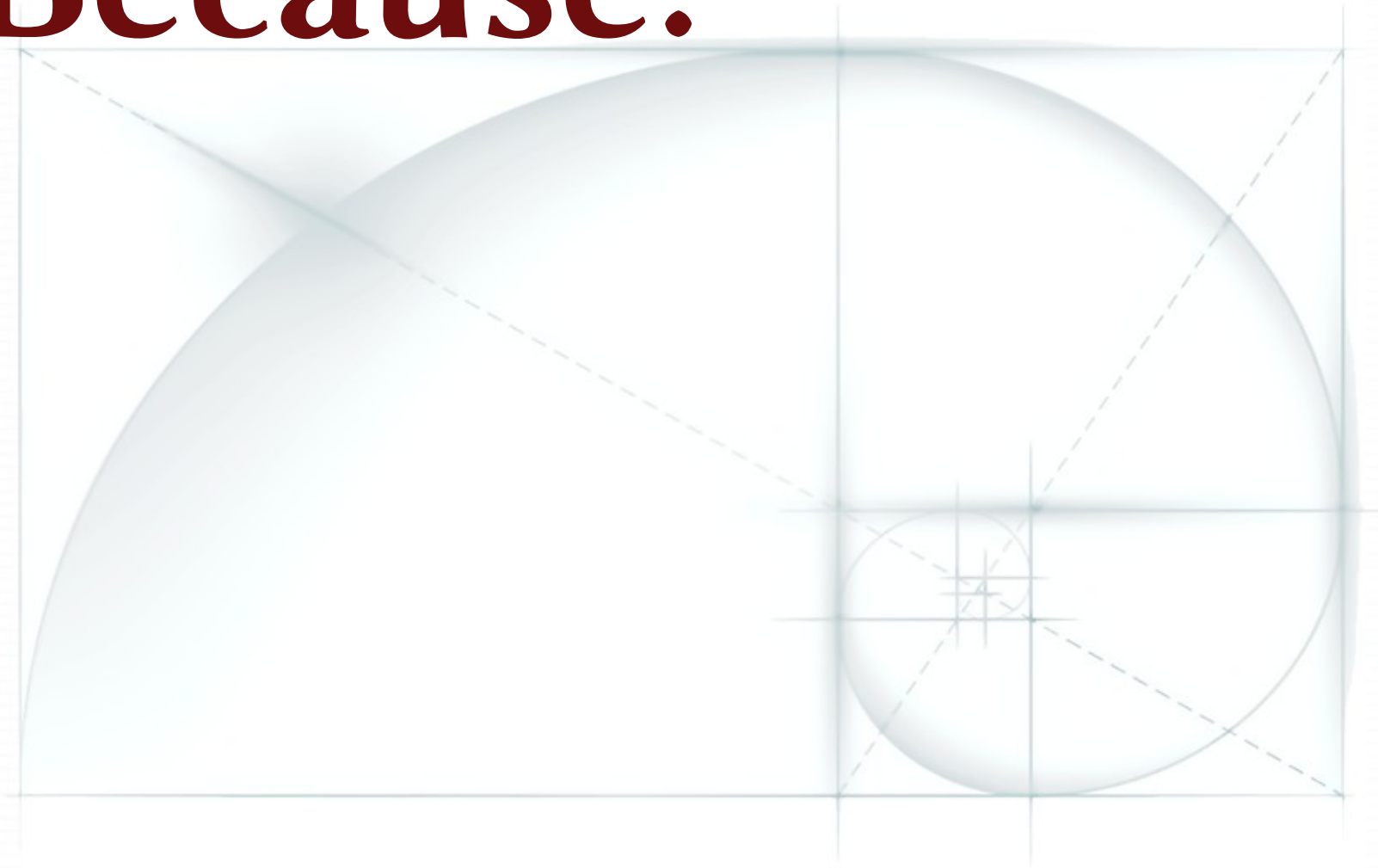
#at:put: behavior



Build Your Own Hashed Collections

Build Your Own Hashed Collections

Because:



Because:

- Special Requirements

Because:

- Special Requirements
- Security

Because:

- Special Requirements
- Security
- Performance

Because:

- Special Requirements
- Security
- Performance
  - Space

Because:

- Special Requirements
- Security
- Performance
  - Space
  - Speed

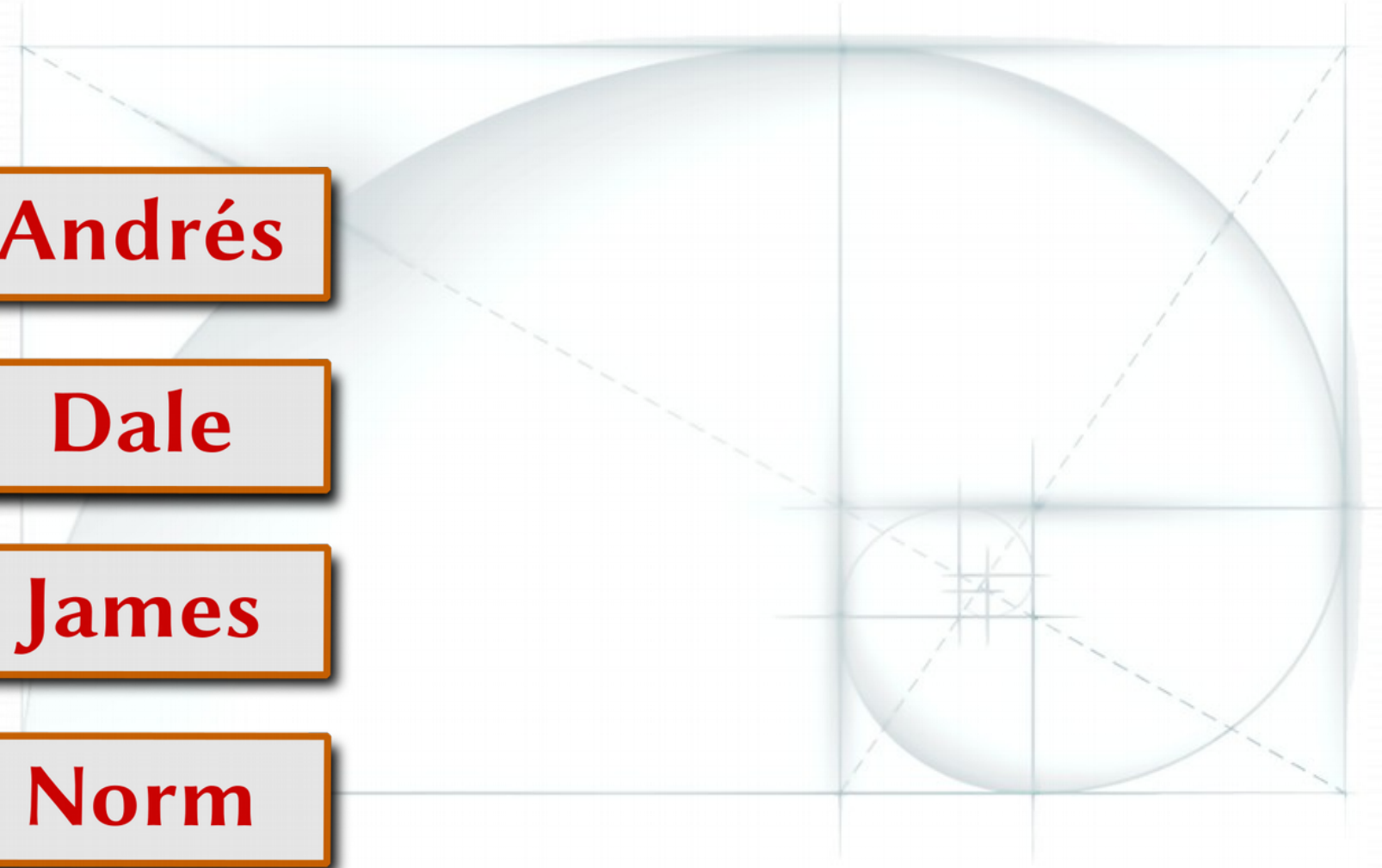
Hash Functions and Domains

Andrés

Dale

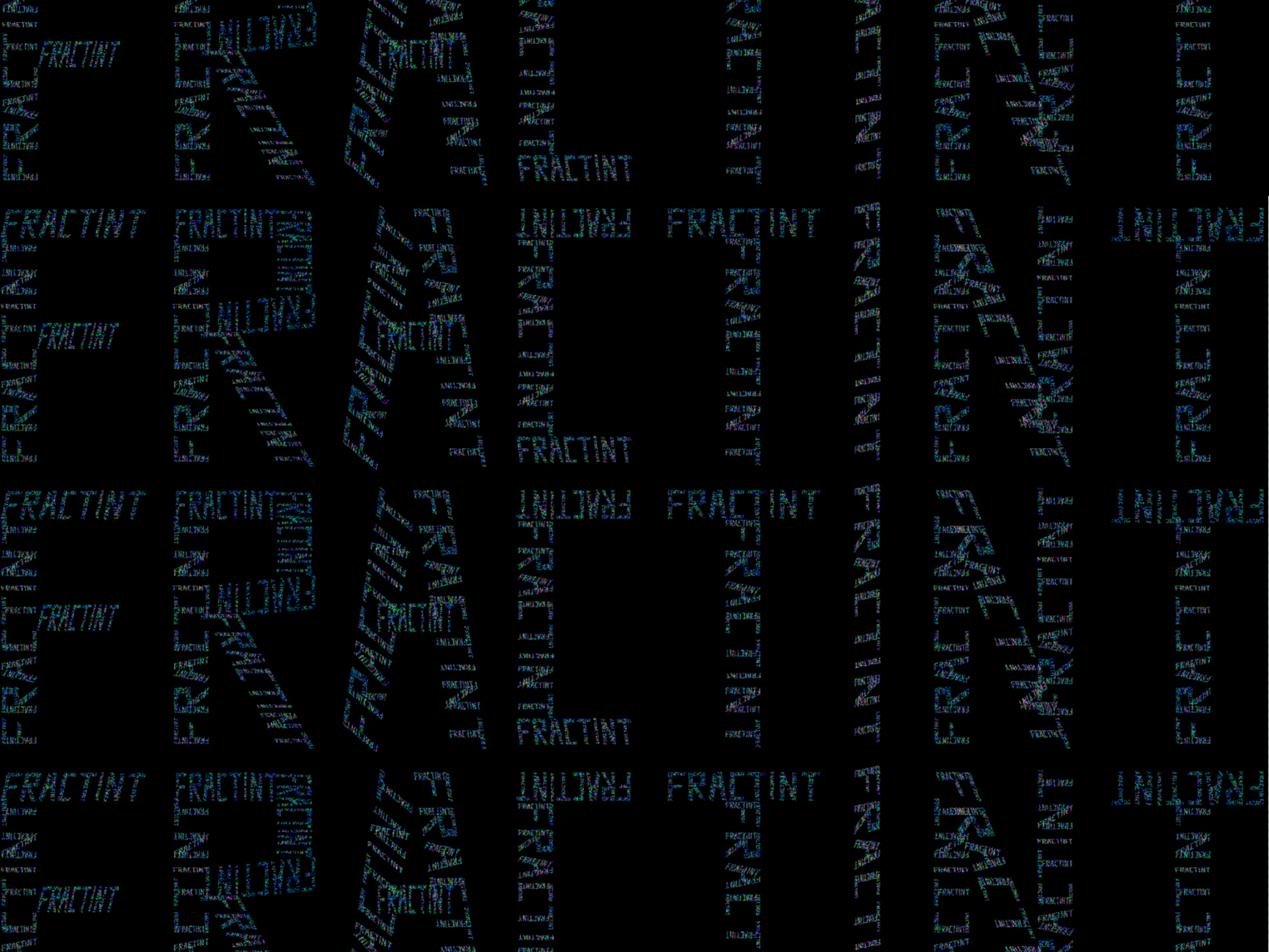
James

Norm



Perfect hash function

Andrés	f(Andrés) = 10
Dale	f(Dale) = 4
James	f(James) = 7
Norm	f(Norm) = 1


FRACTINT

Pragmatic hash functions

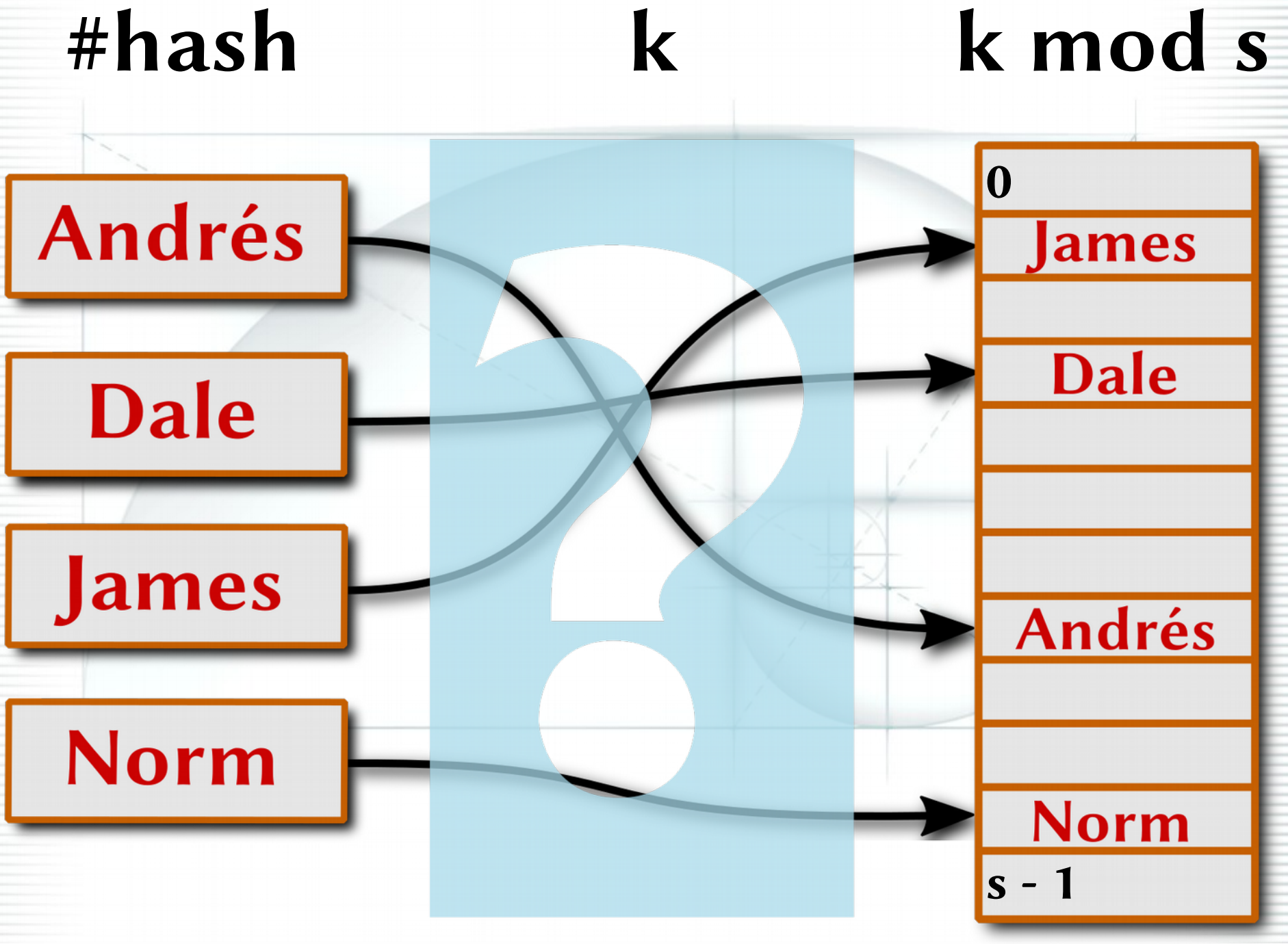
Andrés

Dale

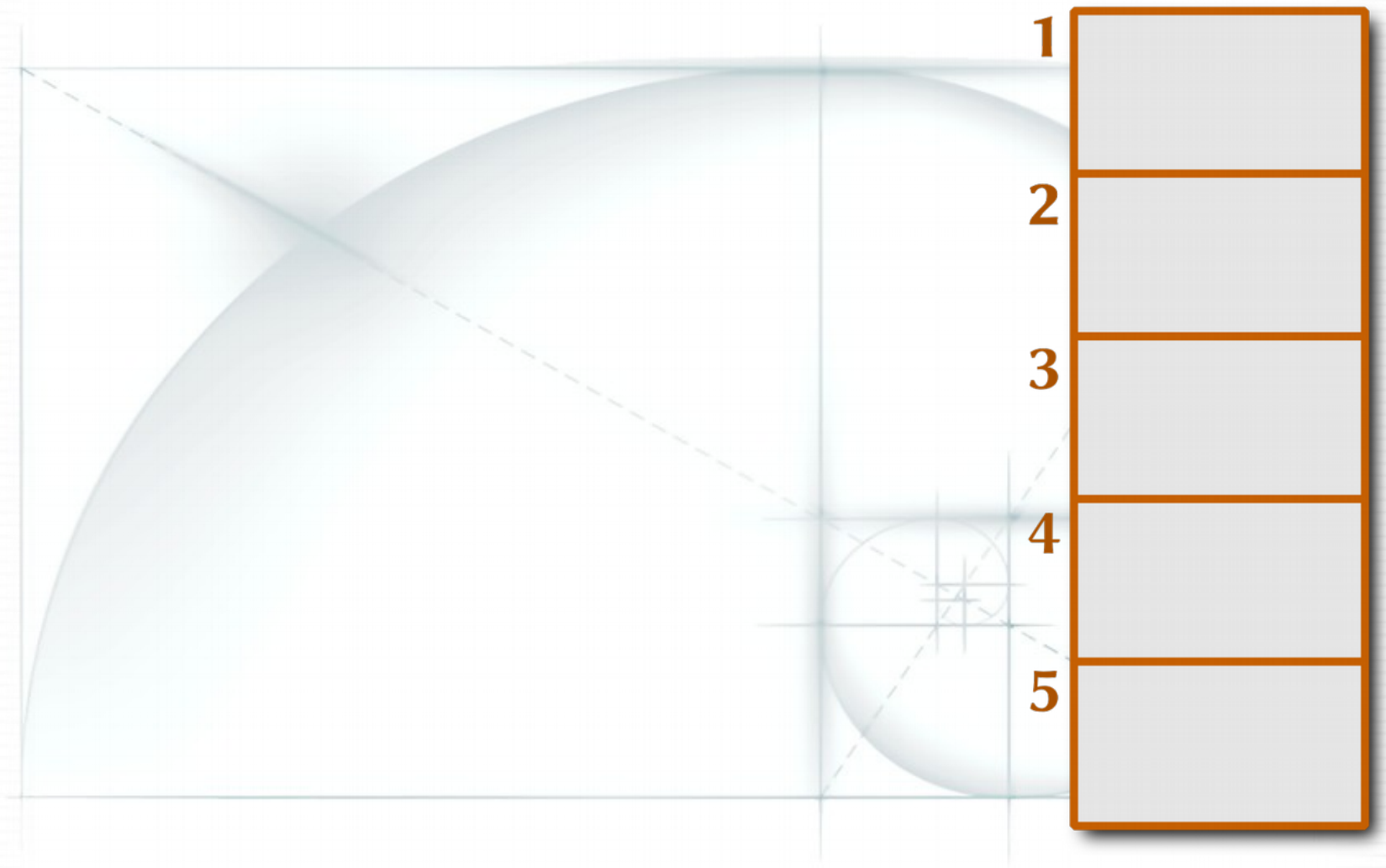
James

Norm

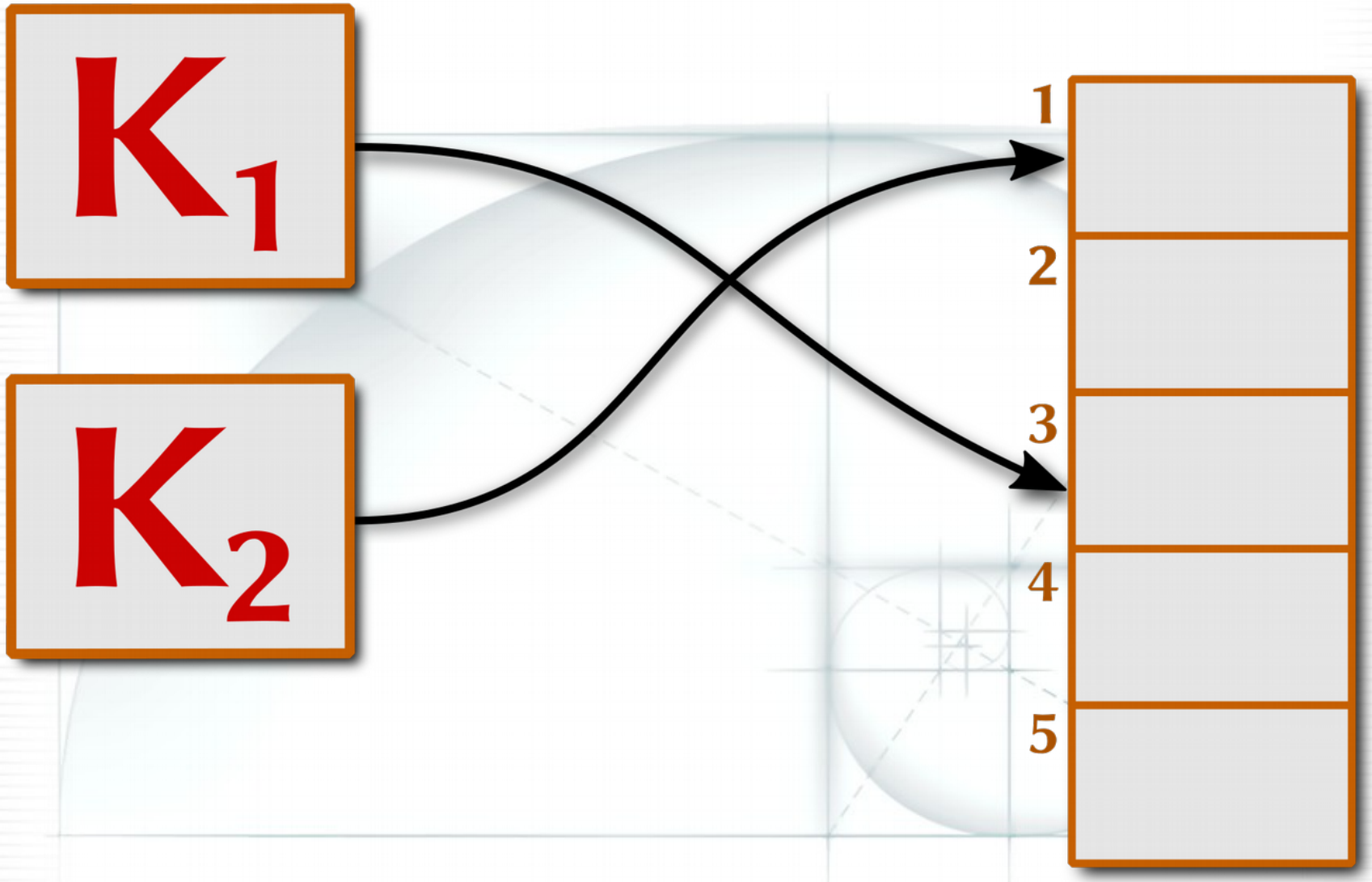
$$f(\text{string}) = k \bmod s$$

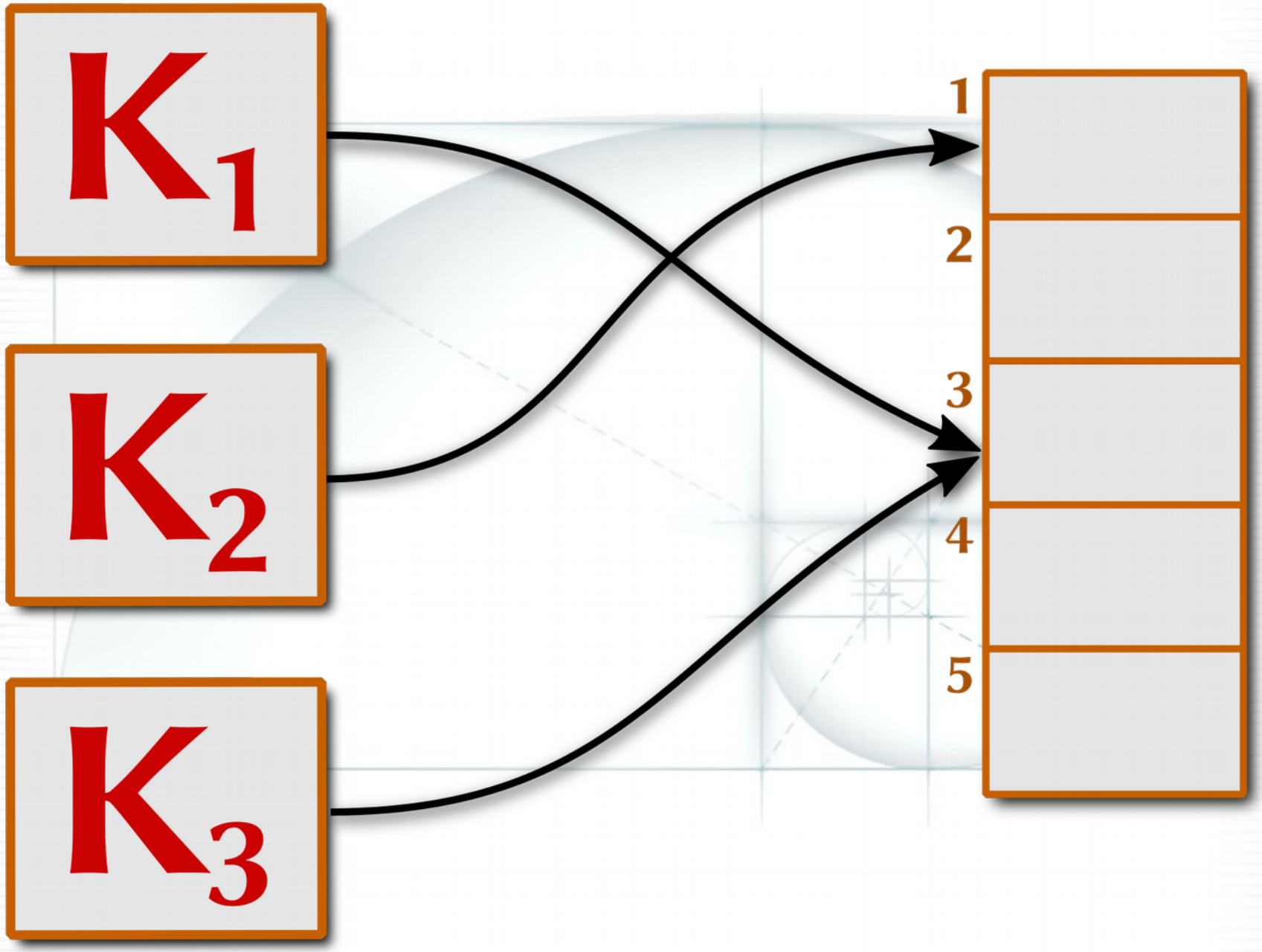

#hash
k
k mod s
Andrés
Dale
James
Norm
0
James
Dale
Andrés
Norm
s - 1

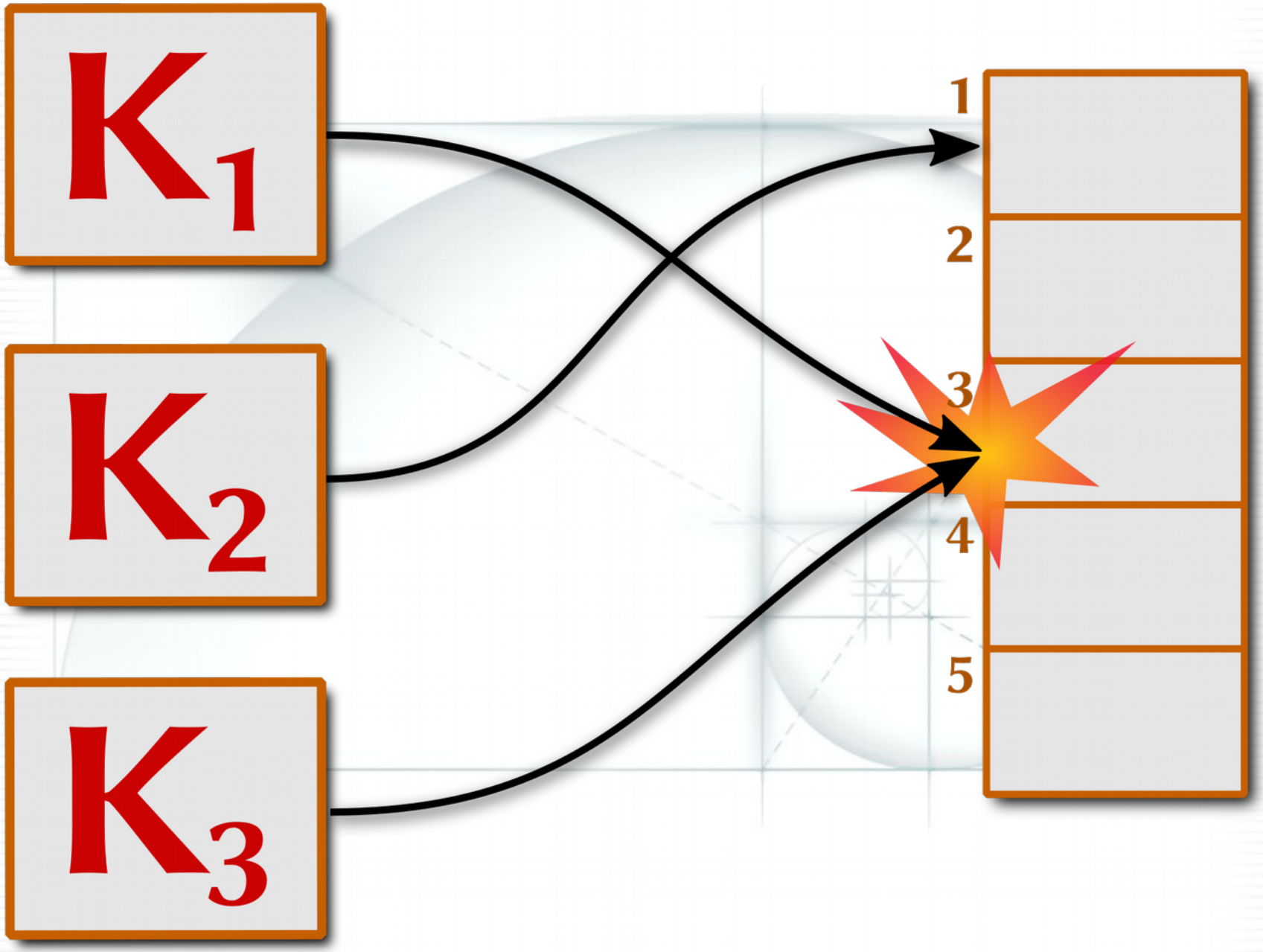
Collisions

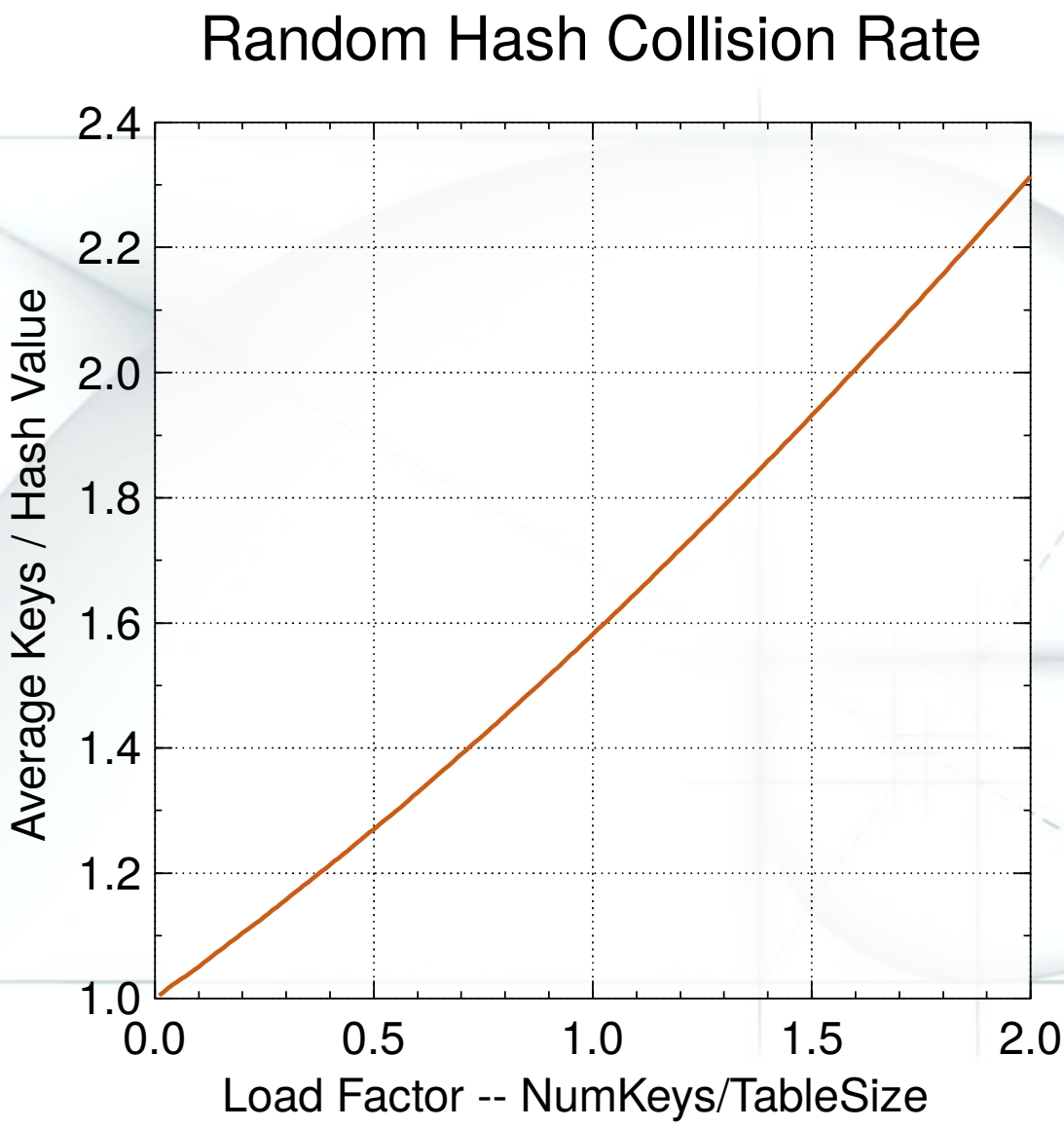

1
2
3
4
5

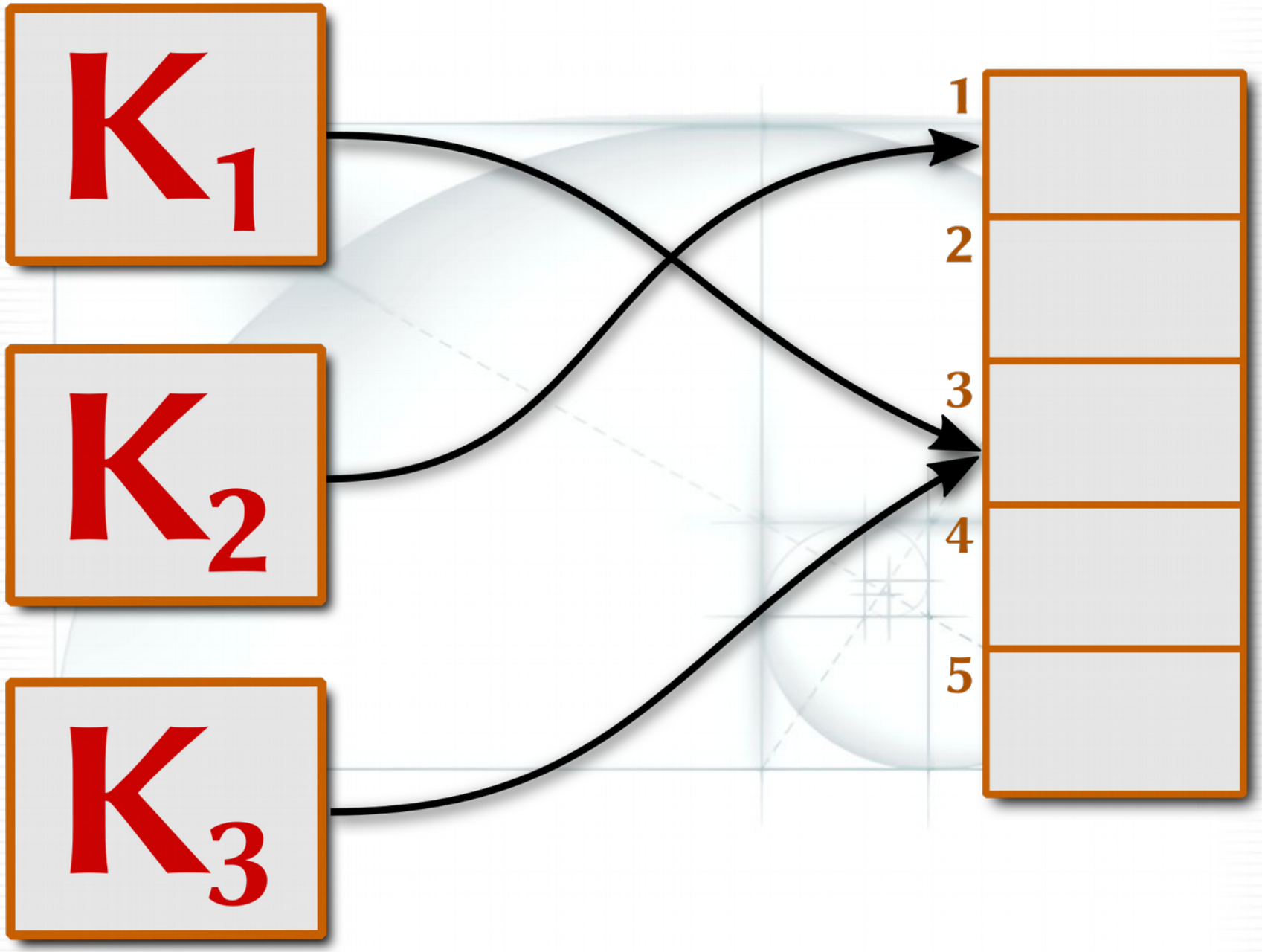

K_1
1
2
3
4
5


K1
K2
1
2
3
4
5


K1
K2
K3
1
2
3
4
5


K1
K2
K3
1
2
3
4
5


Random Hash Collision Rate
Average Keys / Hash Value
2.4
2.2
2.0
1.8
1.6
1.4
1.2
1.0
0.0
0.5
1.0
1.5
2.0
Load Factor -- NumKeys/TableSize


K1
K2
K3
1
2
3
4
5

Buckets


K


K
K

Linked Buckets

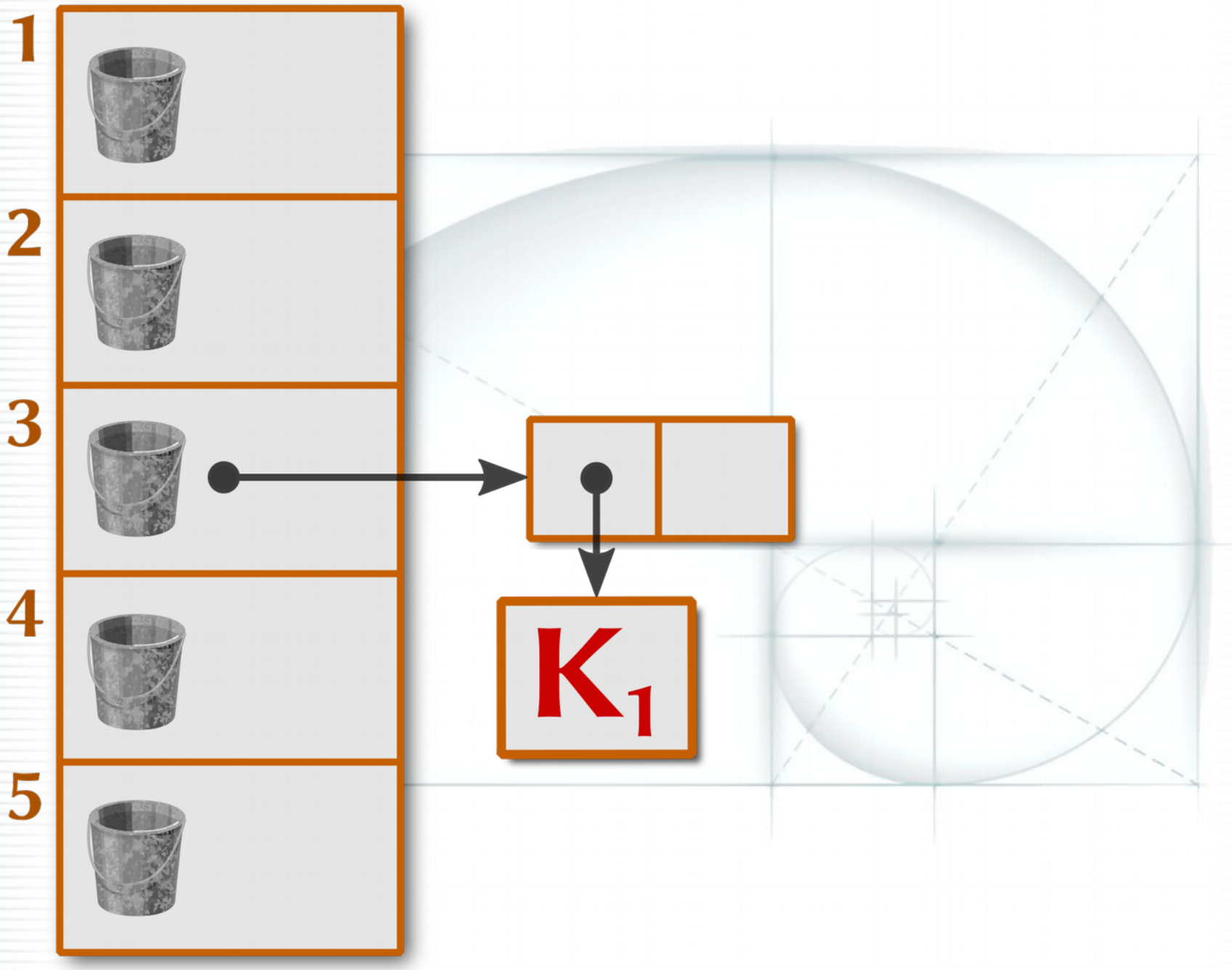
1

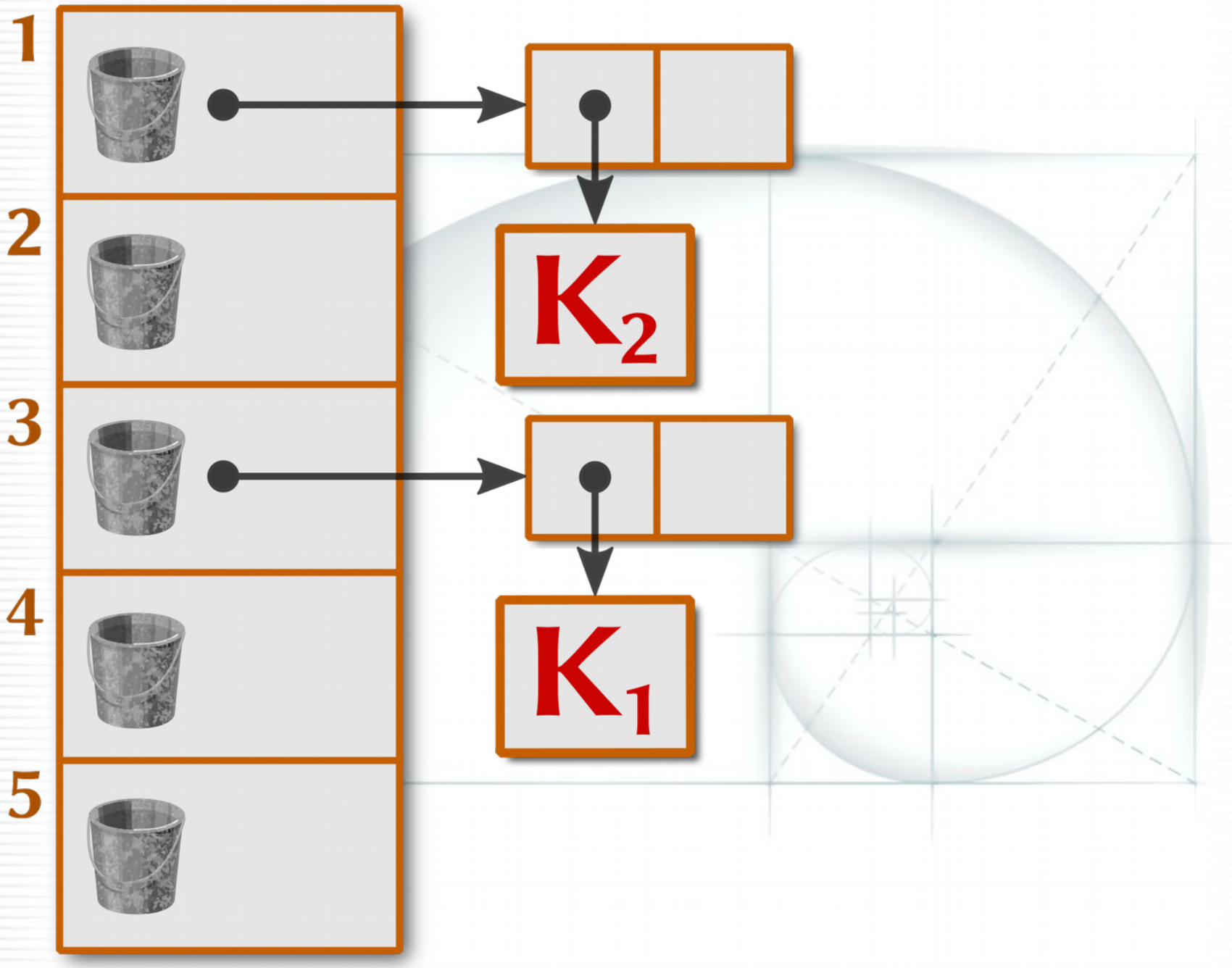
2

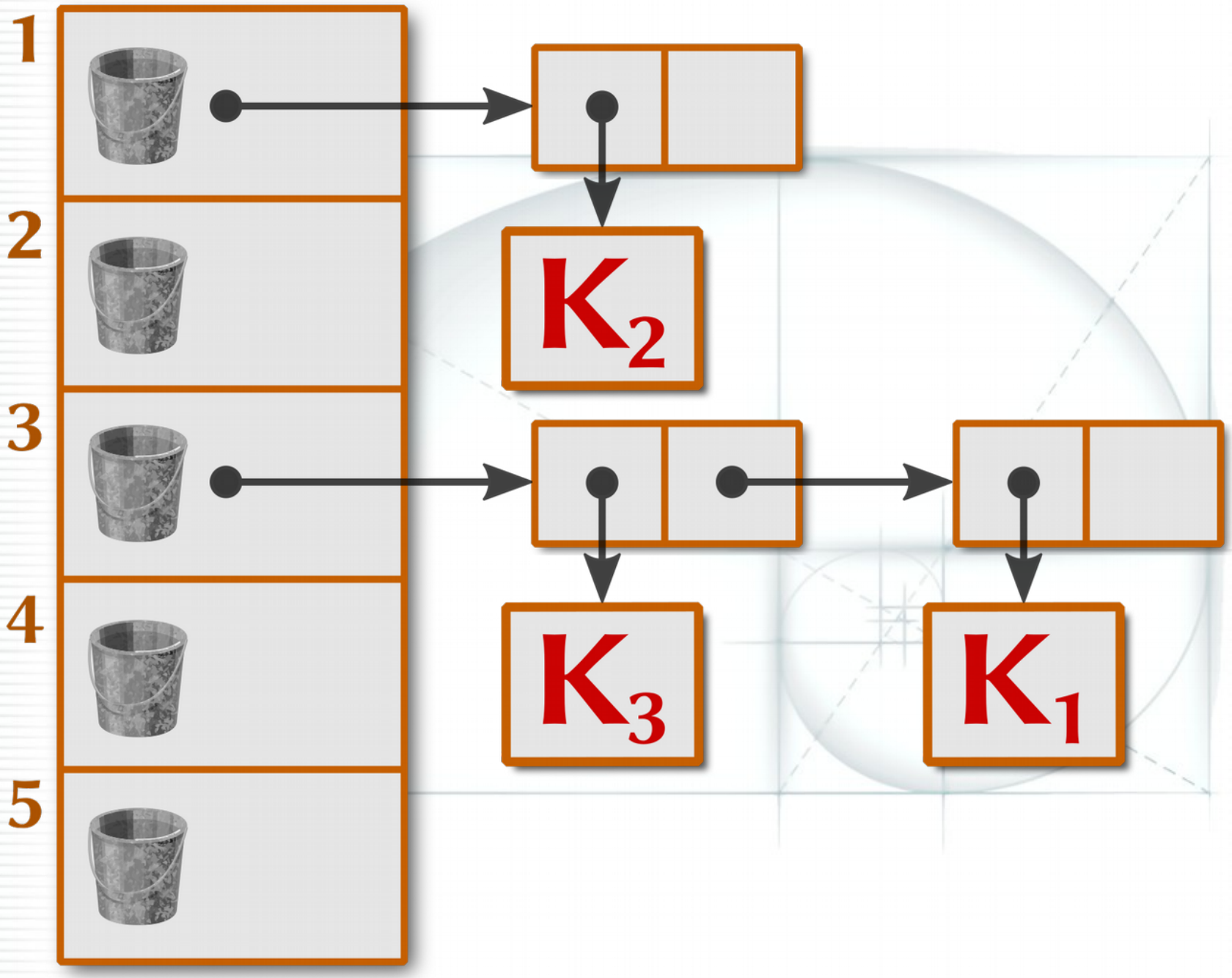
3

4

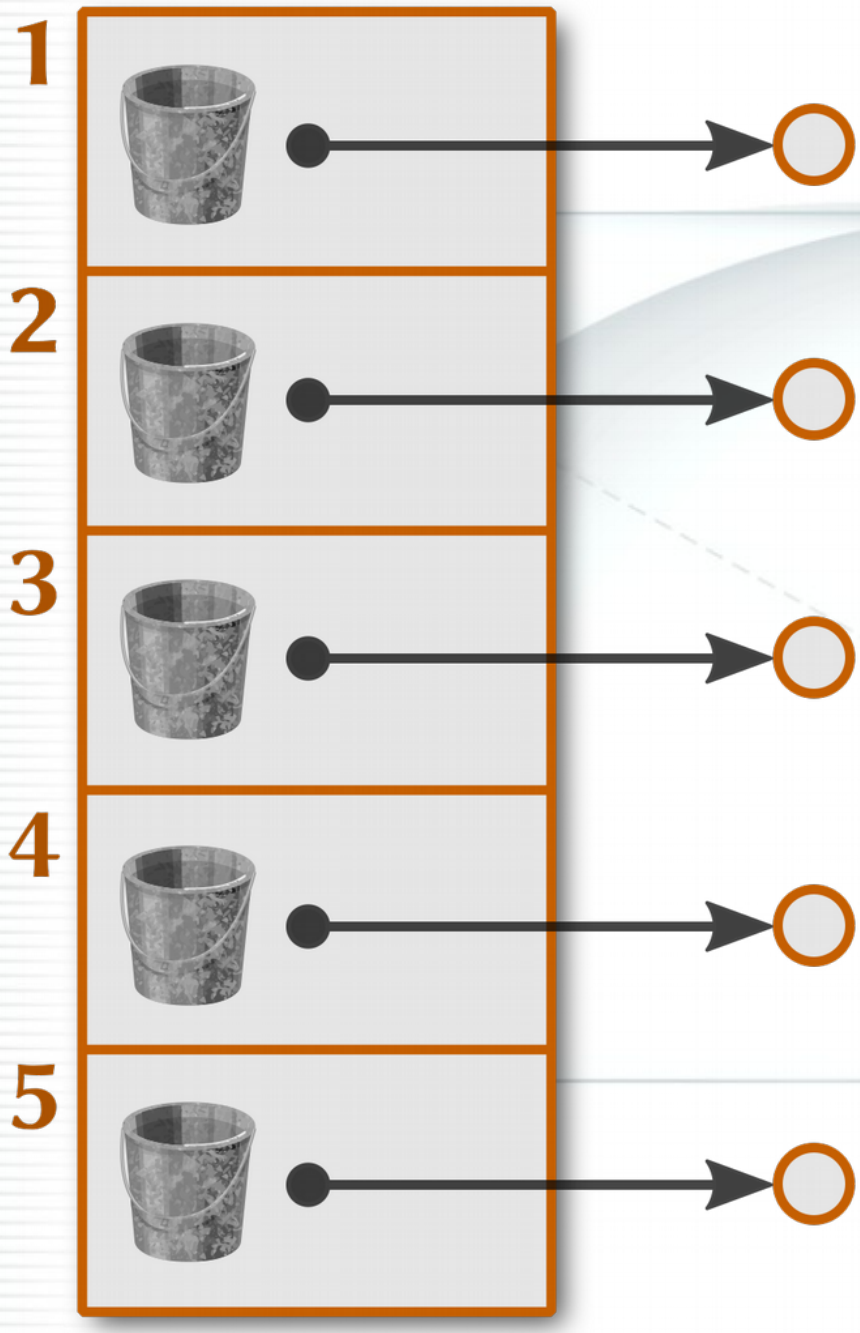
5

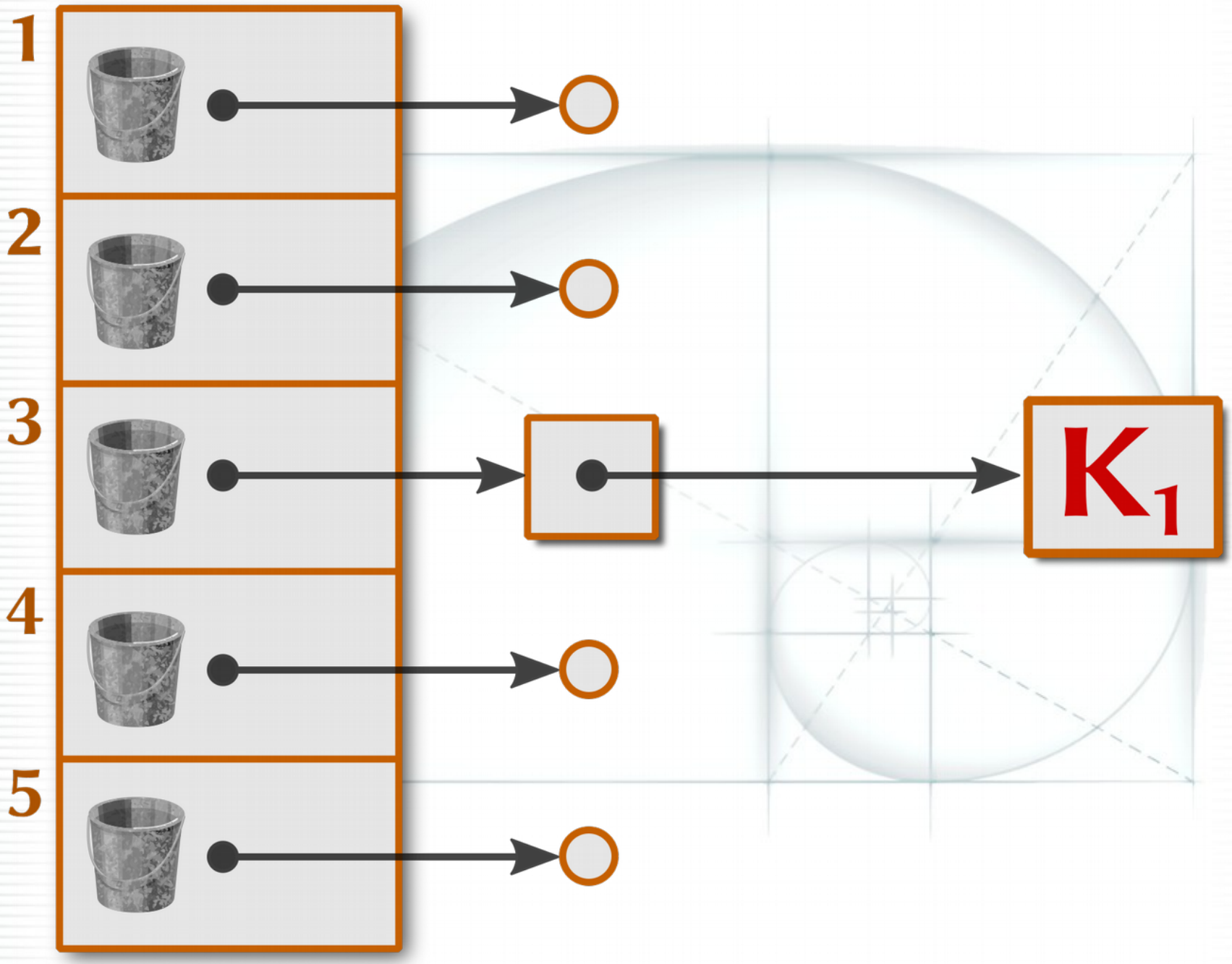

1
2
3
4
5
K1

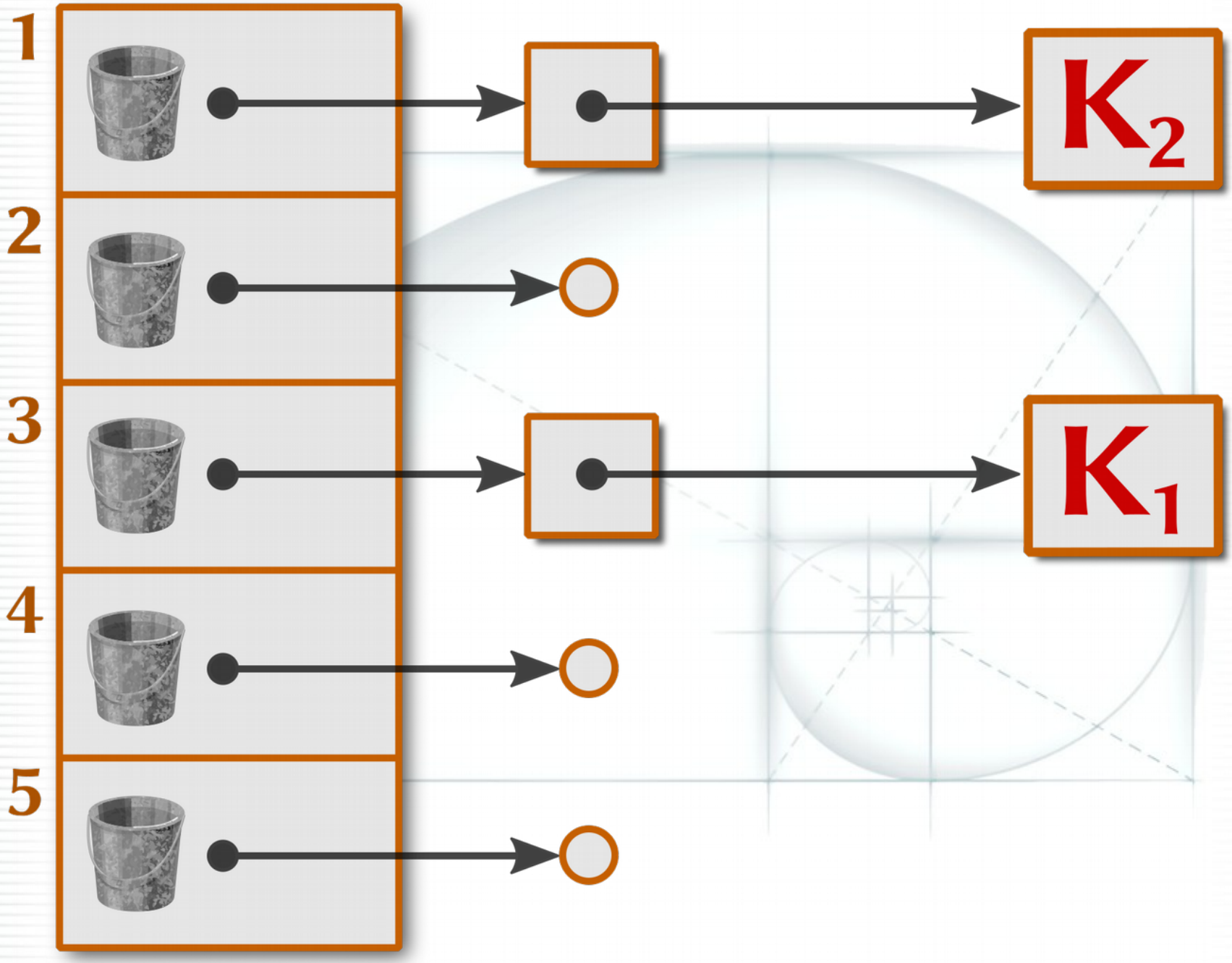

1
2
3
4
5
K2
K1

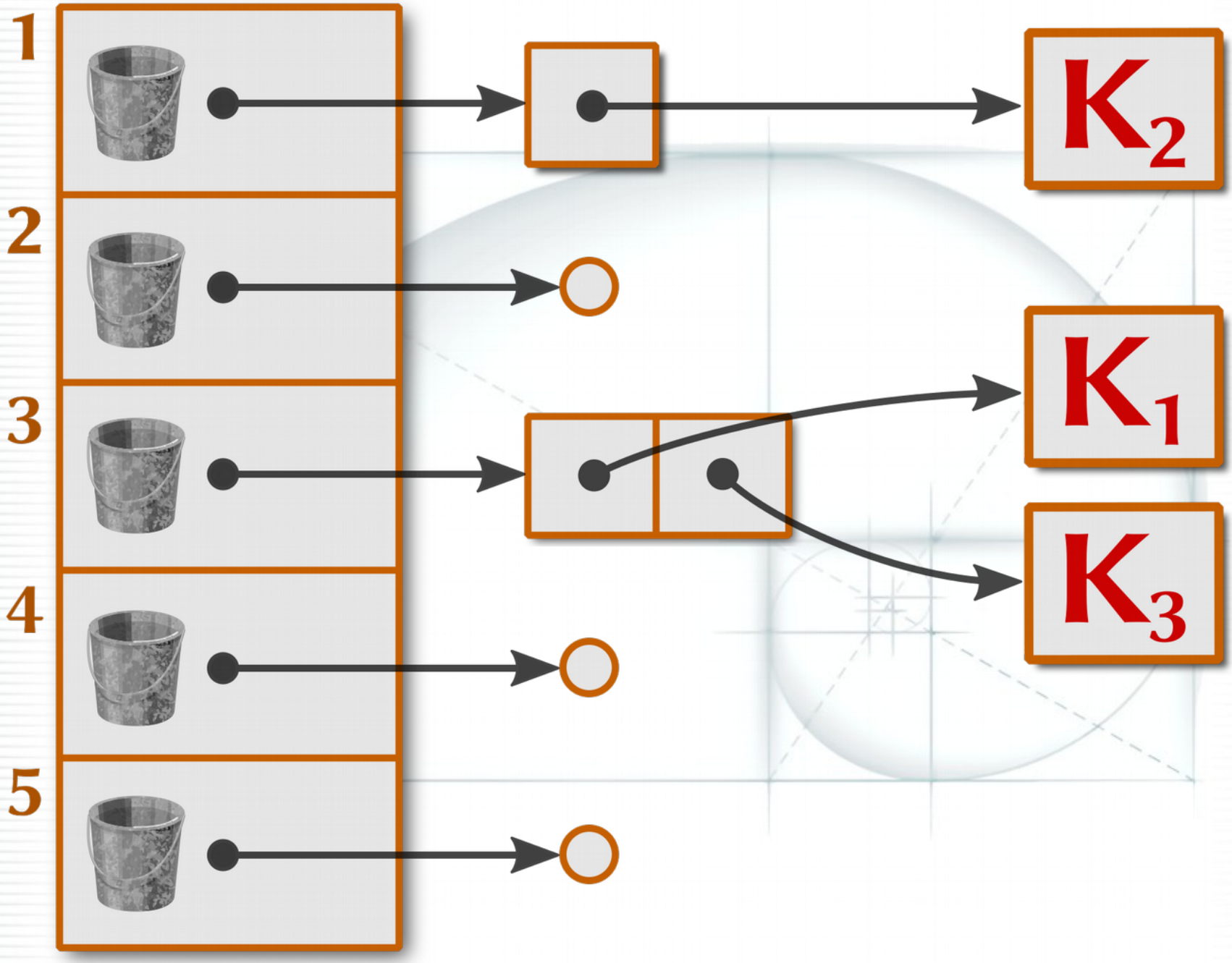

1
2
3
4
5
K2
K3
K1

Array Buckets

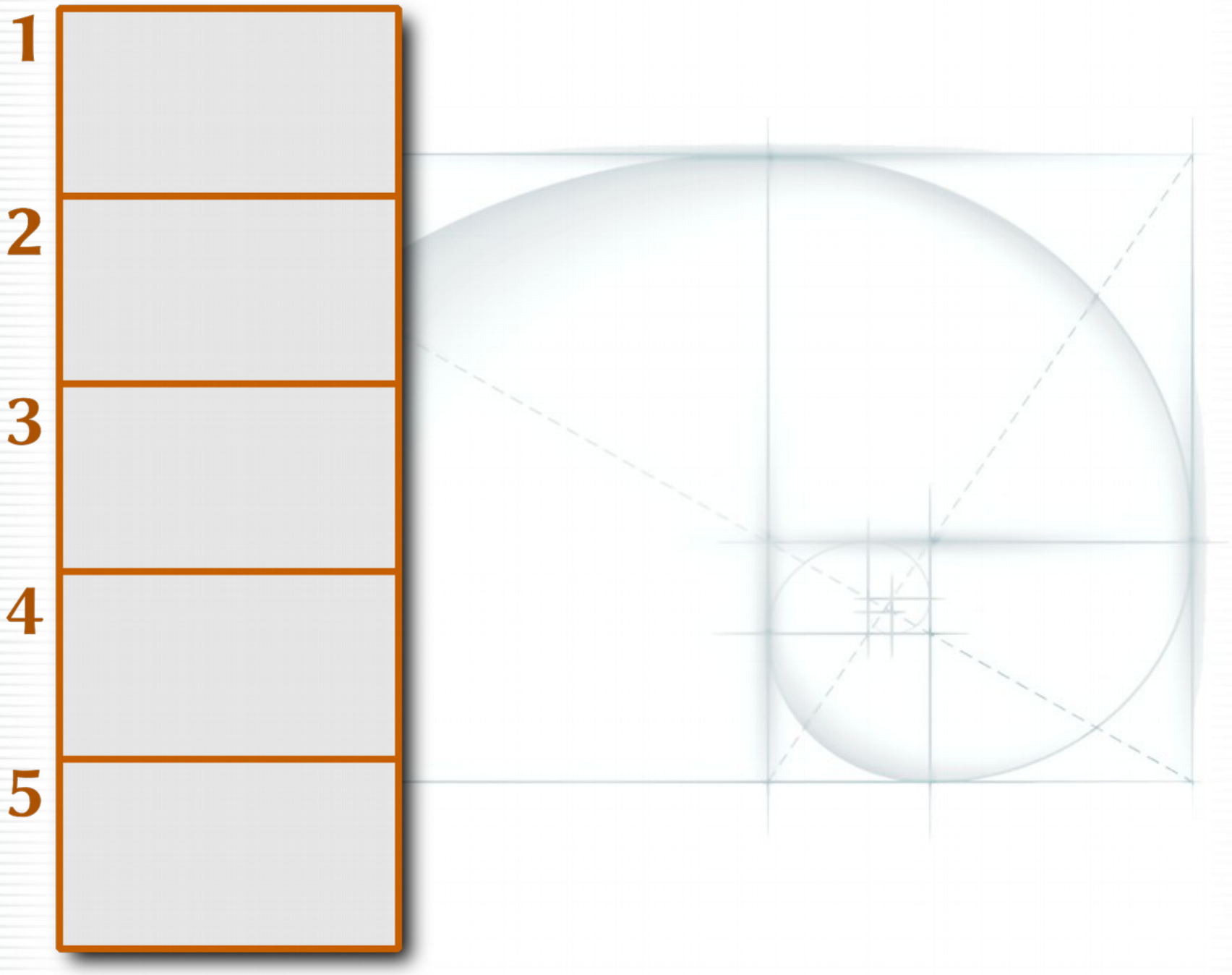

1
2
3
4
5

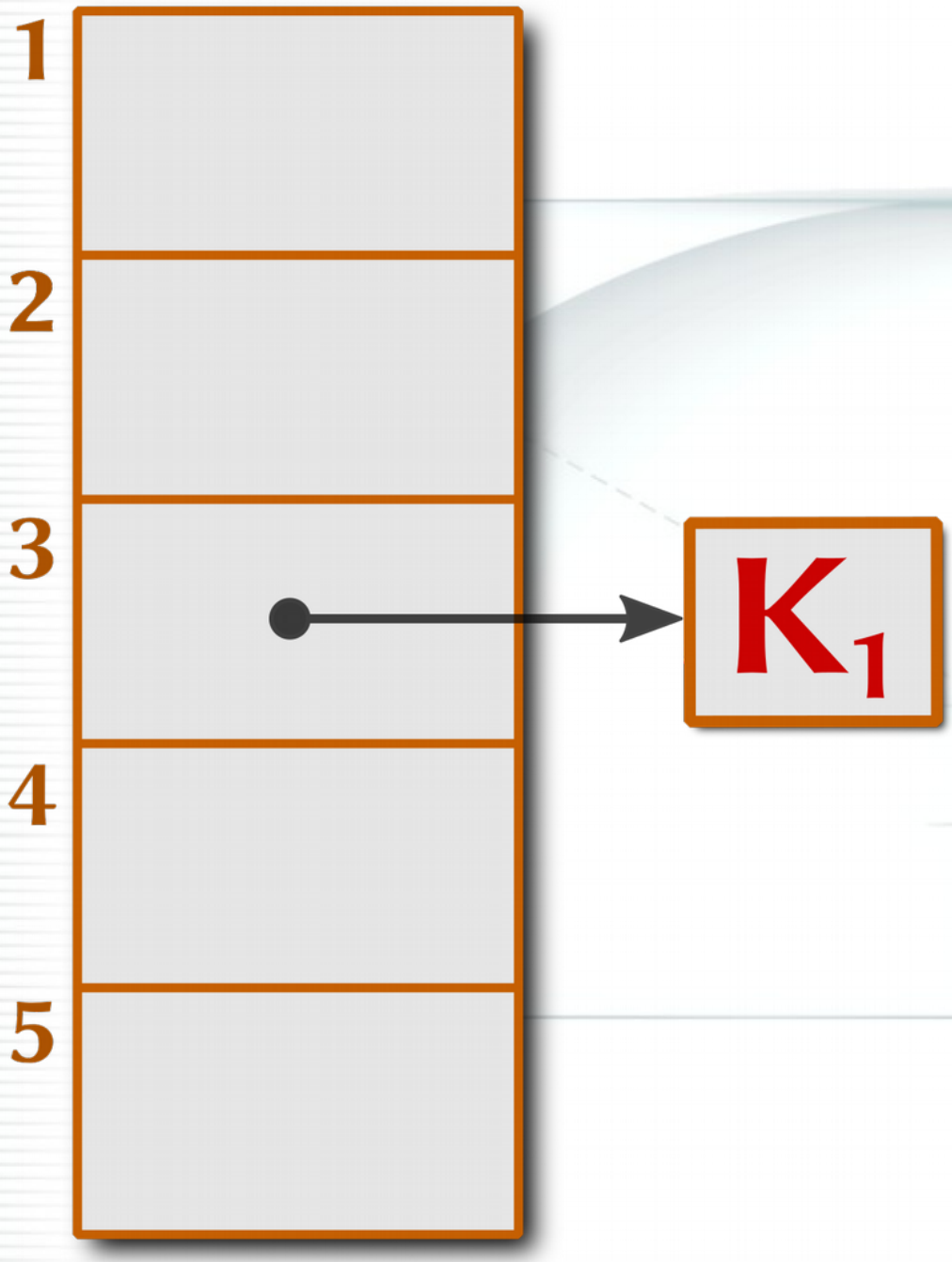

1
2
3
4
5
K1

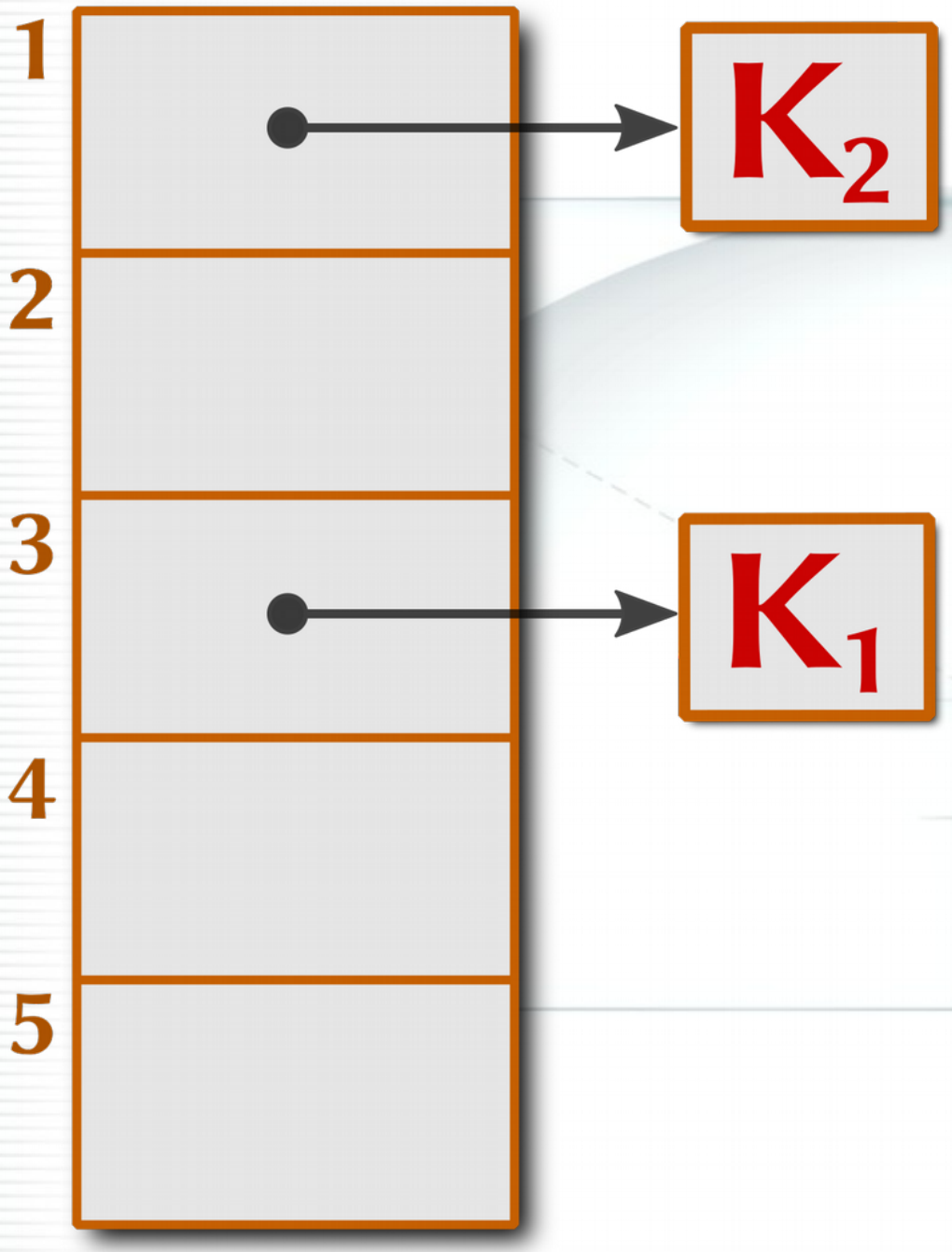

1
2
3
4
5
K2
K1

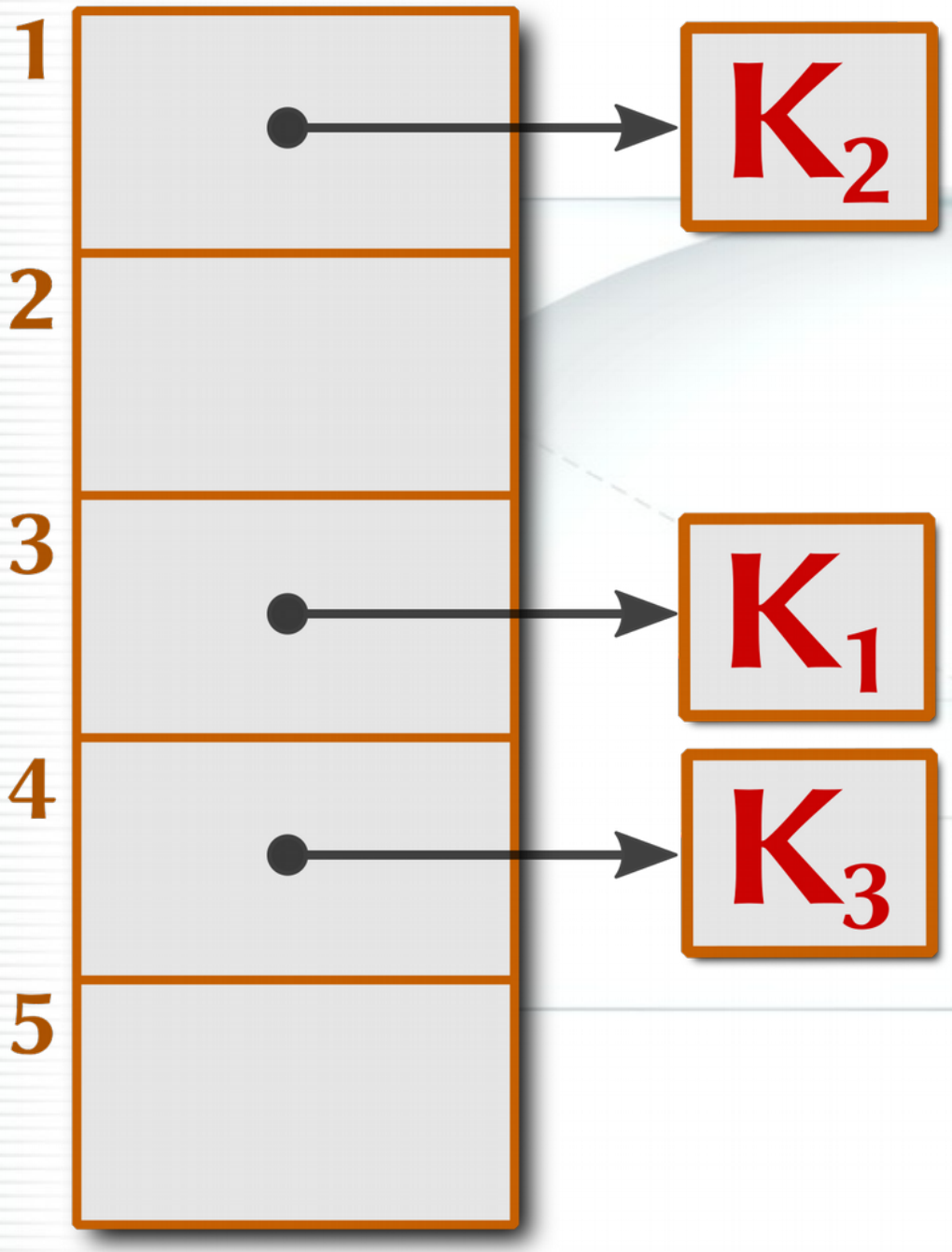

1
2
3
4
5
K2
K1
K3

Linear Probing


1
2
3
4
5


1
2
3
4
5
K1


1
2
3
4
5
K2
K1


1
2
3
4
5
K2
K1
K3

“Lazy” Buckets

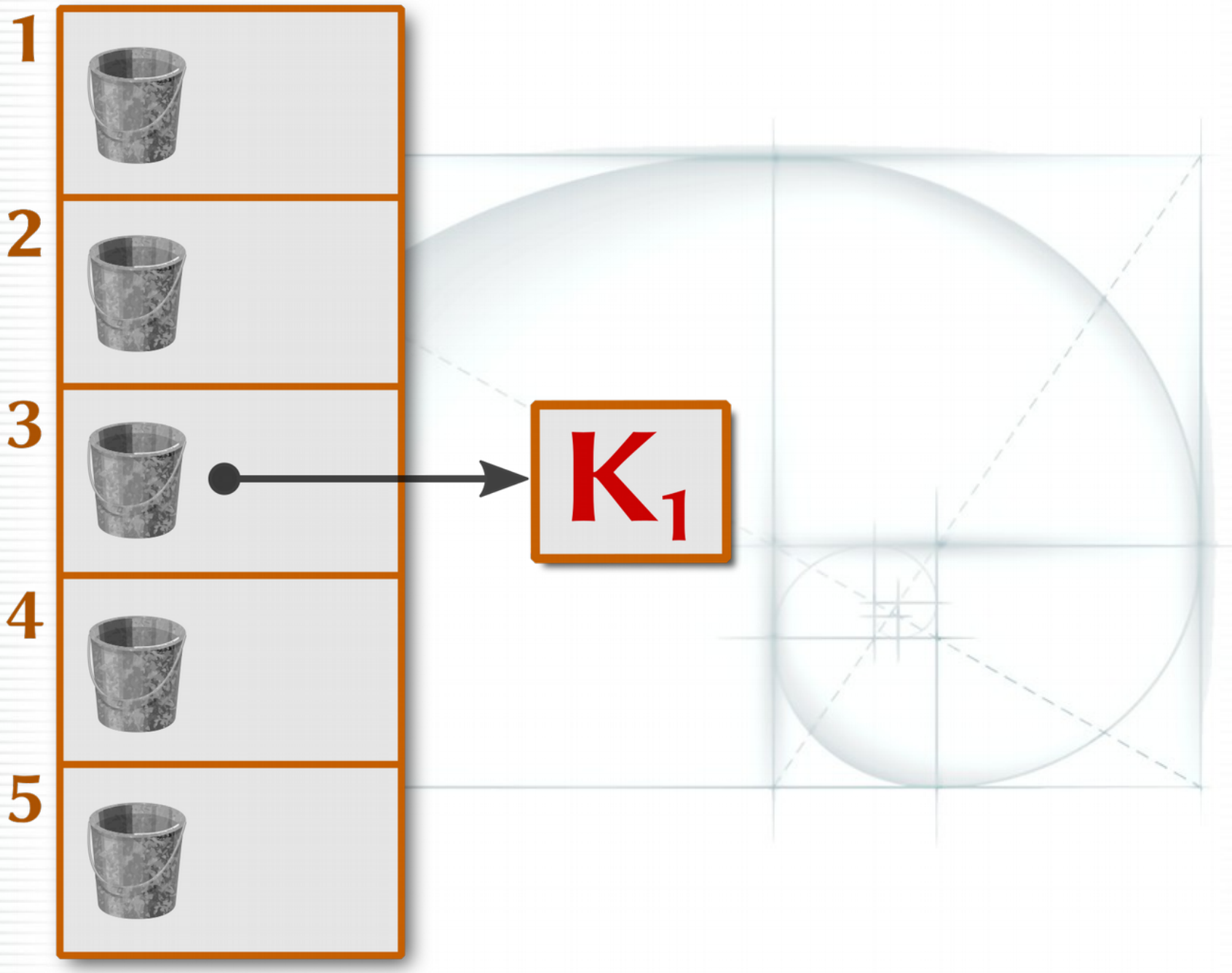
1

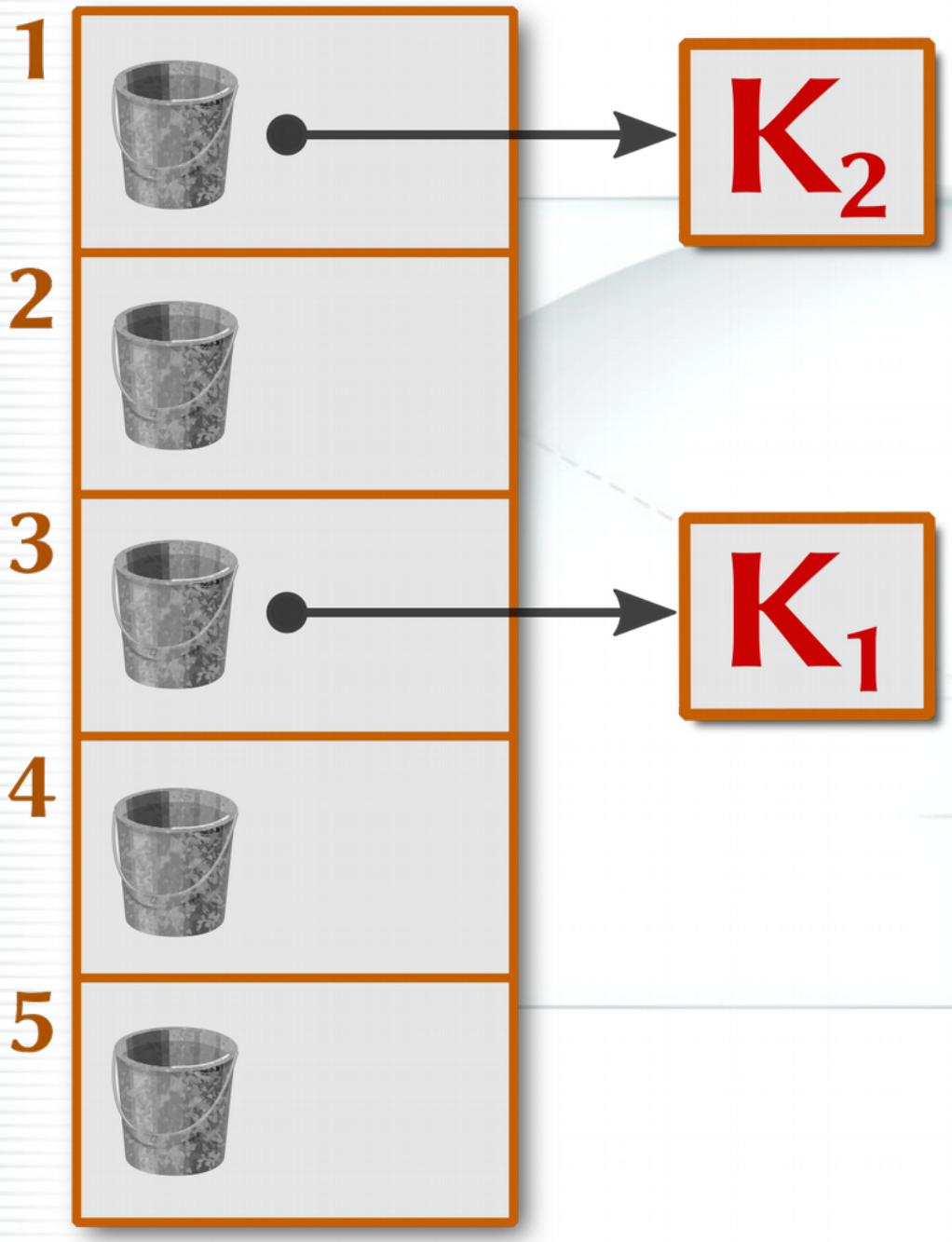
2

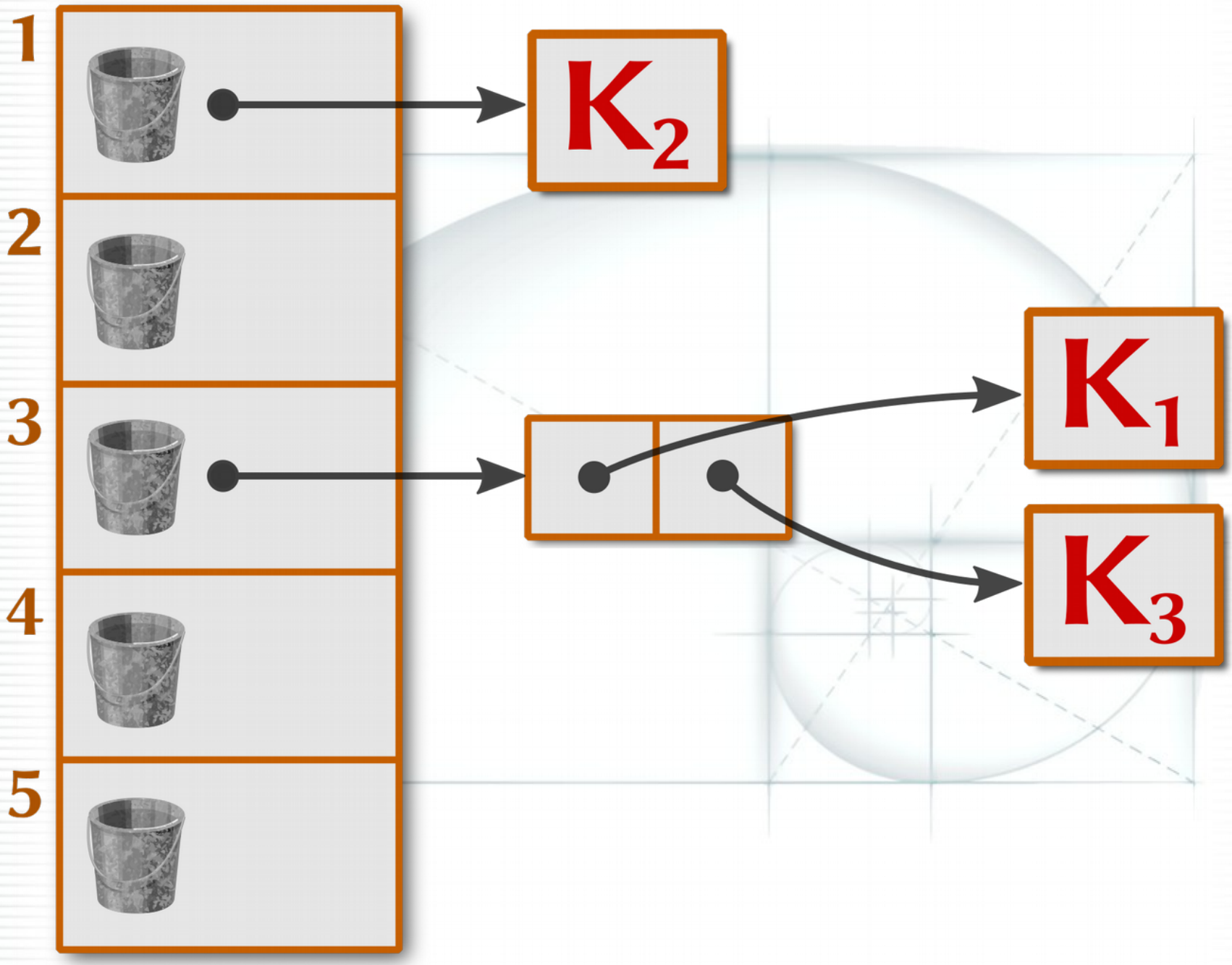
3

4

5

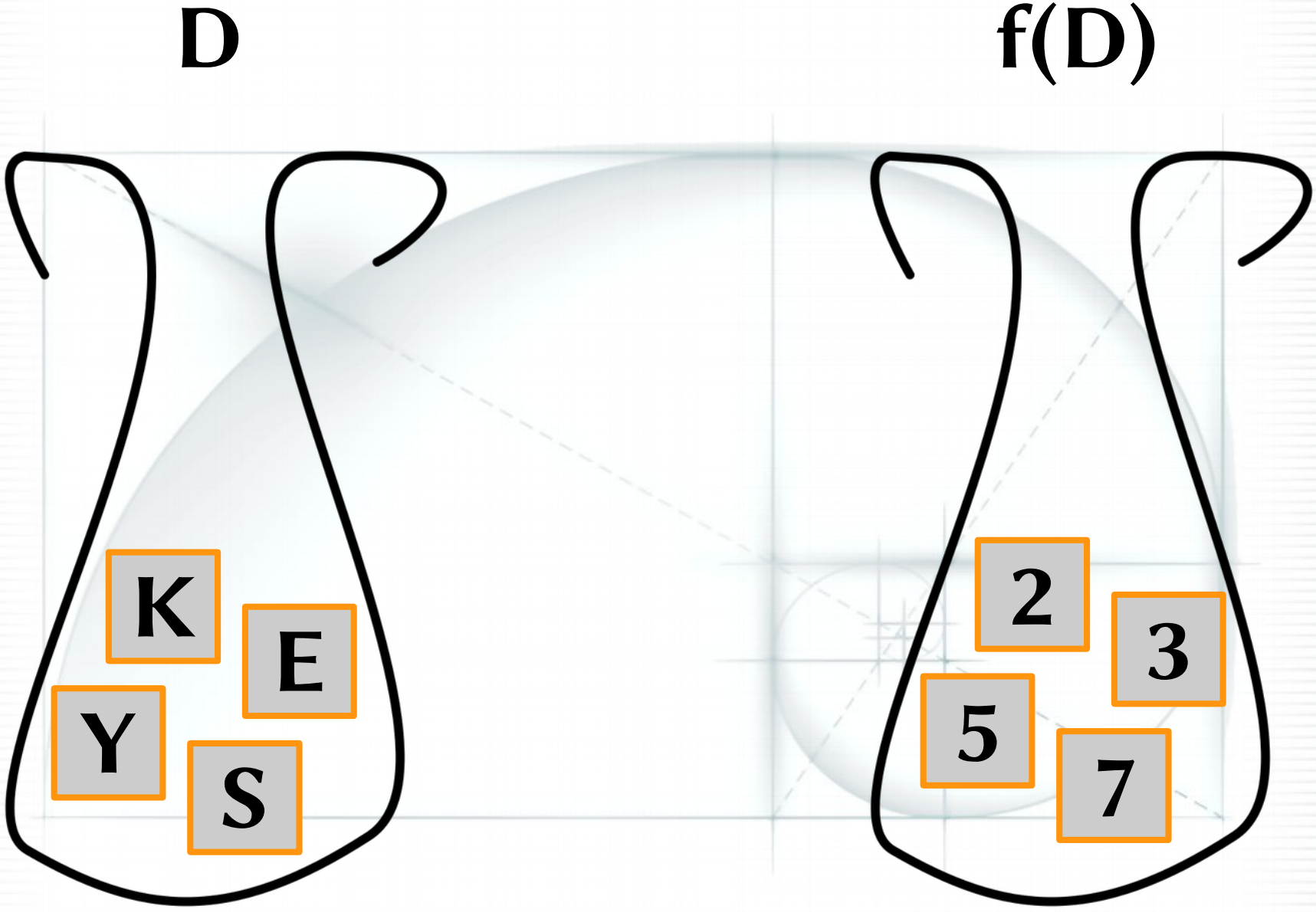

1
2
3
4
5
K_1


1
2
3
4
5
K2
K1


1
2
3
4
5
K2
K1
K3

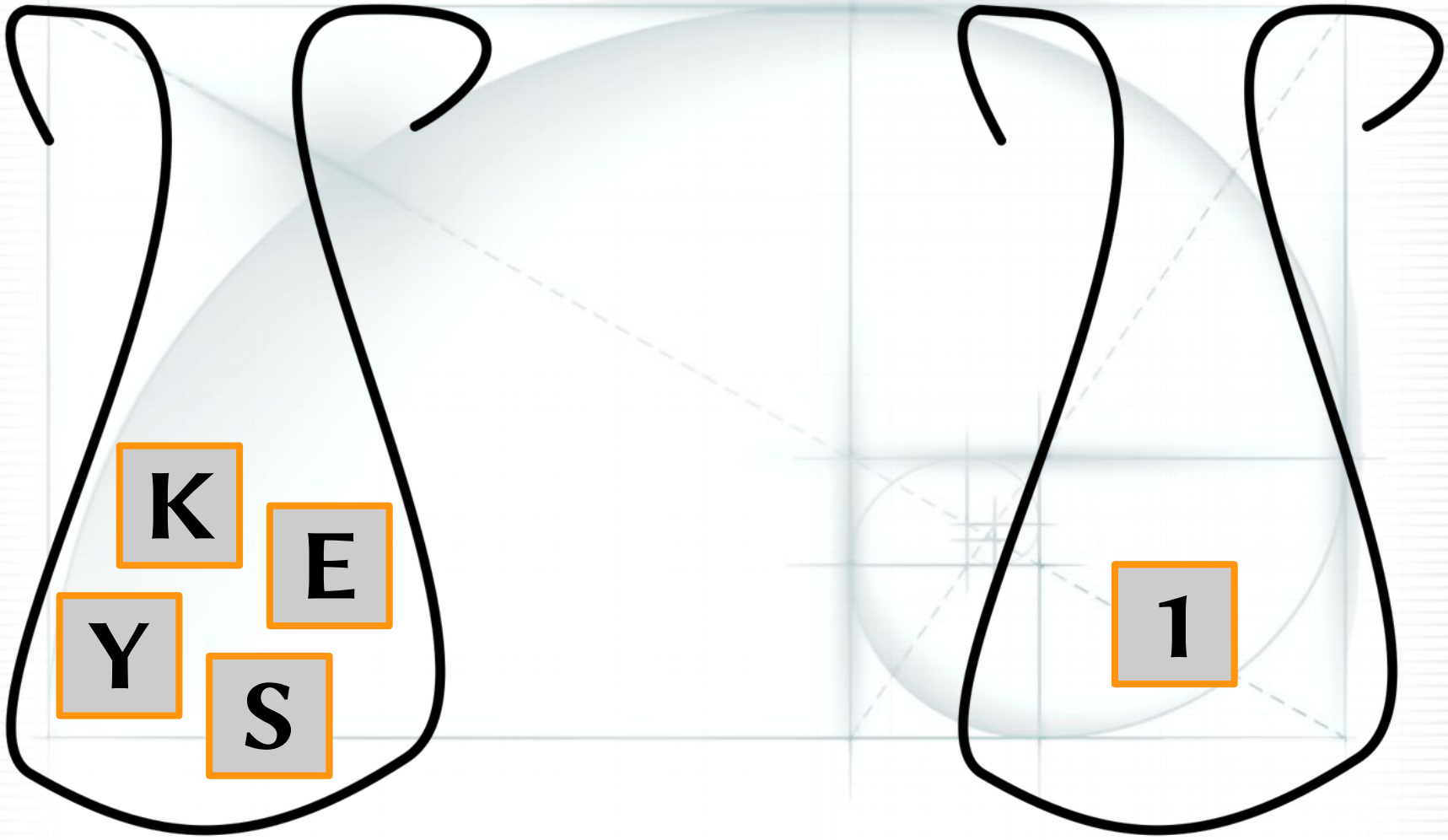
Some Advice

Choosing a Hash Function


D
f(D)
K
E
Y
S
2
3
5
7
Data Set
#hash Values

D

f(D)

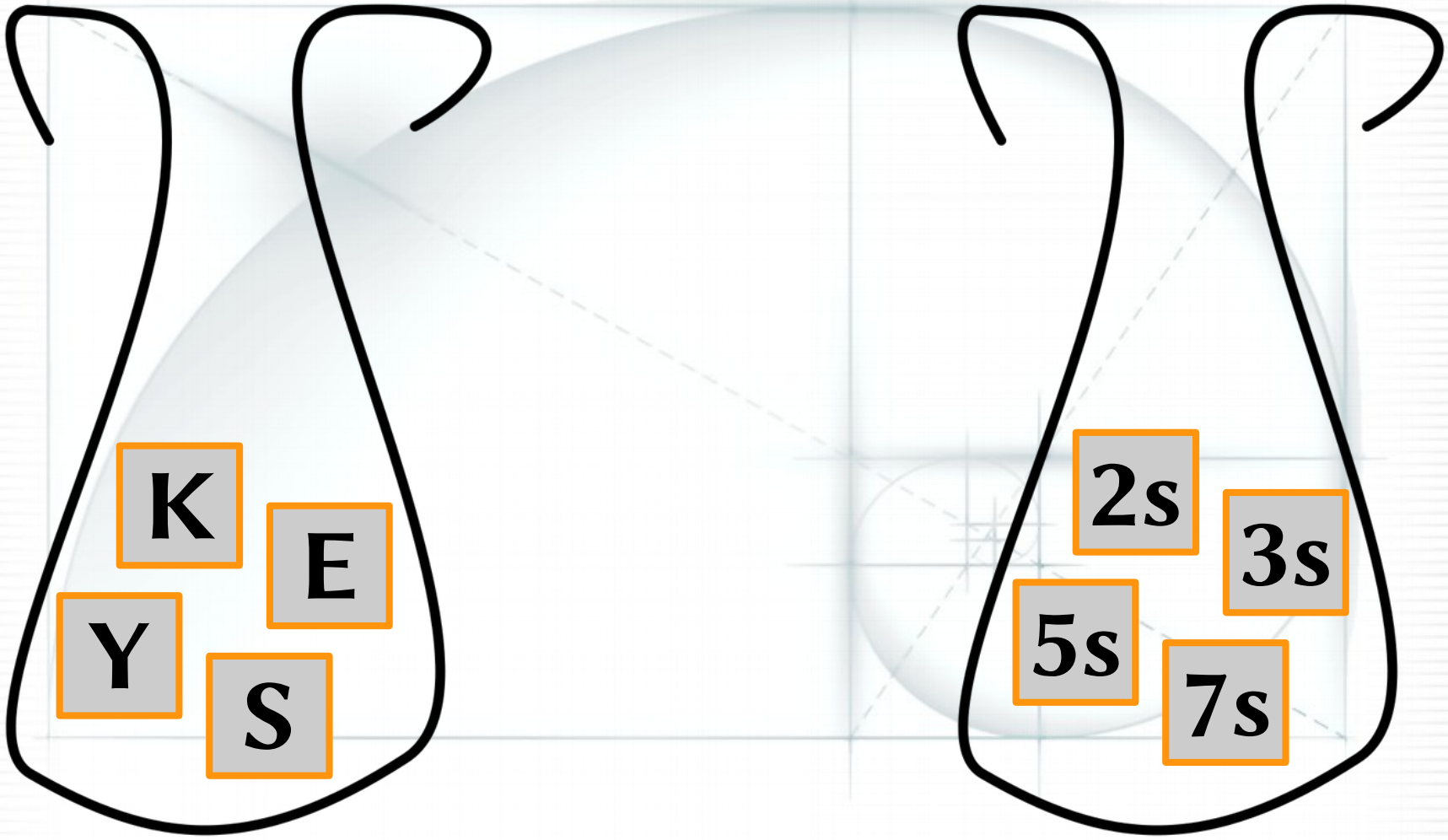


Data Set

#hash Values

D

f(D)

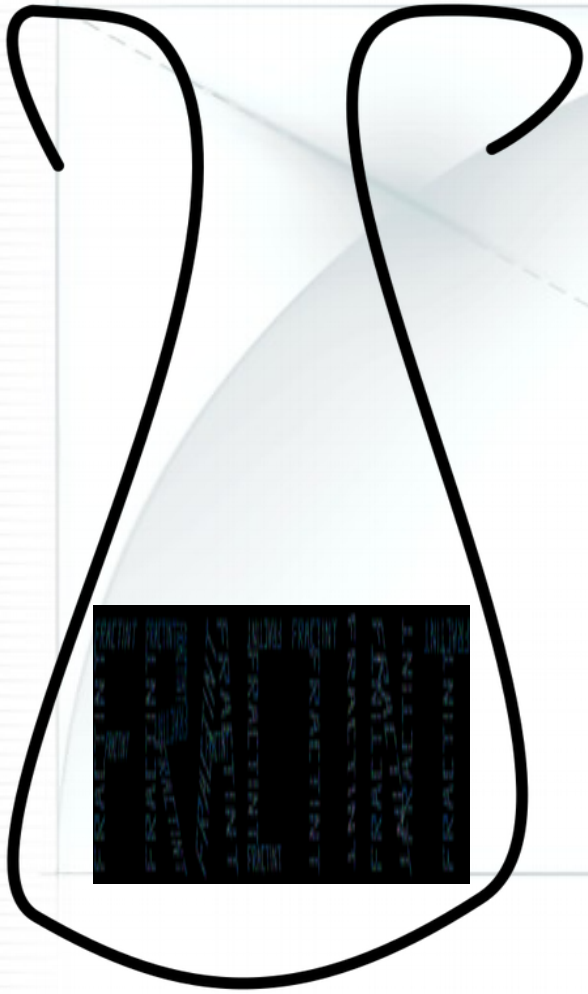


Data Set

#hash Values

D

f(D)



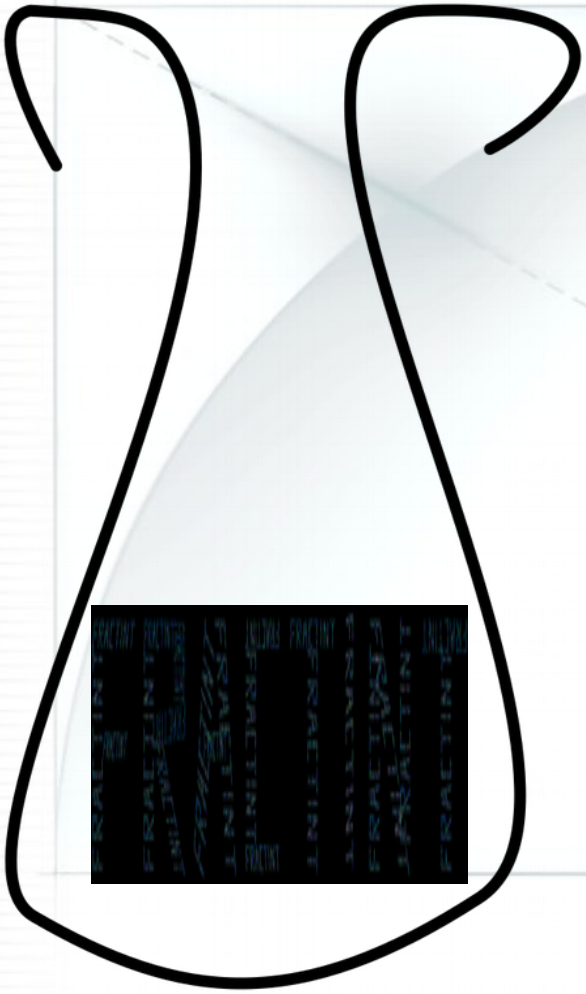
random
mod s

Data Set

#hash Values

D

f(D)



crypto
hash

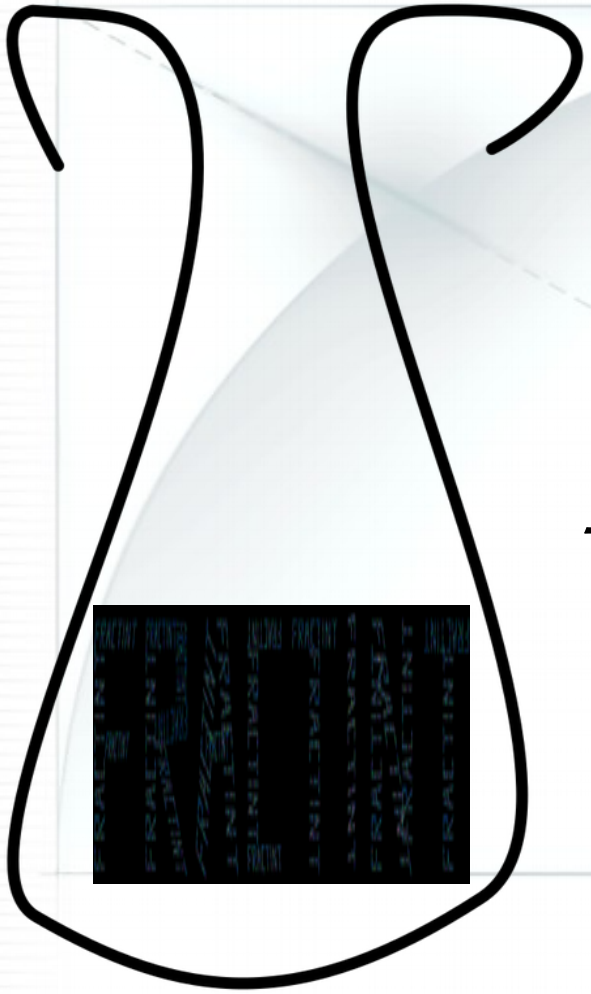
random
mod s

Data Set

#hash Values

D

f(D)



“hash function”

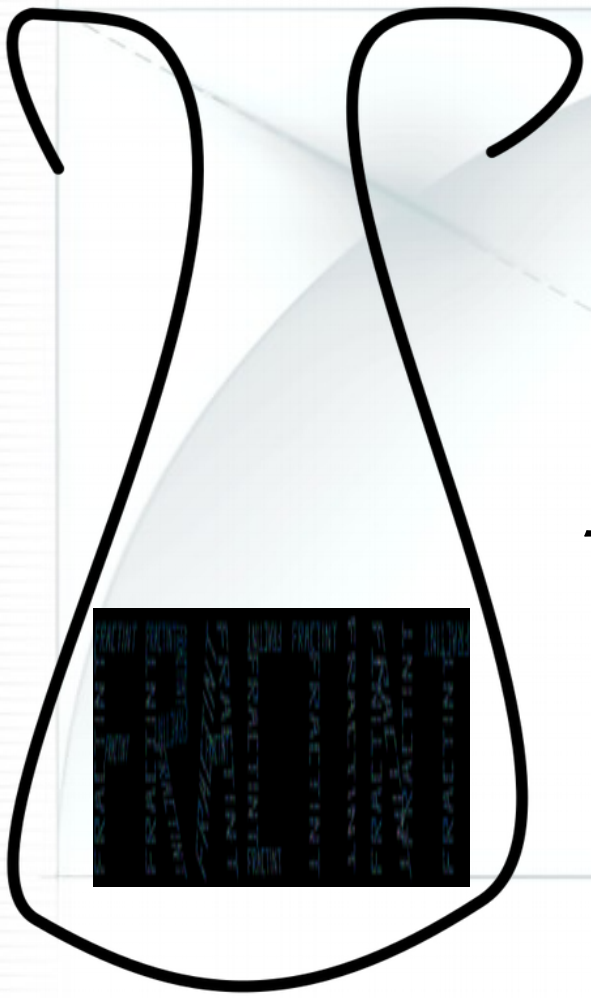
maybe random mod s

Data Set

#hash Values

D

f(D)



"hash function"

prime s

maybe random mod s

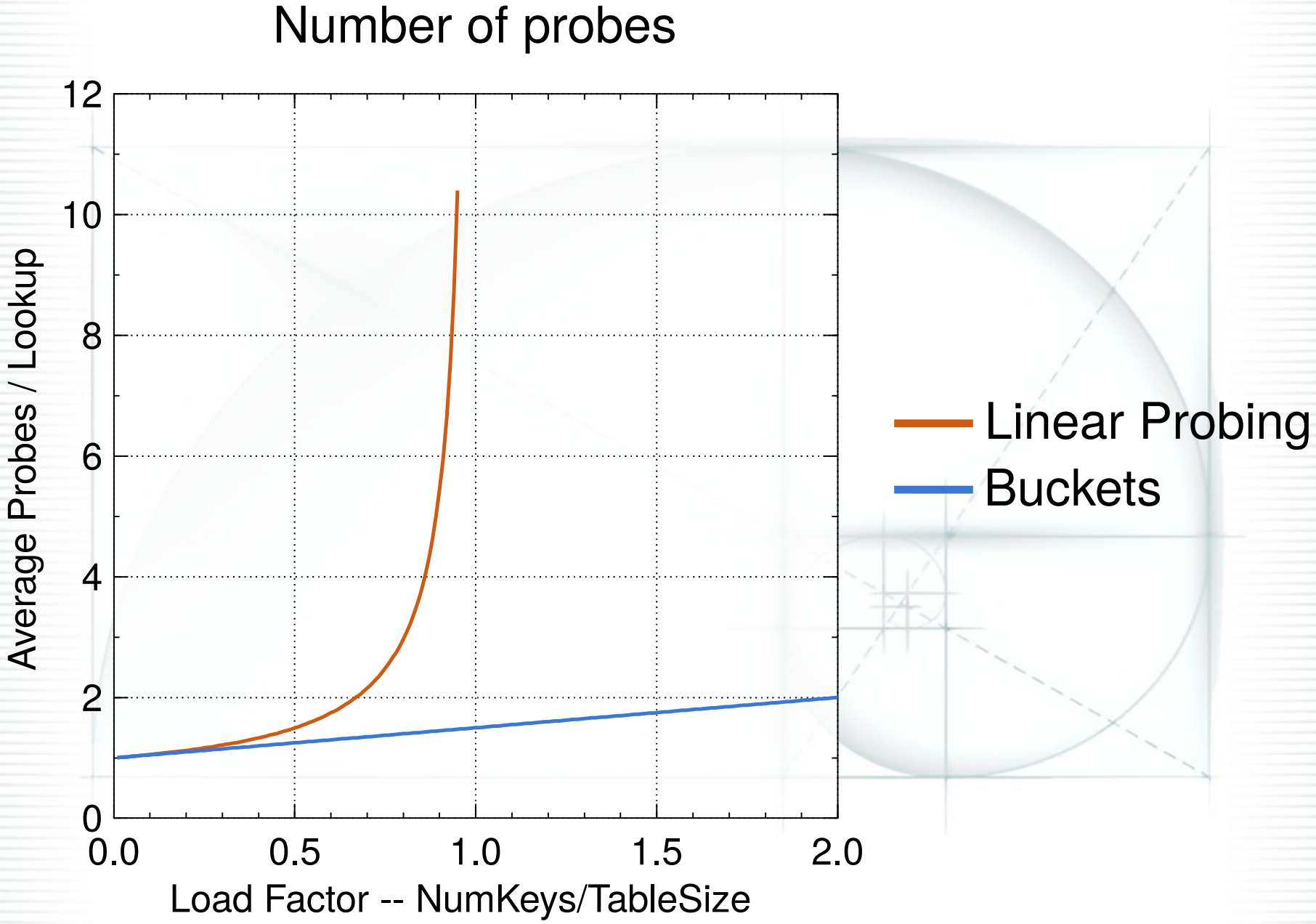
Data Set

#hash Values

Collisions

Collisions

…again


Number of probes
Average Probes / Lookup
Load Factor -- NumKeys/TableSize
0
2
4
6
8
10
12
0.0
0.5
1.0
1.5
2.0
Linear Probing
Buckets

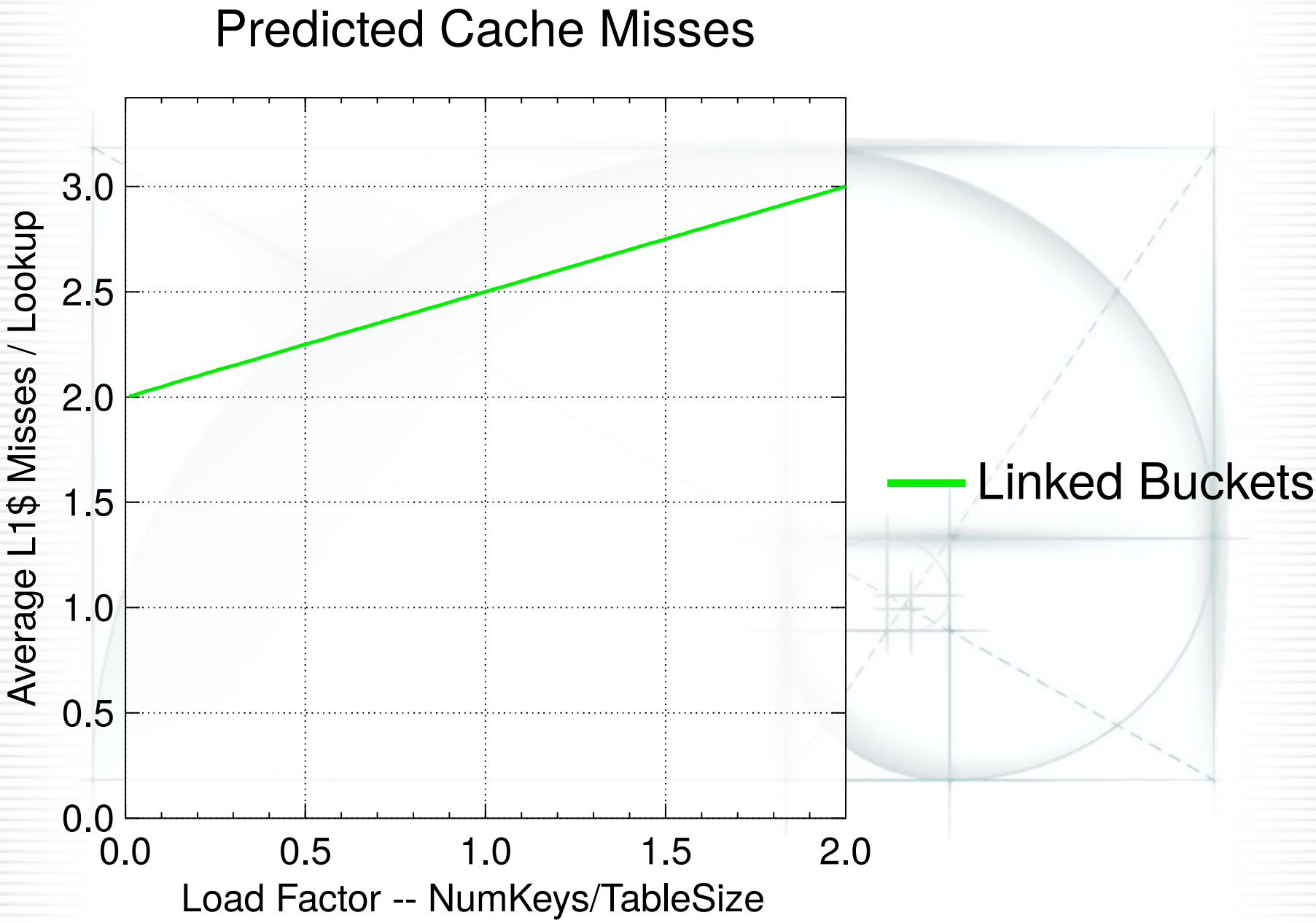
€/probe

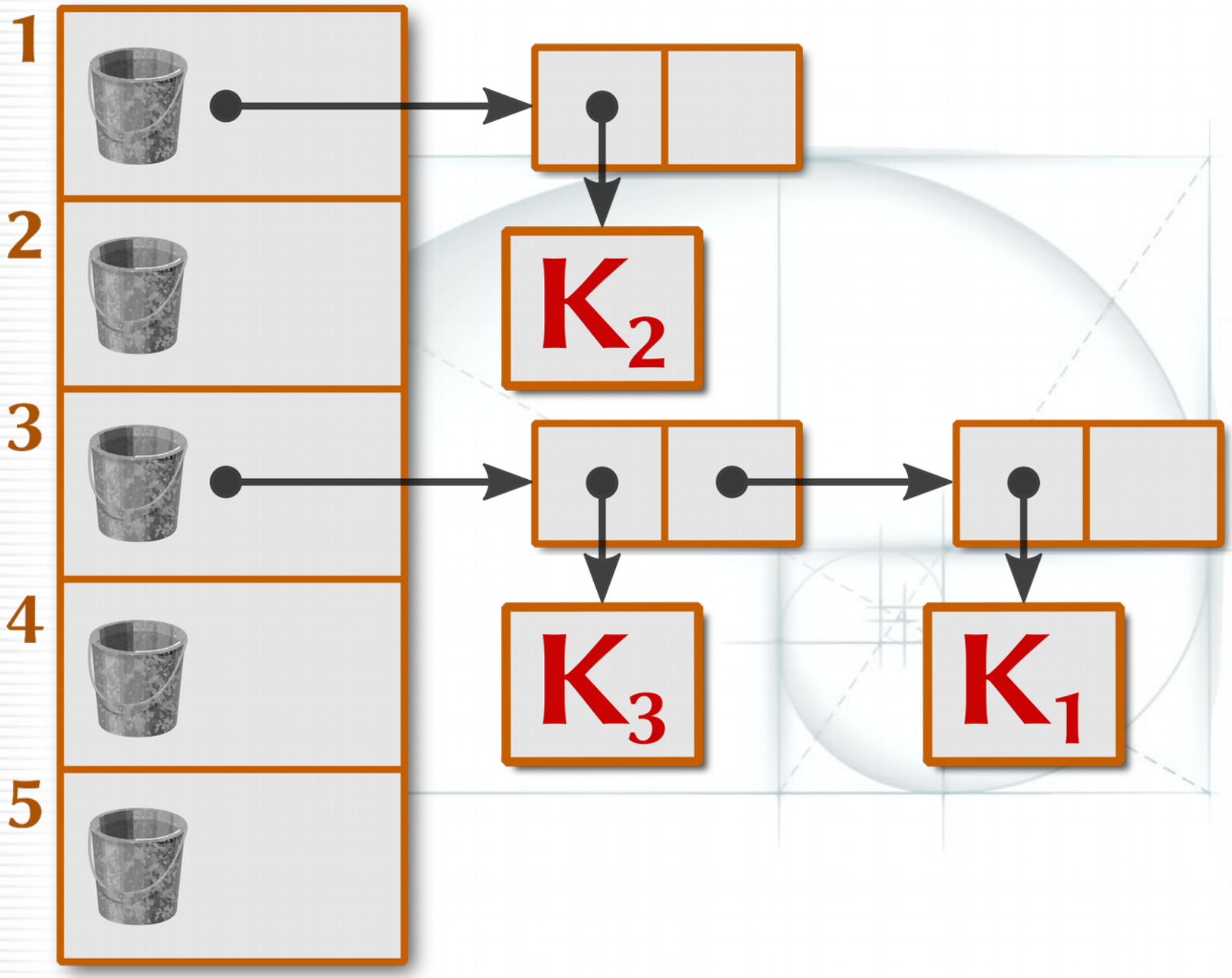
≠

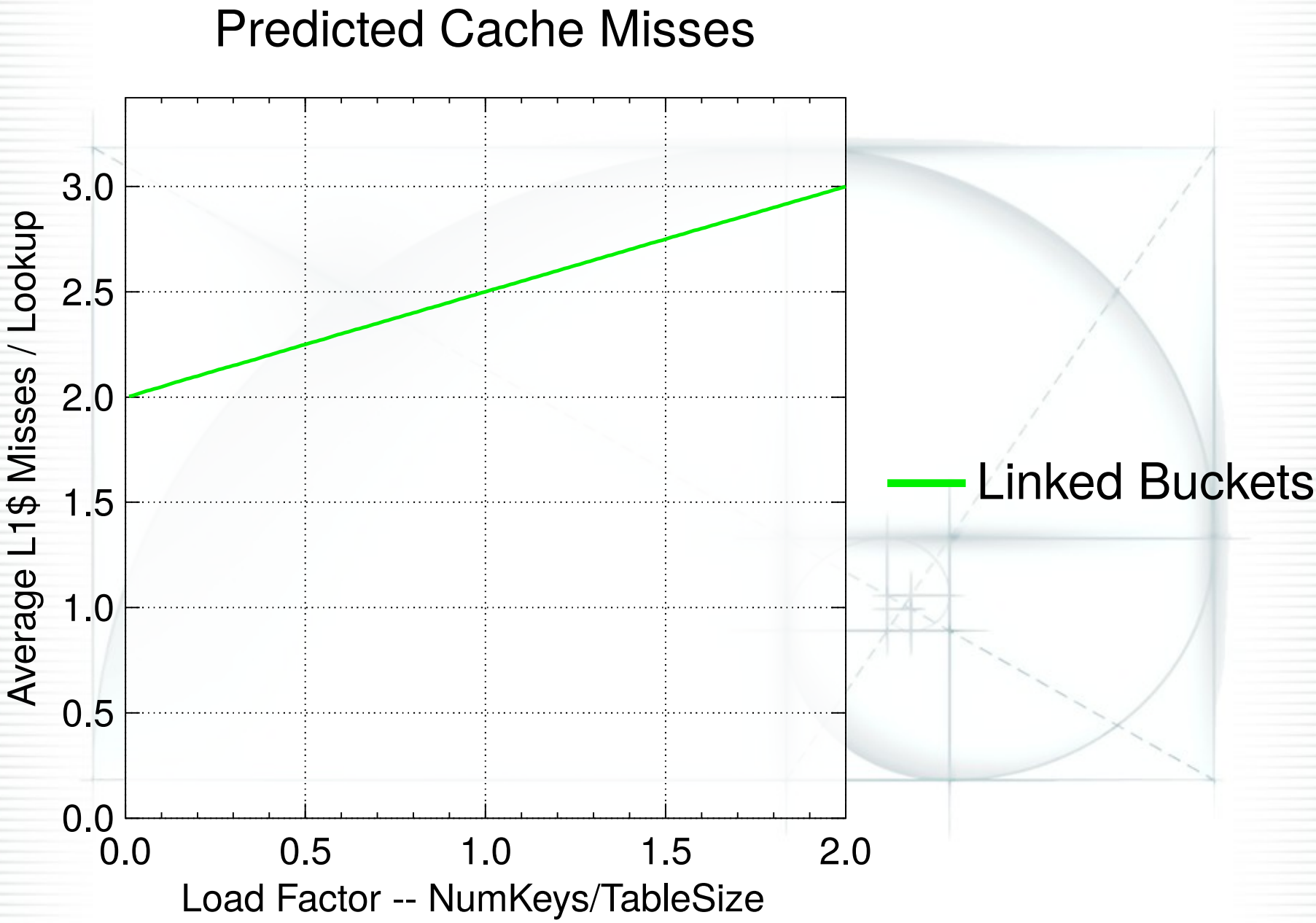
Hypothesis:

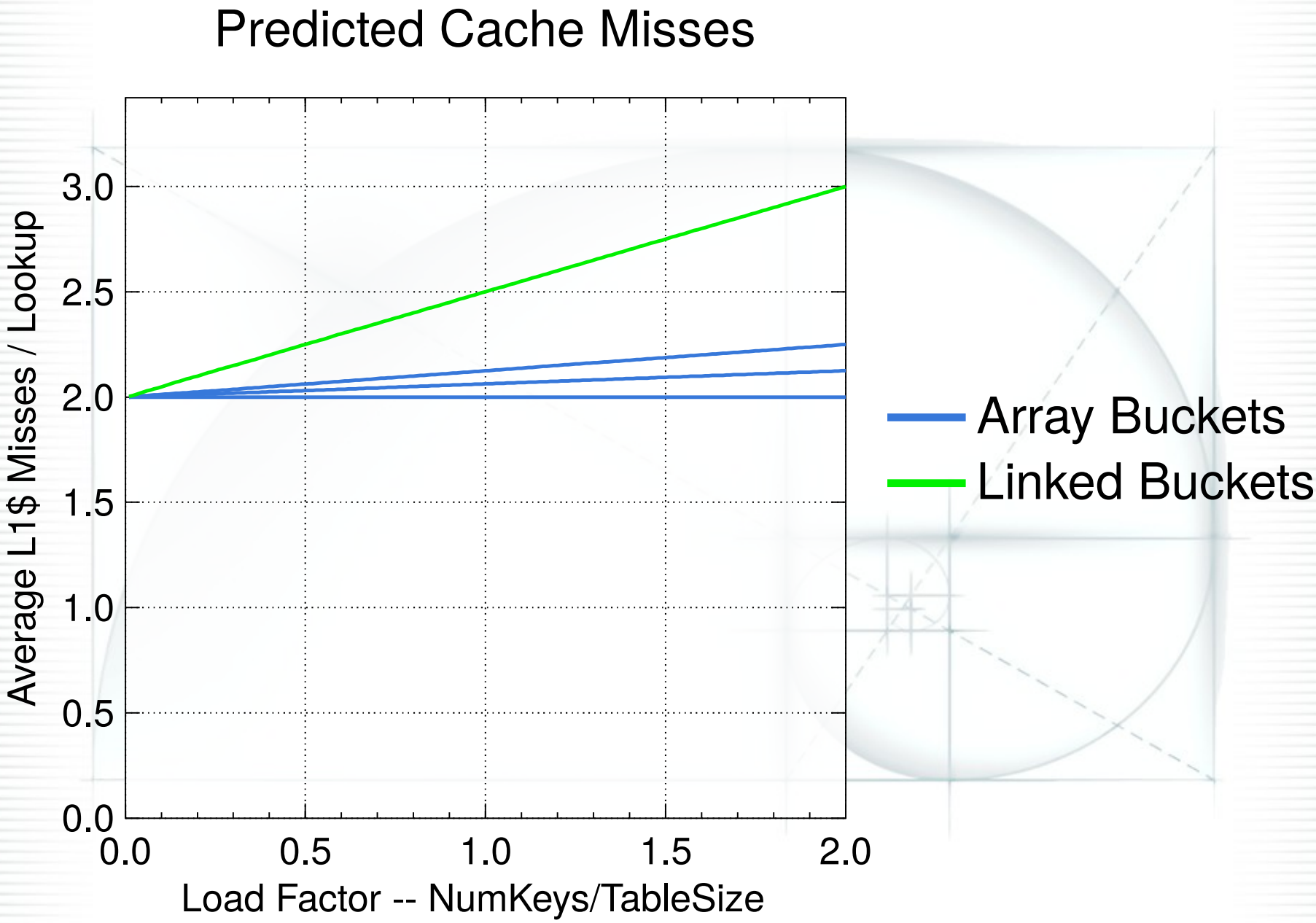
€ ∝

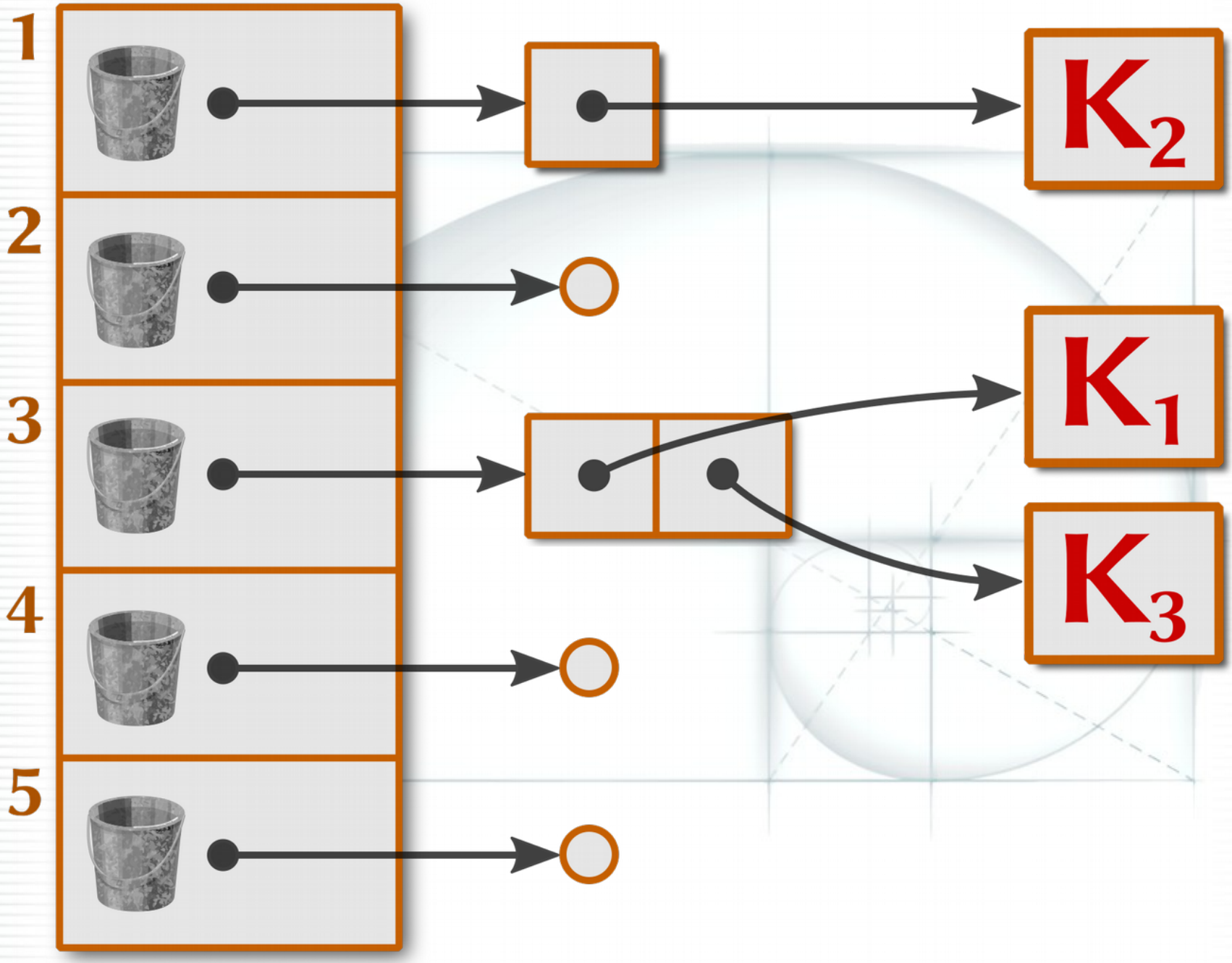
L1$ misses

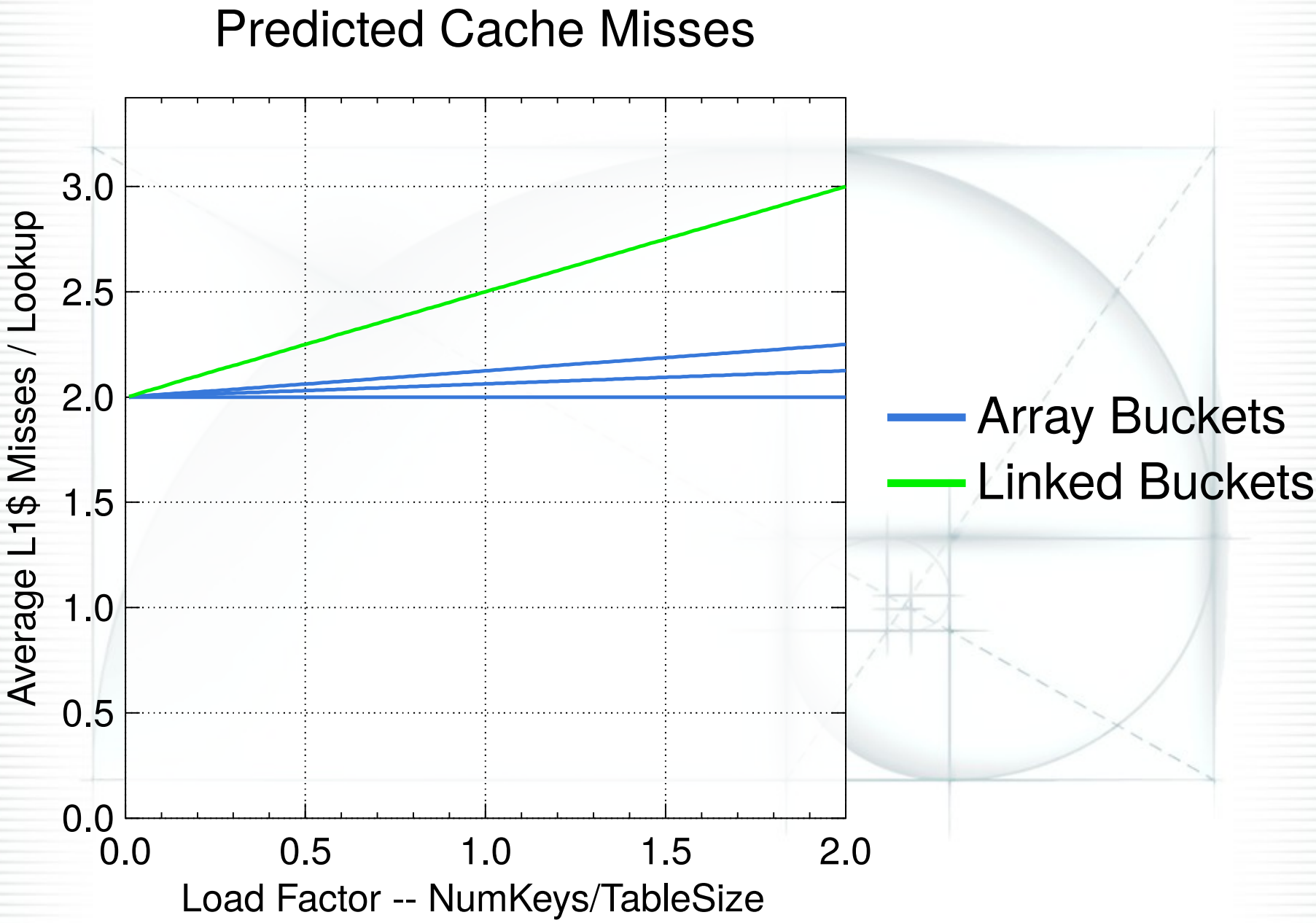

Predicted Cache Misses
Average L1$ Misses / Lookup
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.5
1.0
1.5
2.0
Load Factor -- NumKeys/TableSize
Linked Buckets


1
2
3
4
5
K2
K3
K1

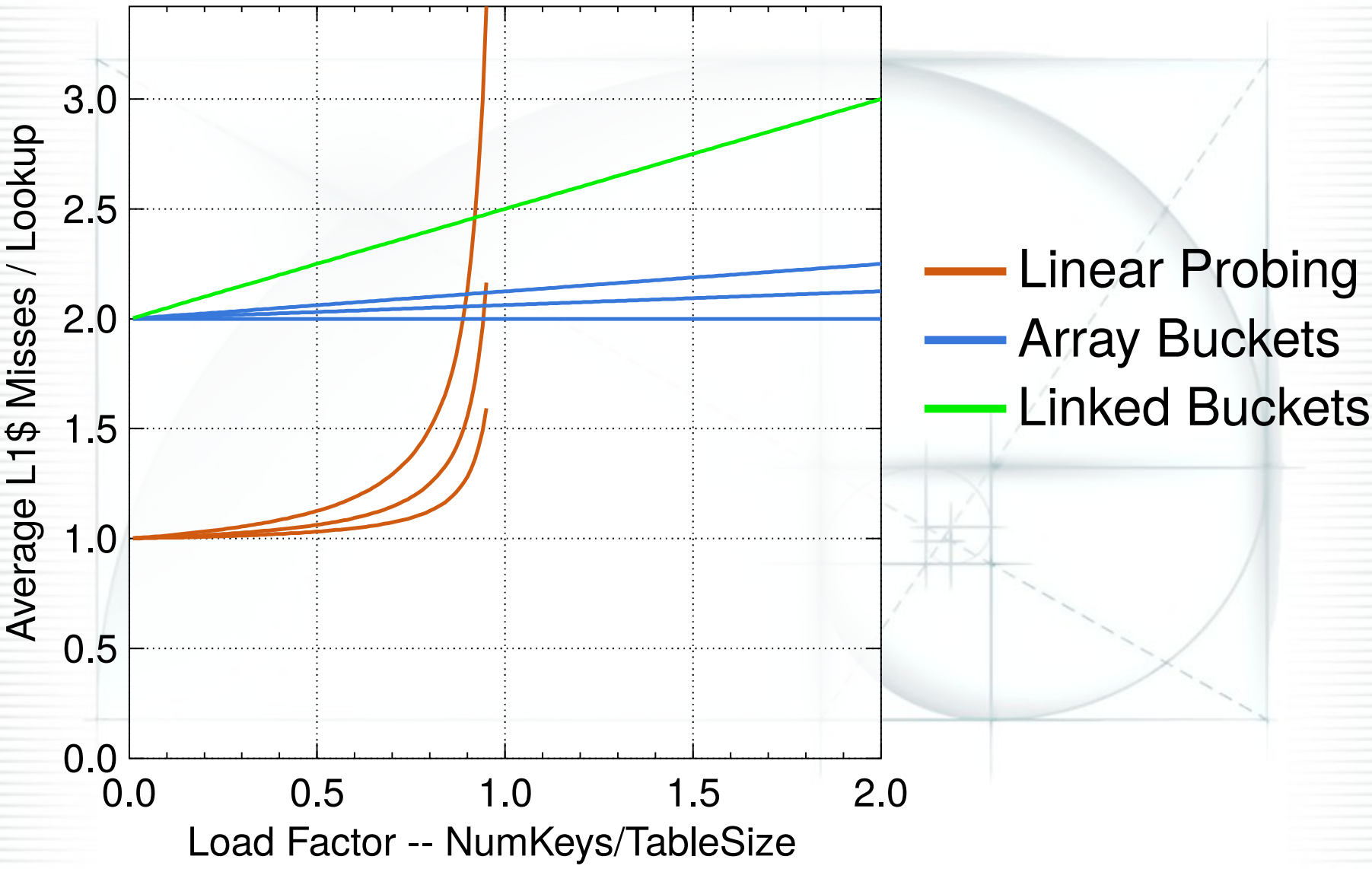

Predicted Cache Misses
Average L1$ Misses / Lookup
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.5
1.0
1.5
2.0
Load Factor -- NumKeys/TableSize
Linked Buckets

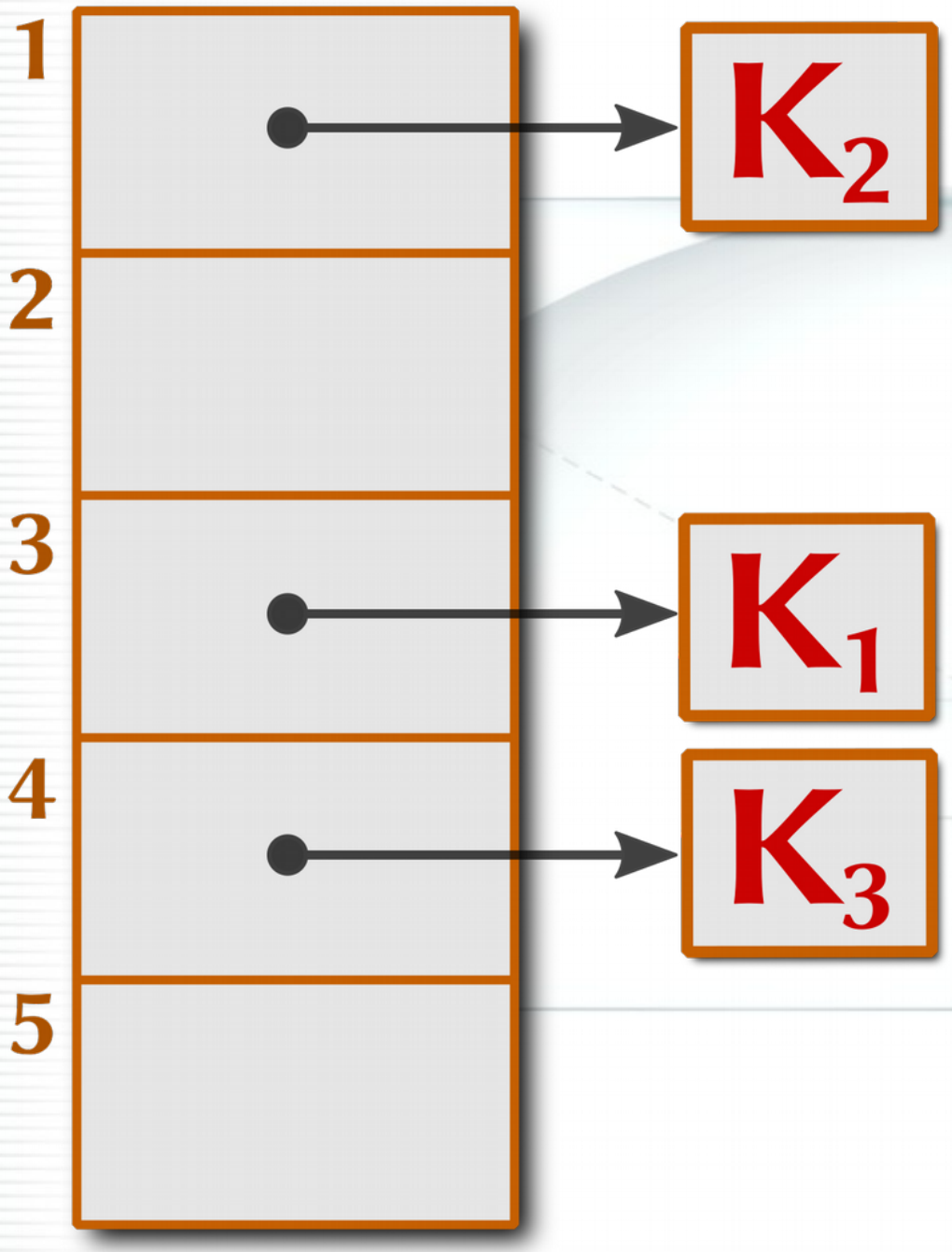

Predicted Cache Misses
Average L1$ Misses / Lookup
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.5
1.0
1.5
2.0
Load Factor -- NumKeys/TableSize
Array Buckets
Linked Buckets


1
2
3
4
5
K2
K1
K3

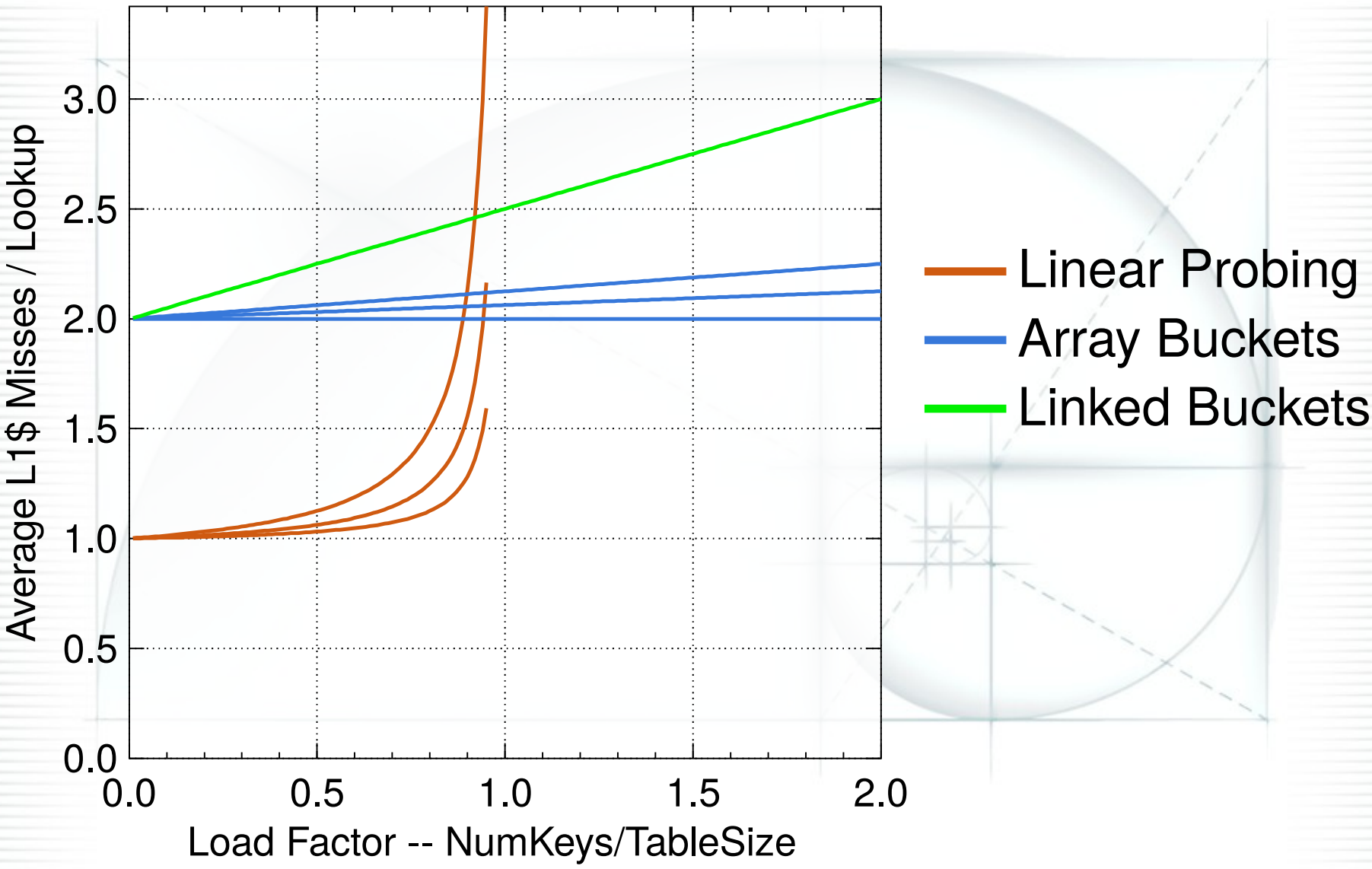

Predicted Cache Misses
Average L1$ Misses / Lookup
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.5
1.0
1.5
2.0
Load Factor -- NumKeys/TableSize
Array Buckets
Linked Buckets

Predicted Cache Misses

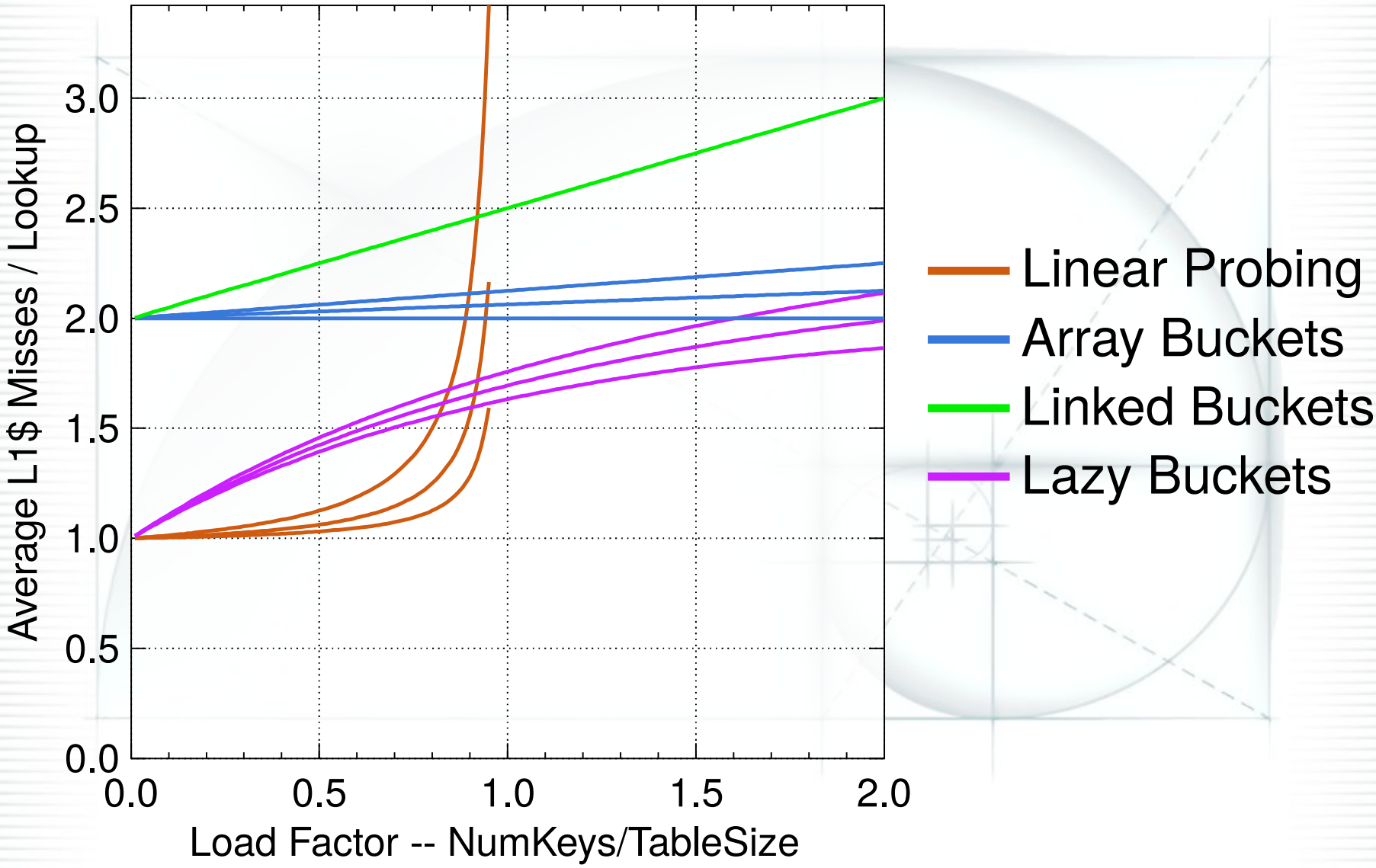


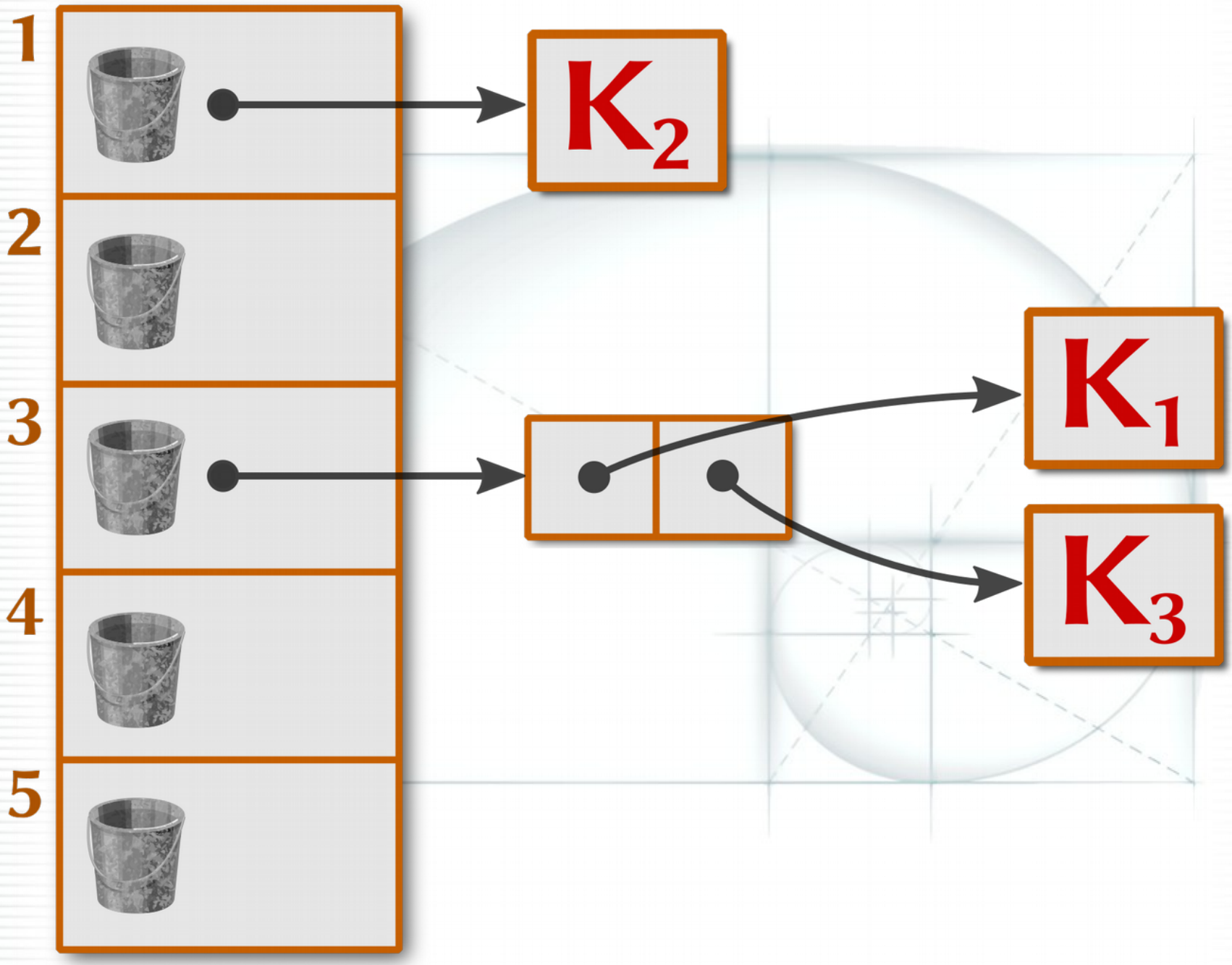

1
2
3
4
5
K2
K1
K3

Predicted Cache Misses

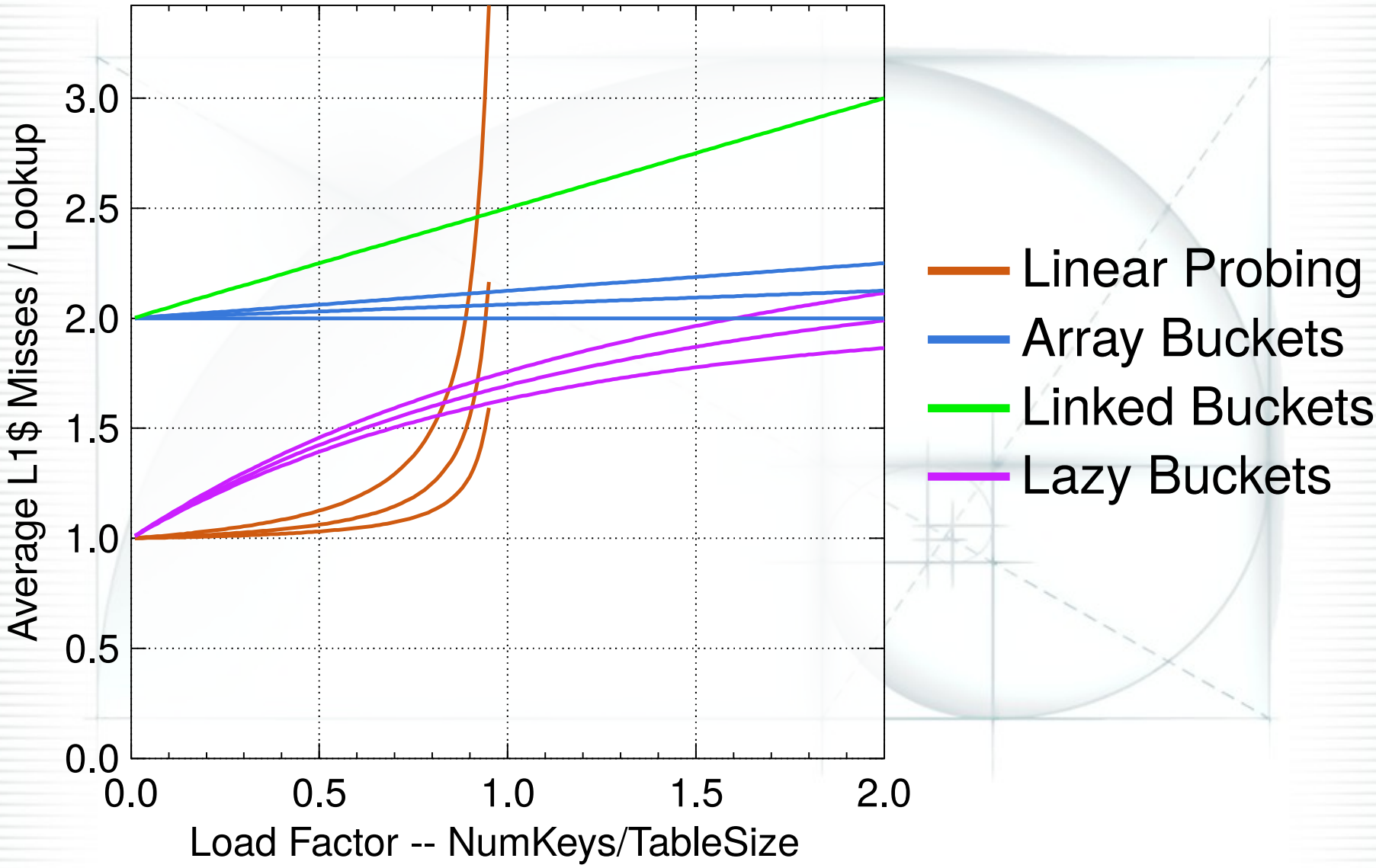


Predicted Cache Misses

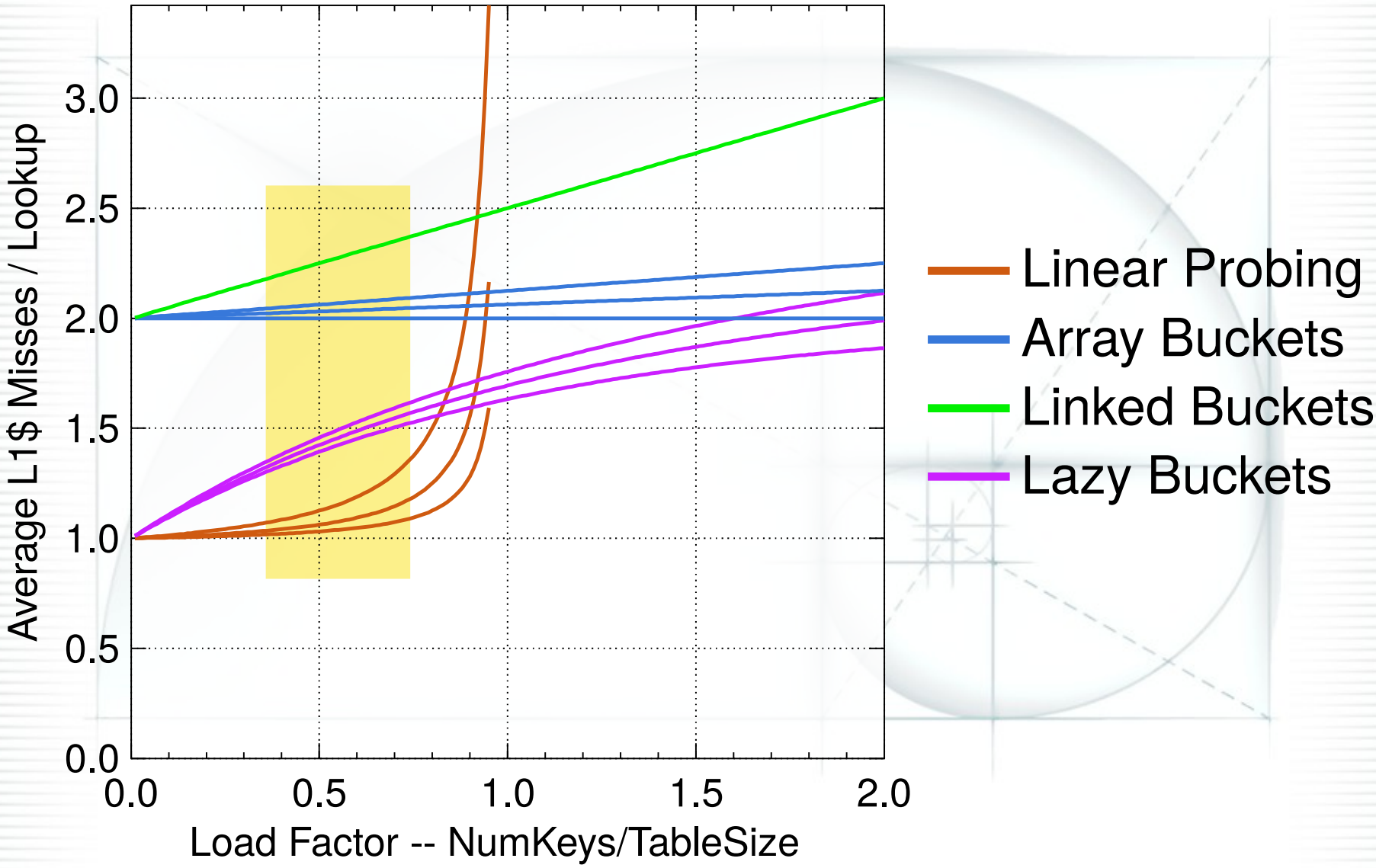



1
2
3
4
5
K2
K1
K3

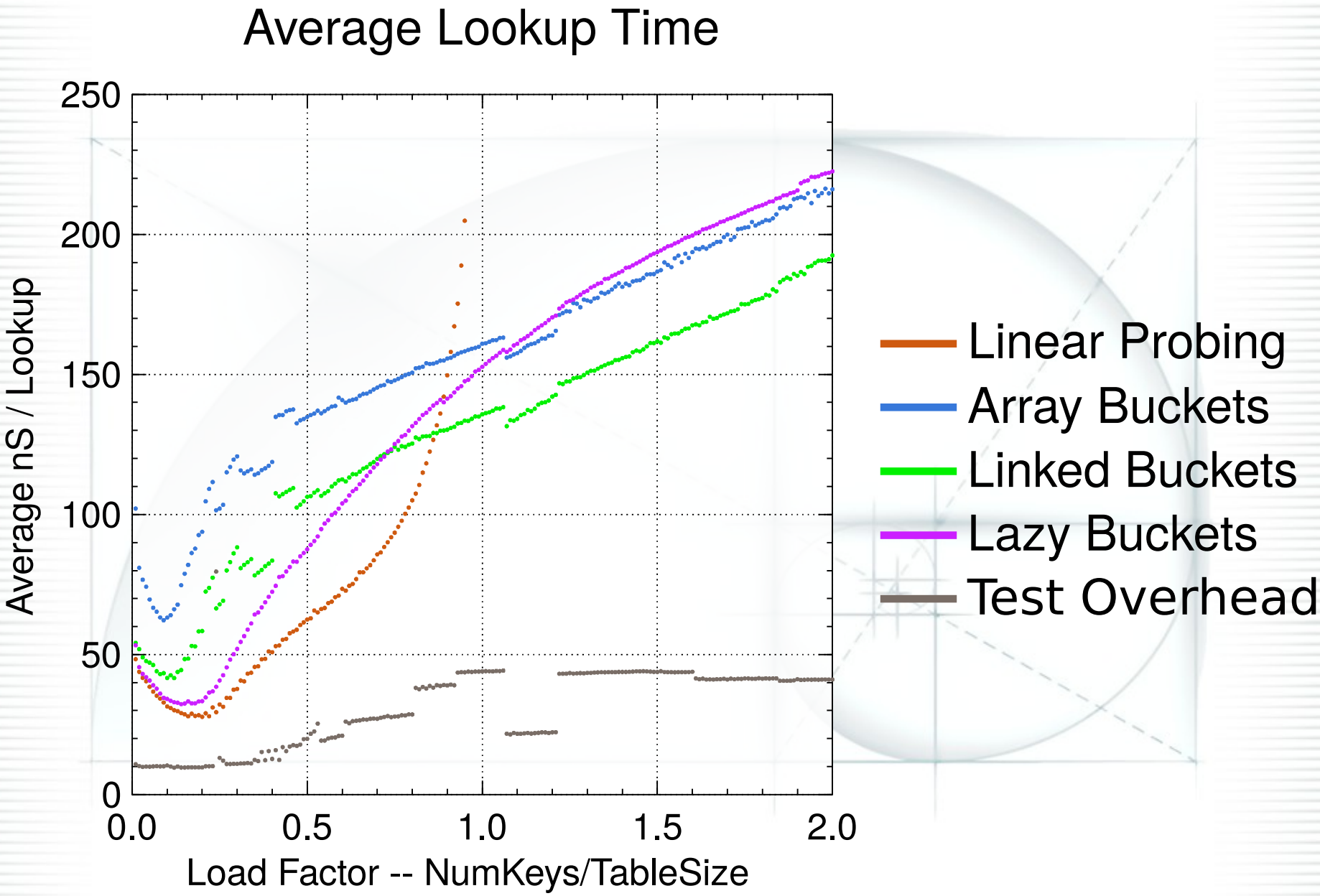
Predicted Cache Misses

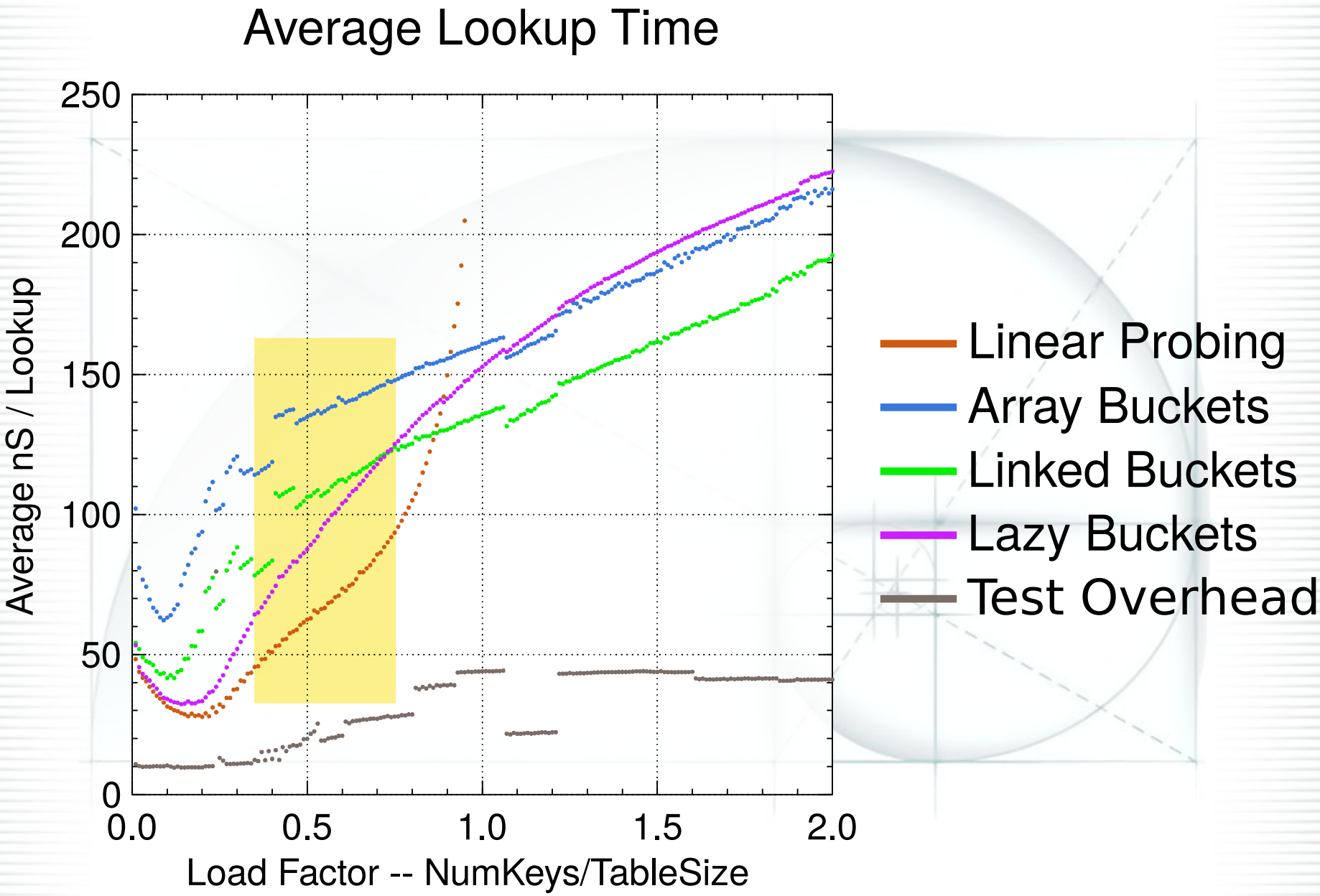


Predicted Cache Misses



Test the Hypothesis

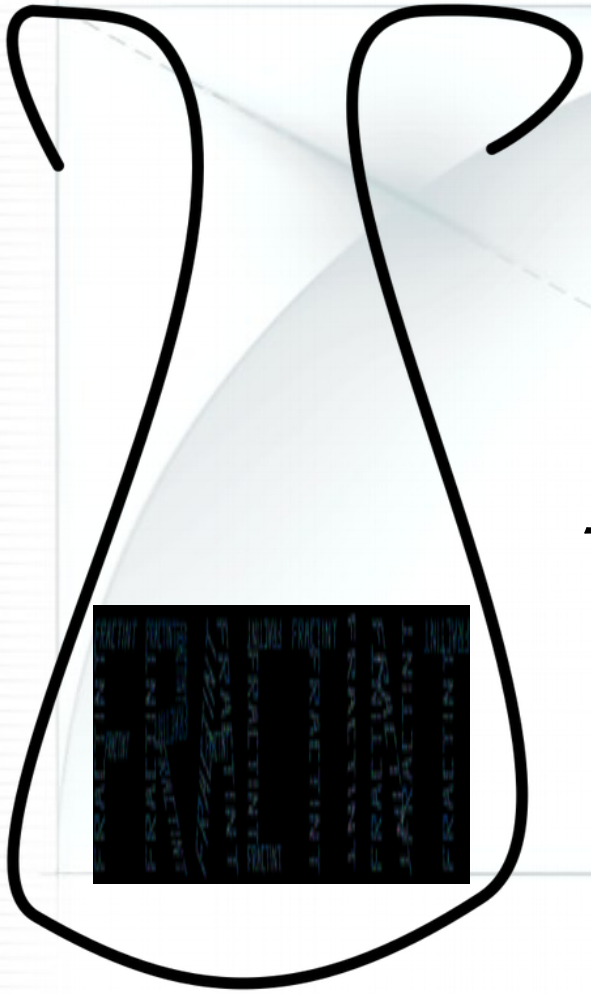

Average Lookup Time
Average nS / Lookup
250
200
150
100
50
0
0.0
0.5
1.0
1.5
2.0
Load Factor -- NumKeys/TableSize
Linear Probing
Array Buckets
Linked Buckets
Lazy Buckets
Test Overhead


Average Lookup Time
Average nS / Lookup
250
200
150
100
50
0
0.0
0.5
1.0
1.5
2.0
Load Factor -- NumKeys/TableSize
Linear Probing
Array Buckets
Linked Buckets
Lazy Buckets
Test Overhead

Security

D

f(D)



“hash function”

maybe random mod s

Data Set

#hash Values

D

f(D)

“hash function”

fixed mod s

Data Set

#hash Values

Why Test?

JDK 1.5

```
/**
 * Returns a hash value for the specified object.  In addition to
 * the object's own hashCode, this method applies a "supplemental
 * hash function," which defends against poor quality hash functions.
 * This is critical because HashMap uses power-of two length
 * hash tables.<p>
 *
 * The shift distances in this function were chosen as the result
 * of an automated search over the entire four-dimensional search space.
 */
static int hash(Object x) {
    int h = x.hashCode();

    h += ~(h << 9);
    h ^=  (h >>> 14);
    h +=  (h << 4);
    h ^=  (h >>> 10);
    return h;
}
```

Multiplicative hash function

```
String>>hash

        | answer |
        answer := self size.
        self do:
                [:each |
                        answer := answer * magic bitAnd: 2^k.
                        answer := answer + each asInteger
                ].
        ^answer
```

Multiplicative hash function

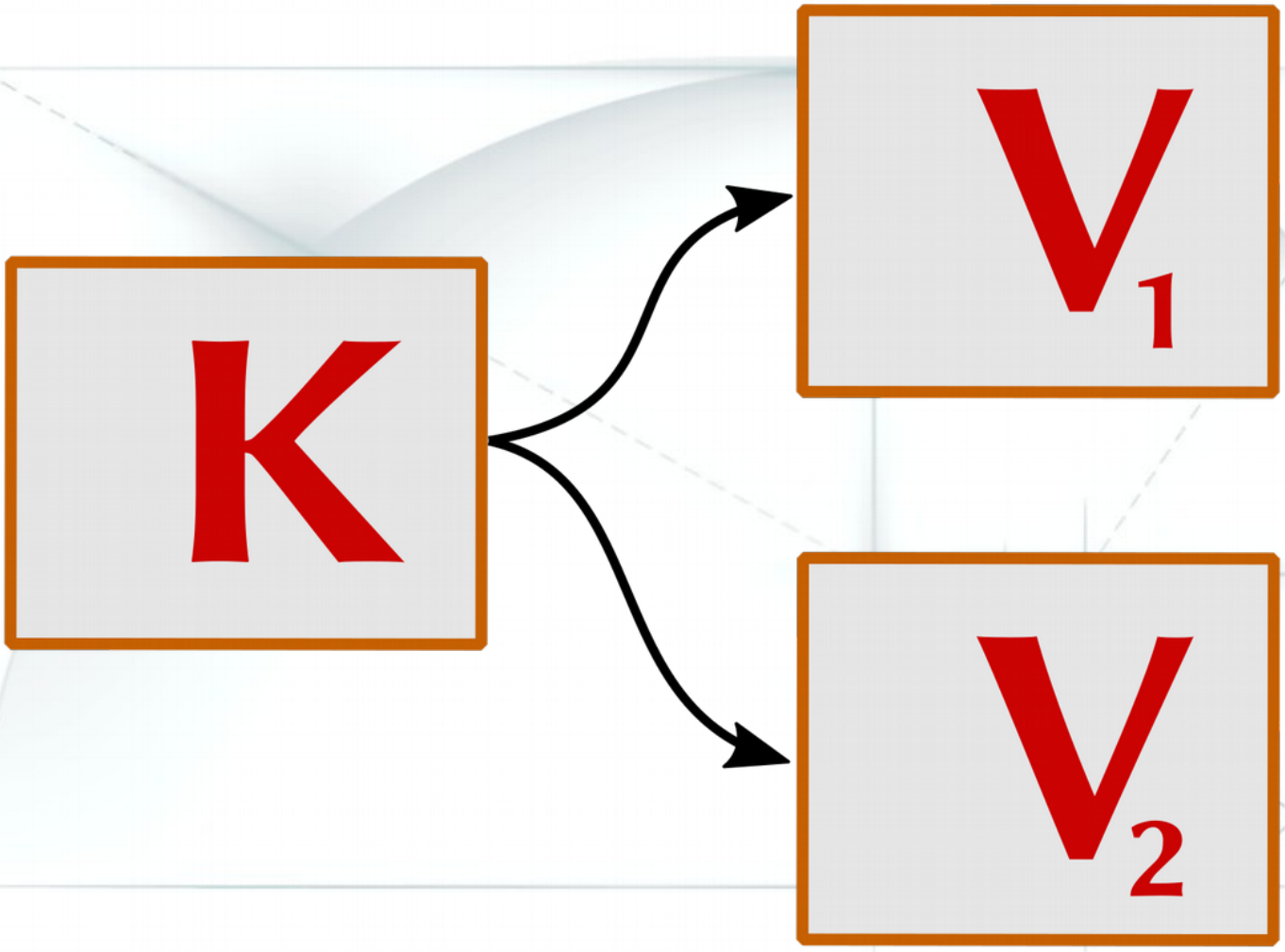
Magic	Hash function
31, 131, 1313, 13131, ...	K&R's digital method, Java
5	C++ STL
1664525	#hashMultiply
19	OCaml
2, 8	Bjarne Stroustrup
13, 53	C# 2005 Professional
32.25	Boost, Justin Sobel
33	Bernstein, Standard ML

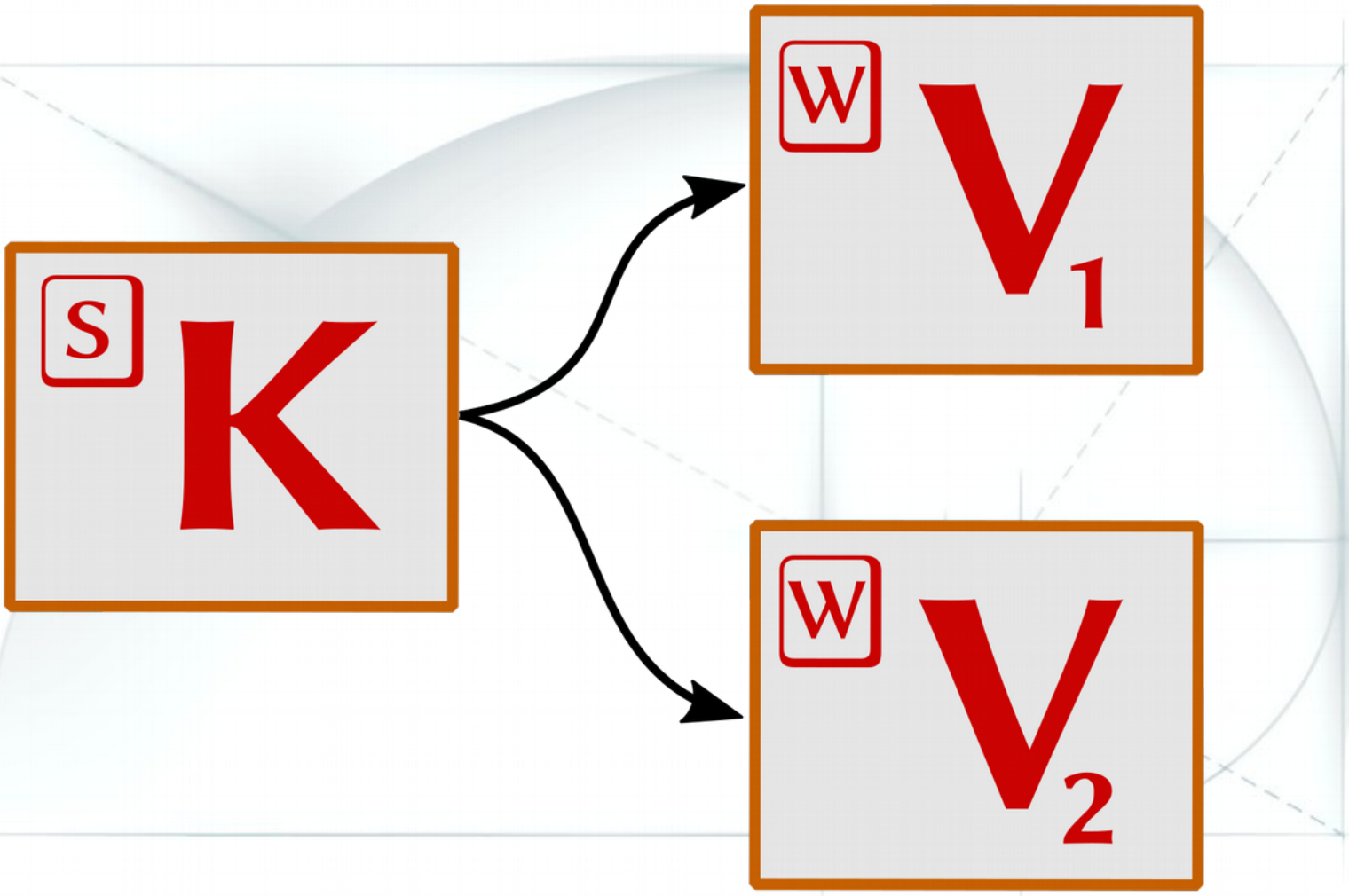
Funny hash methods

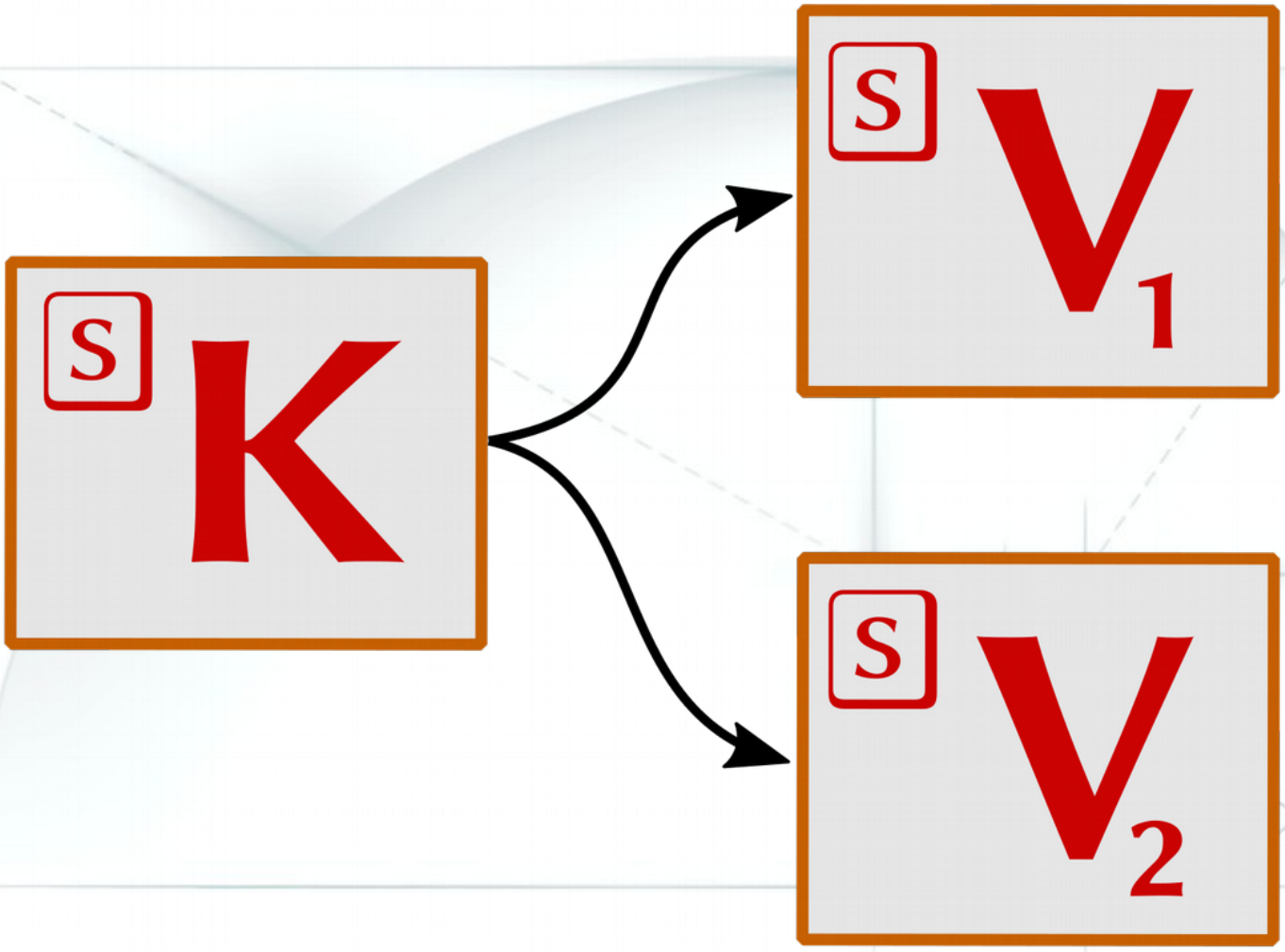
```
SomeClass>>hash

        ^12345
                bitXor: self a hash;
                bitXor: self b hash;
                bitXor: self c hash;
                yourself
```

Real-World Examples

GBS ServerMap

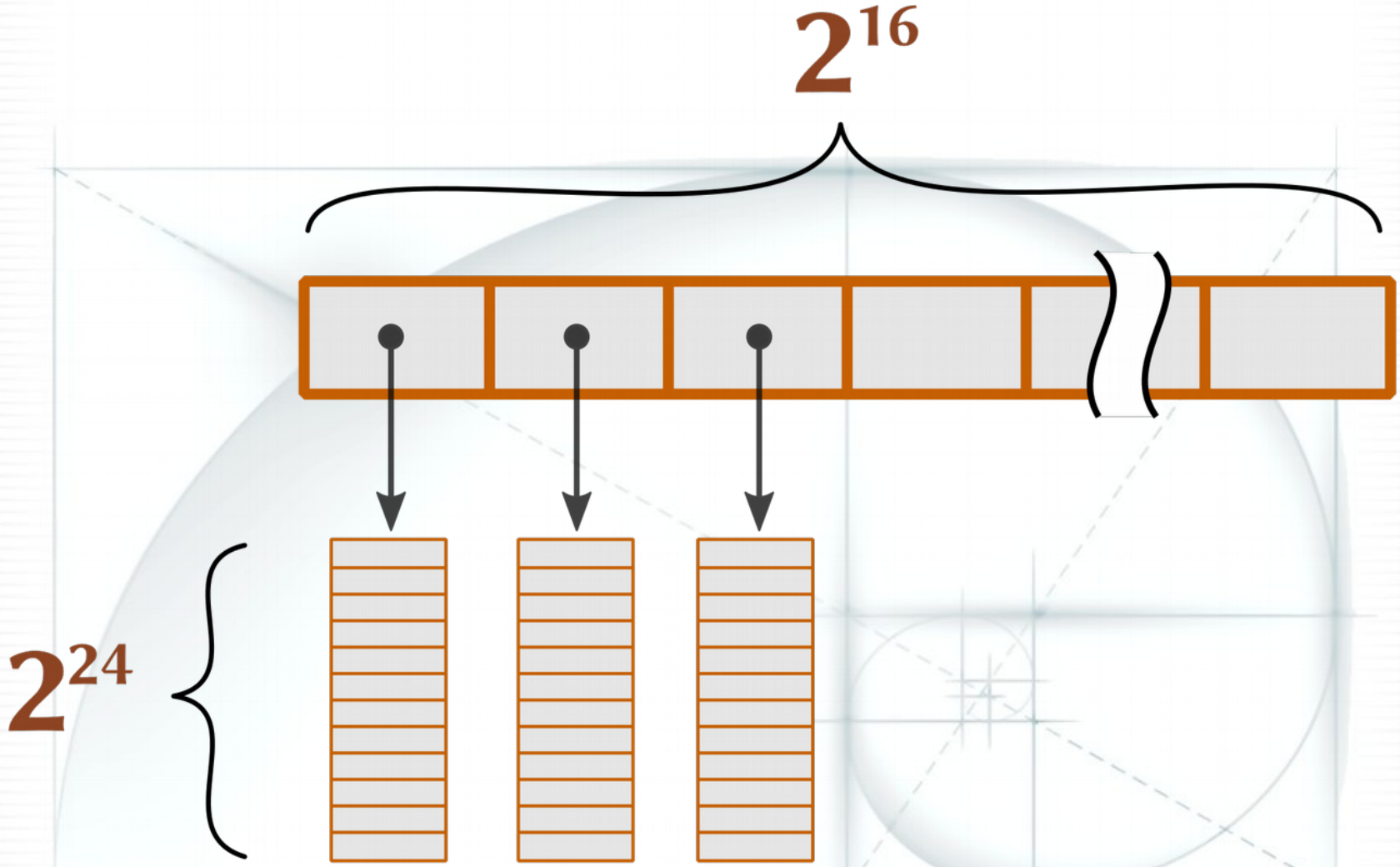

K
V1
V2


S
K
W
V1
W
V2


S
K
S
V1
S
V2

Also

- Integer keys, 1-2^{40} (typically smallish)
- ~2M keys
- ~100K updates/sec
- ~(1-10)×100K lookups/sec

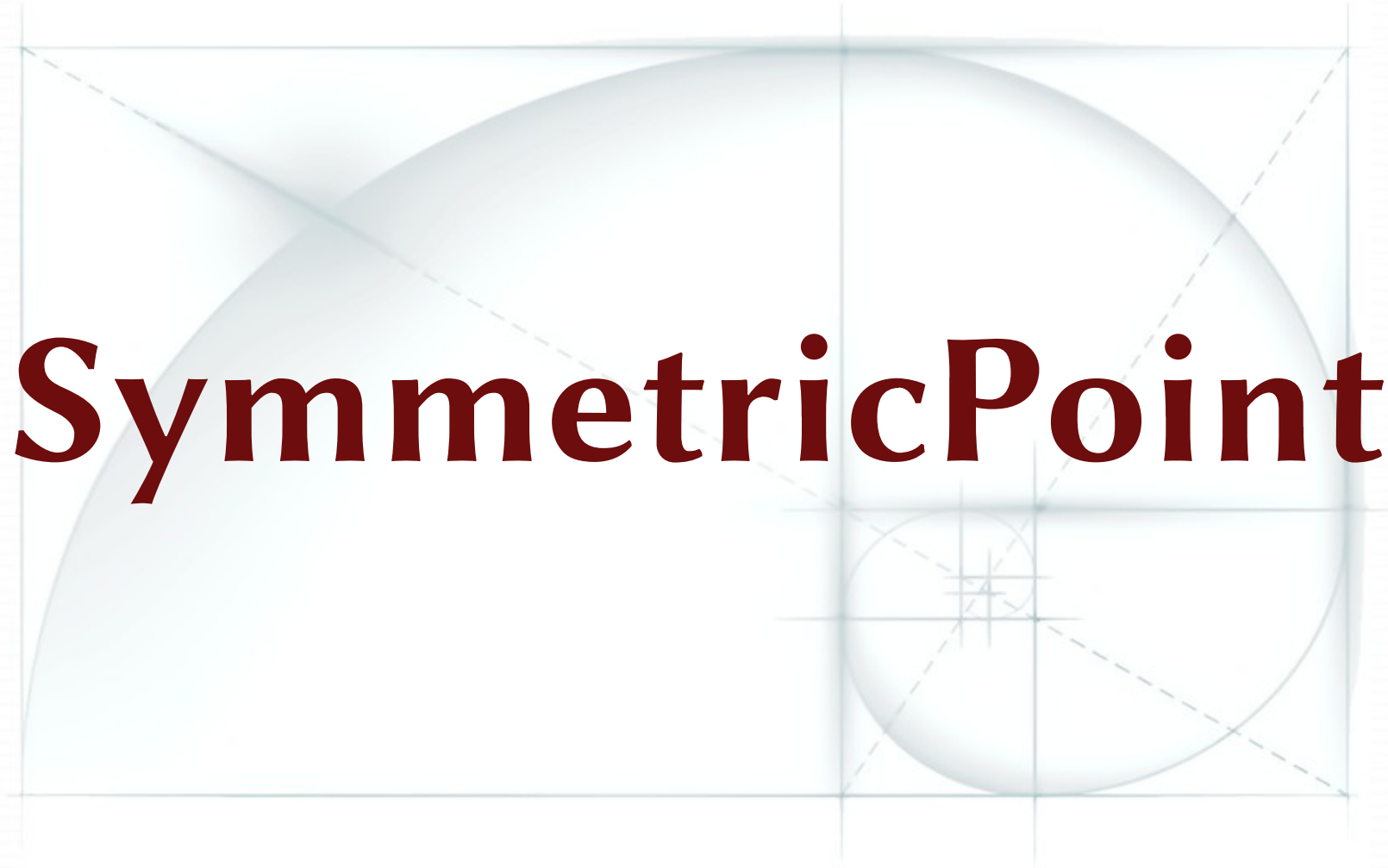

2^{16}
2^{24}


2^{24}
V_2
K
V_1
V_2
K


2^{24}
W
V_2
K
V_1
V_2
K


2^{24}
S
V_2
K
V_1
V_2
K

#hashMultiply


SymmetricPoint

Symmetric points

```
SymmetricPoint>>= aPoint

        (self x = aPoint x
                and: [self y = aPoint y]) ifTrue: [^true].
        (self y = aPoint x
                and: [self x = aPoint y]) ifTrue: [^true].
        ^false
```

Symmetric points

#hash

1x

```
SymmetricPoint>>hash

	^self x + self y
```

5x

```
SymmetricPoint>>hash

	| xSquared ySquared product |
	xSquared := self x * self x.
	ySquared := self y * self y.
	product := (xSquared + 1) * (ySquared + 1).
	^self x * self y + product
```

Symmetric points

#hash

Use case

SymmetricPoint>>hash

1x

```
^self x + self y
```

20x

SymmetricPoint>>hash

5x

```
| xSquared ySquared product |
xSquared := self x * self x.
ySquared := self y * self y.
product := (xSquared + 1) * (ySquared + 1).
^self x * self y + product
```

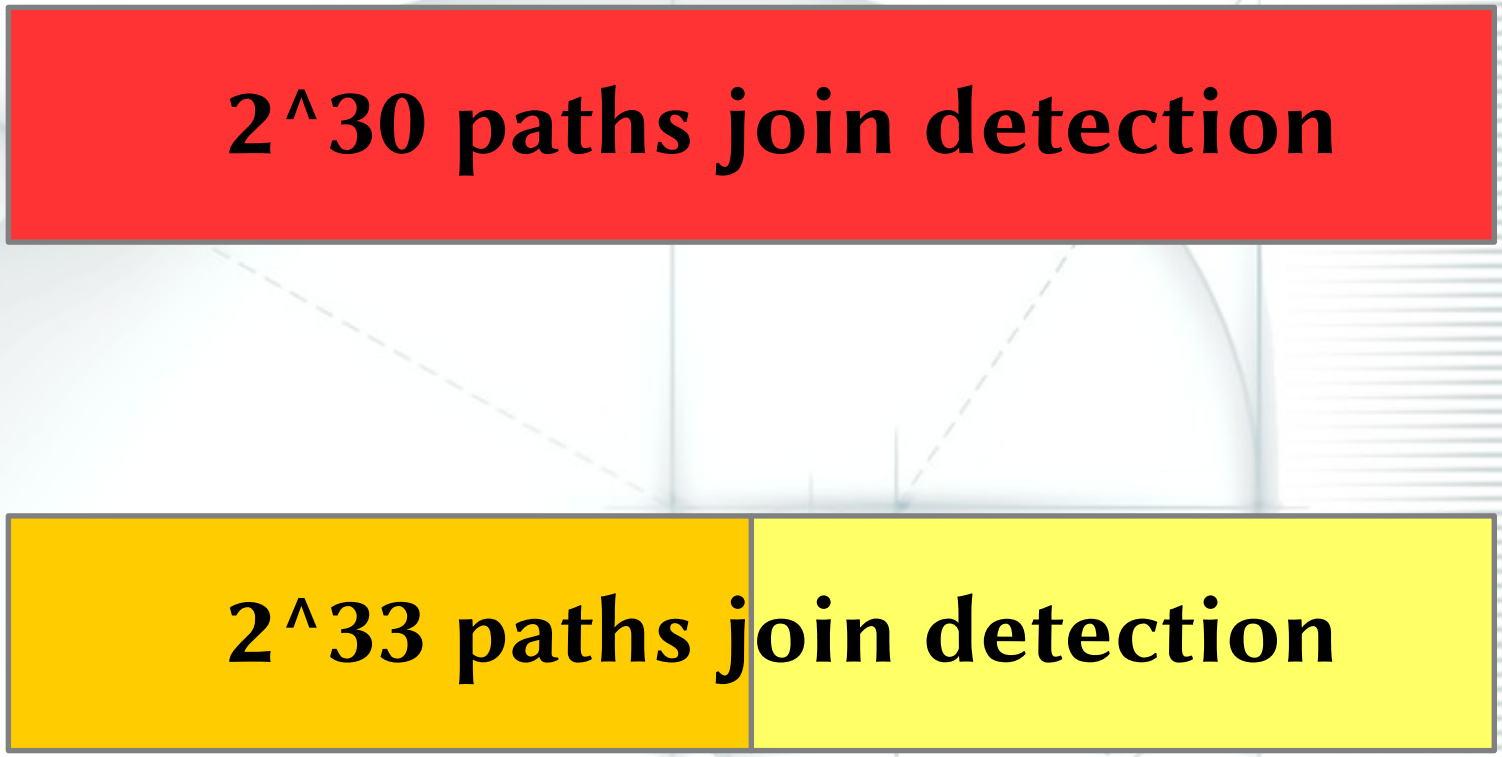
1x

Number crunching

1.5gb

2^30 paths join detection

Number crunching

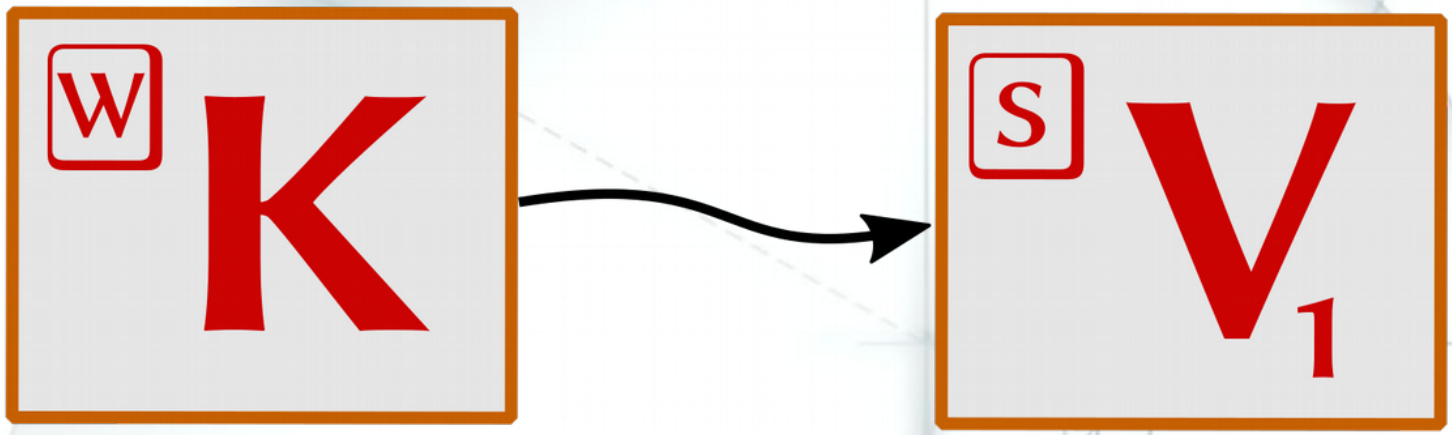


Cheney scavenging

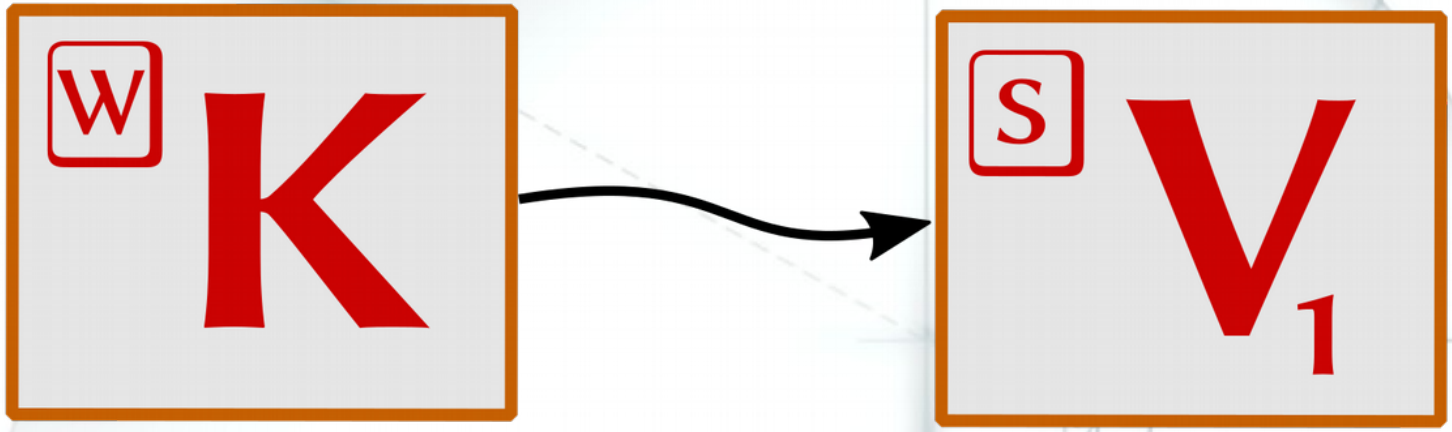
GBS ClientMap

ClientMap

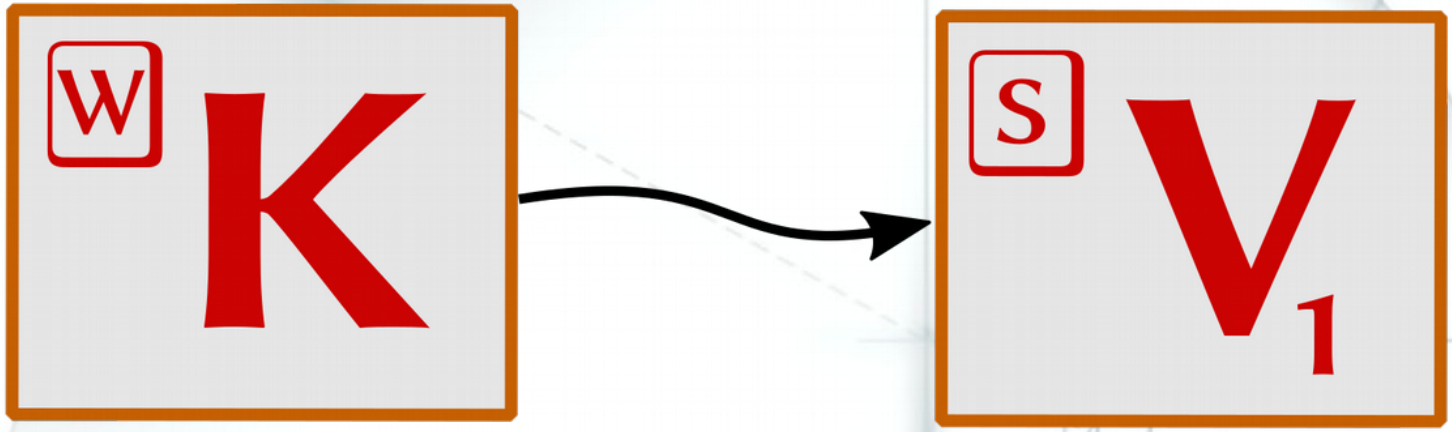
- Identity Dictionary
- ~2M keys
- ~100K updates/sec
- ~(1-10)×100K lookups/sec


W
K
S
V
1

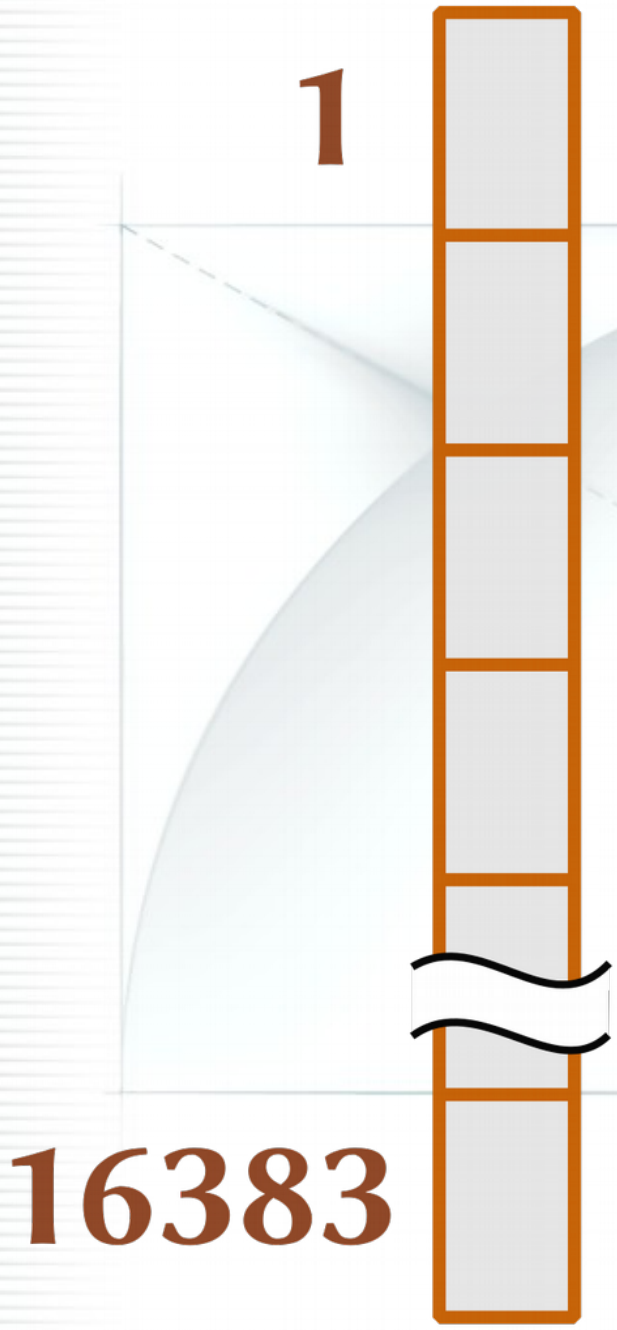
32-bit VW

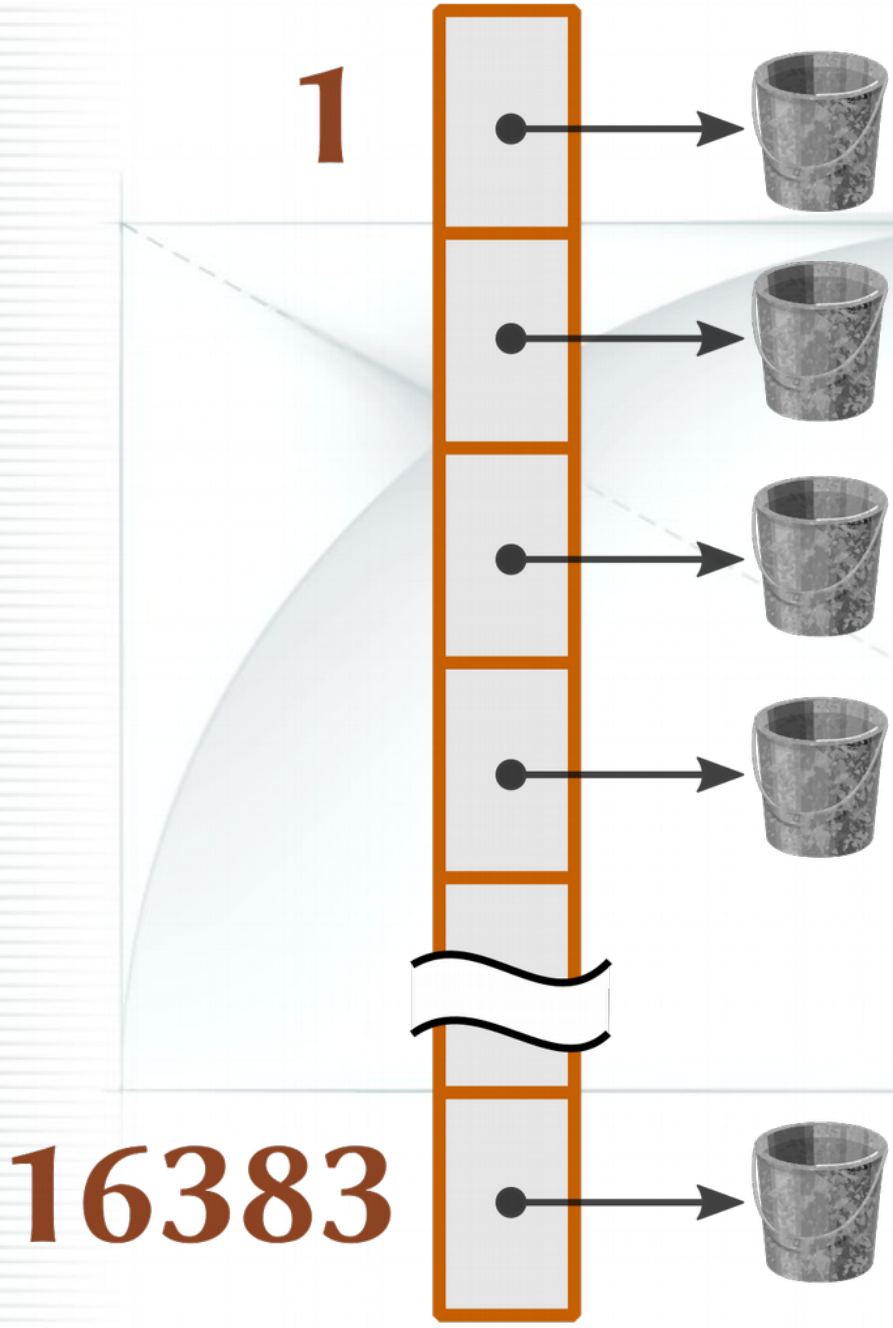


32-bit VW



2^{14} hash values


1
16383


1
16383


V
V
V


W


K
K
K

Ways to Fail
(a very incomplete list)

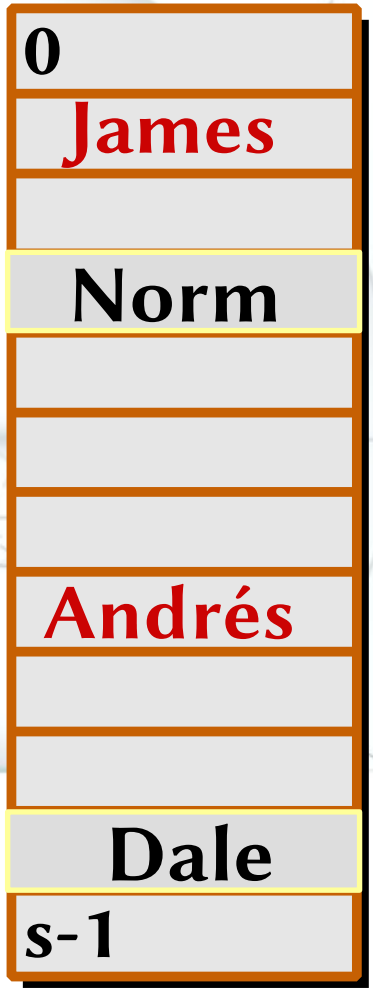
Effects of become:

Effects of become:

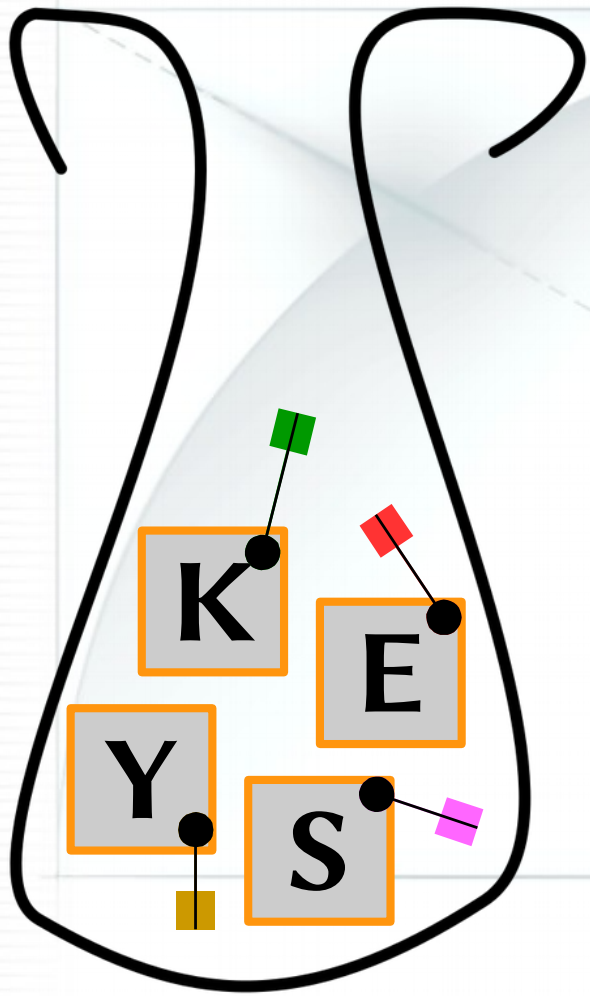
'Norm' oneWayBecome: 'Dale'

Effects of become:

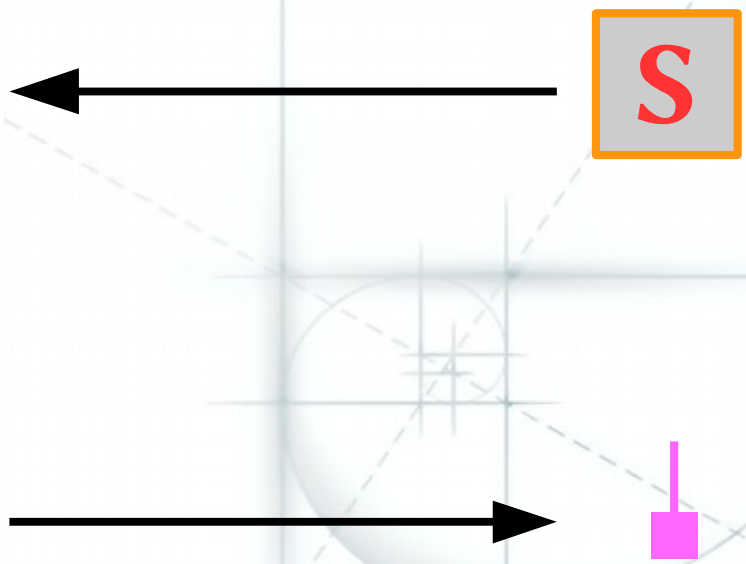
'Norm' become: 'Dale'



Key mutation

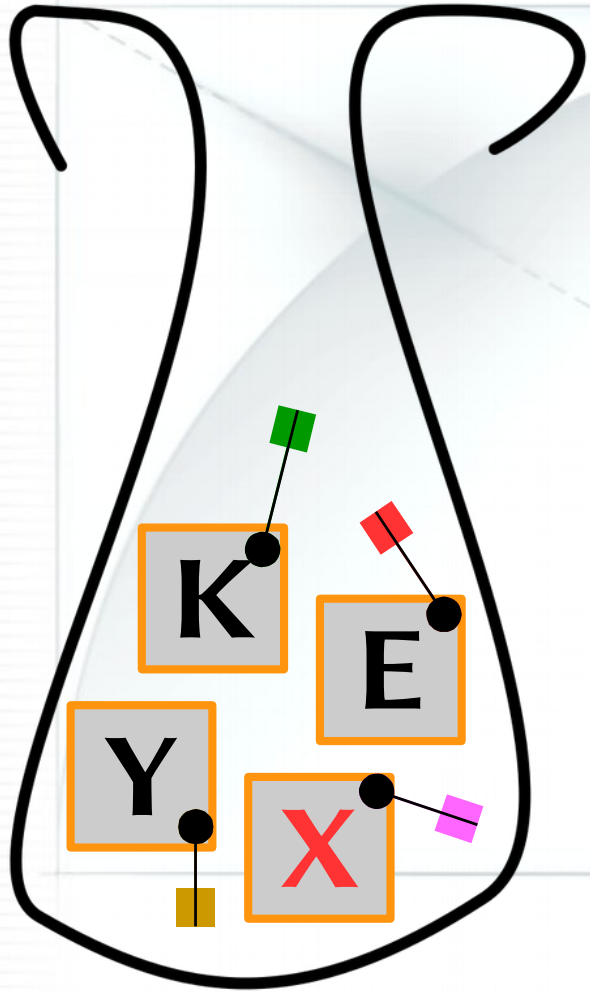


#at: behavior



Dictionary

Key mutation



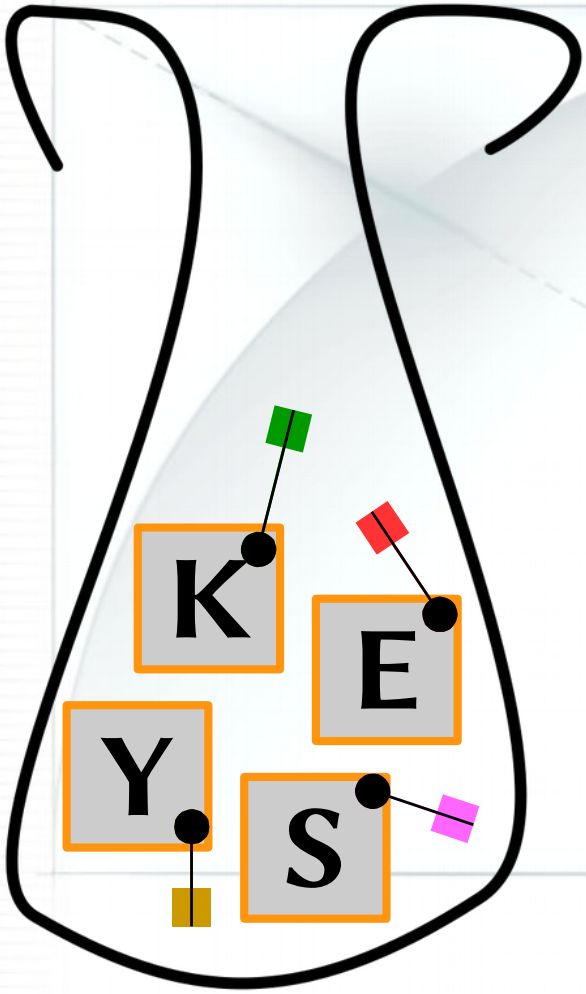
Dictionary

#at: behavior

S

absent

Threading



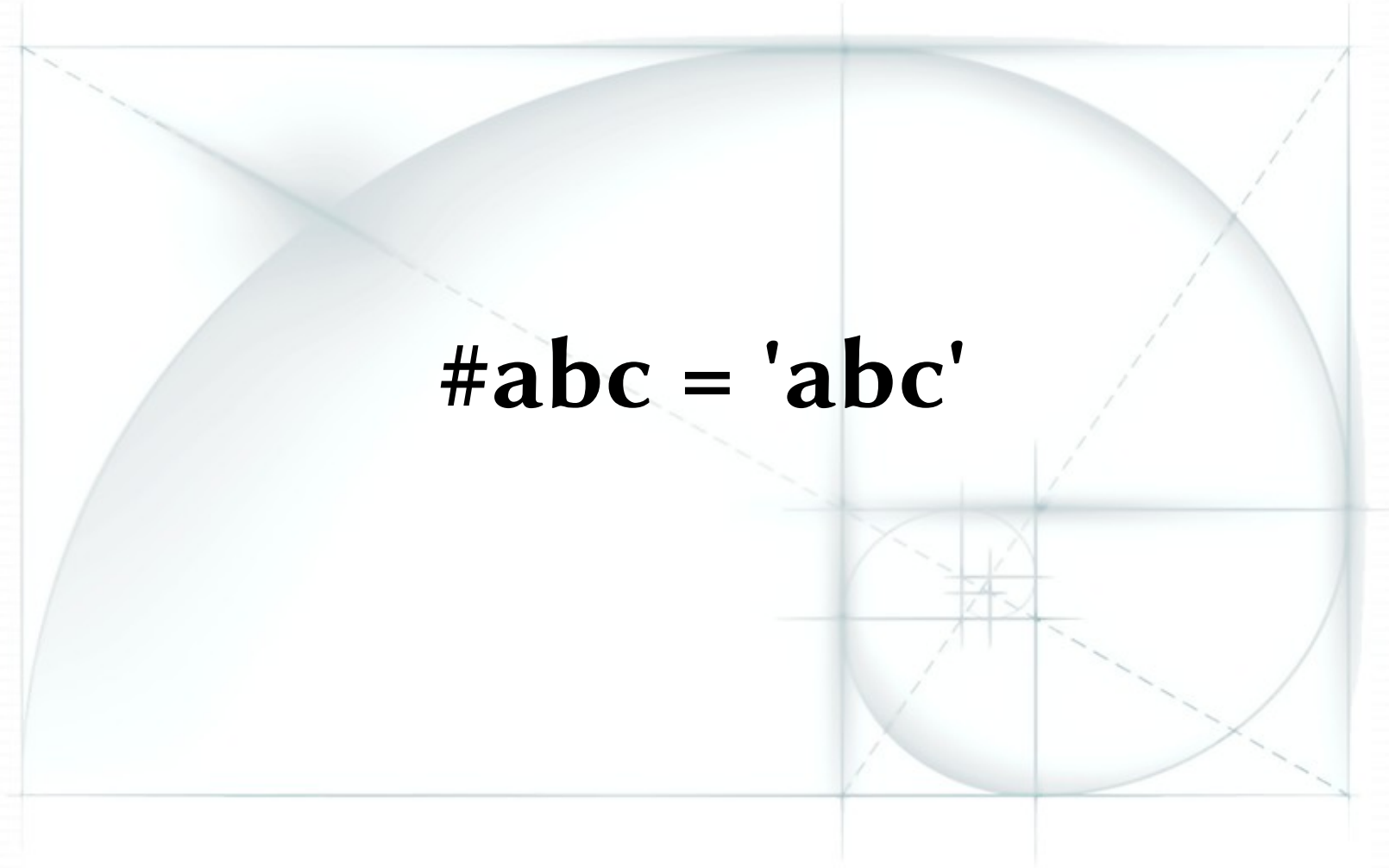
#at:put: ...

<process switch>

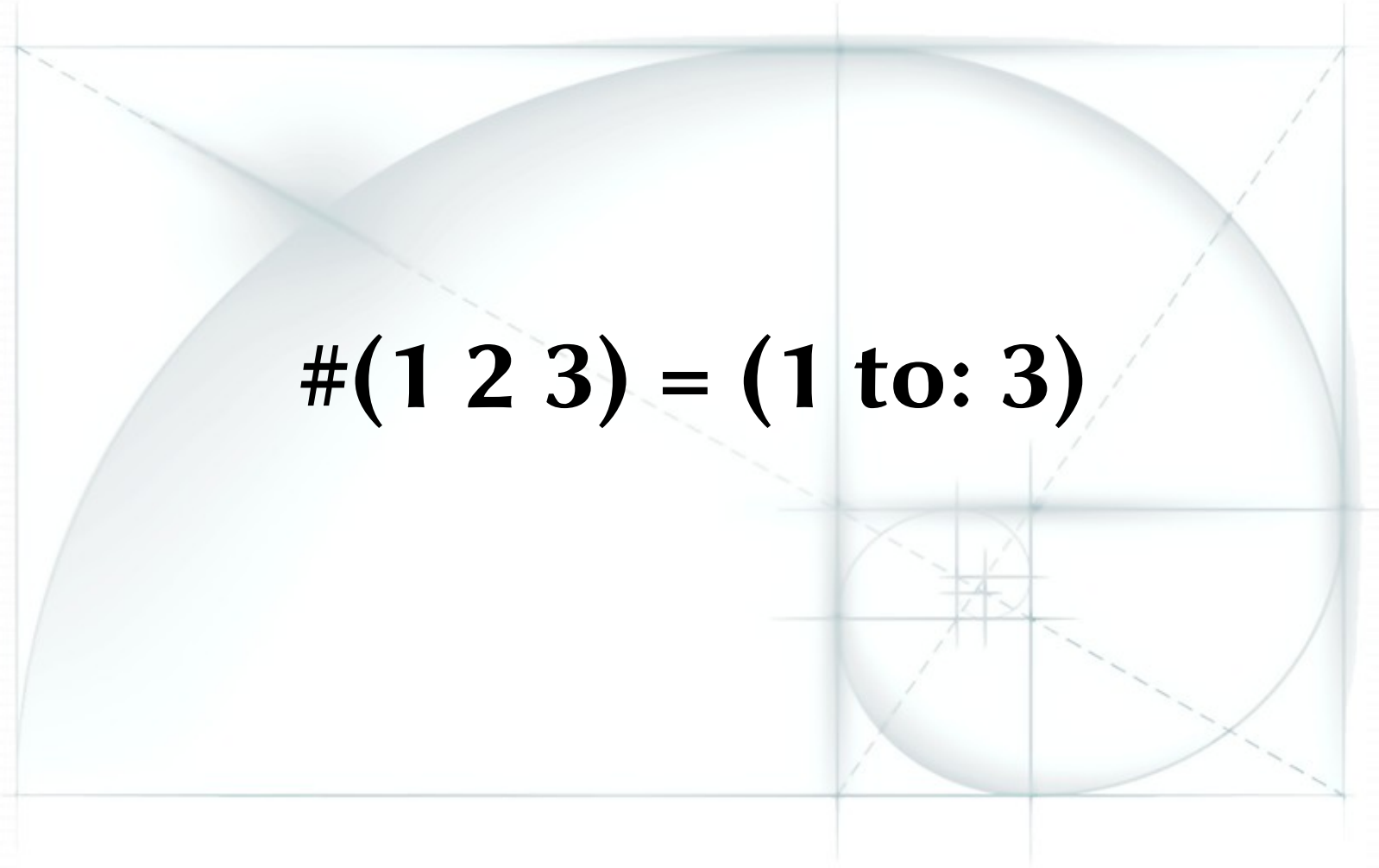
#at:put: ...

Dictionary

Funny equal methods

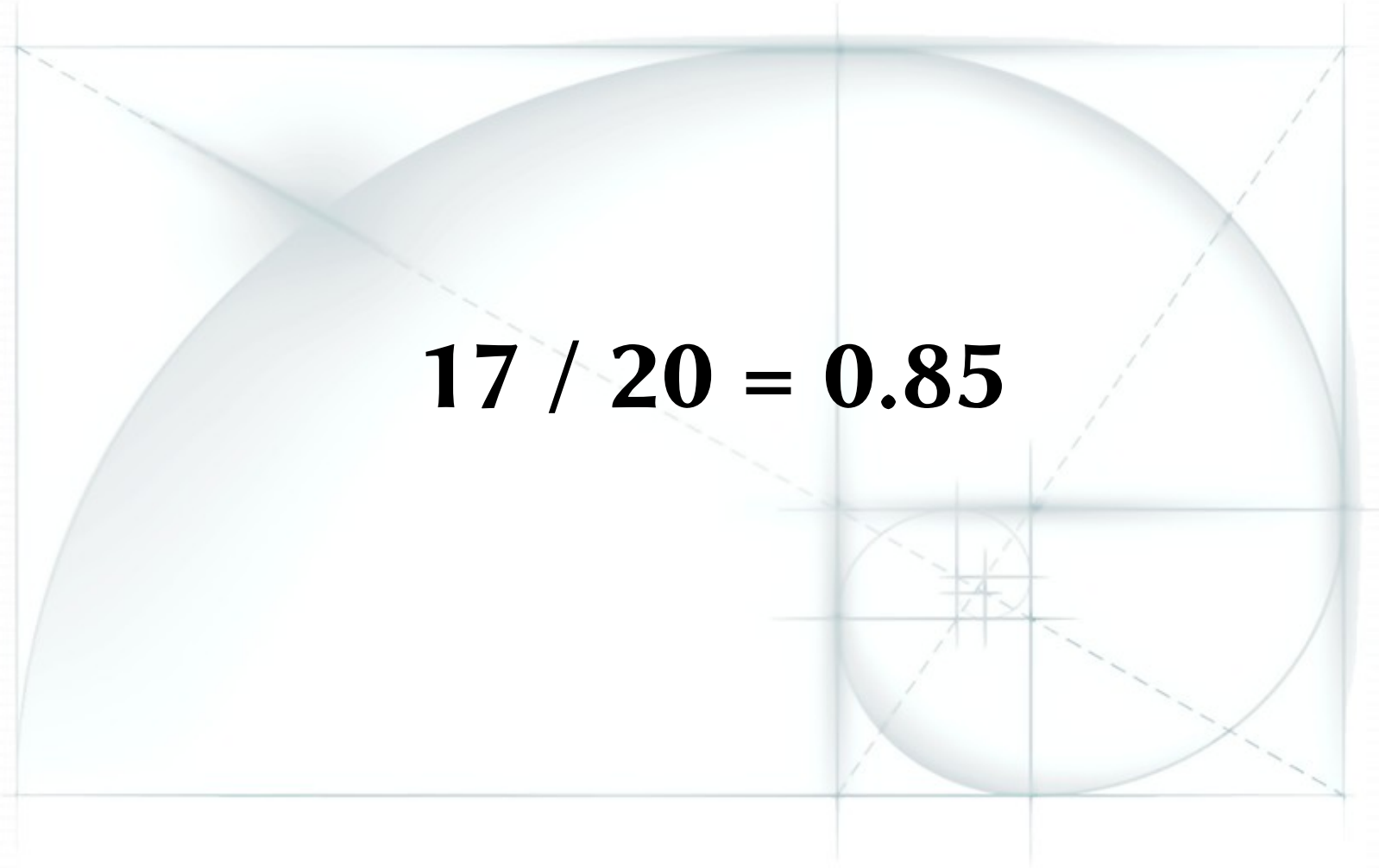


Funny equal methods

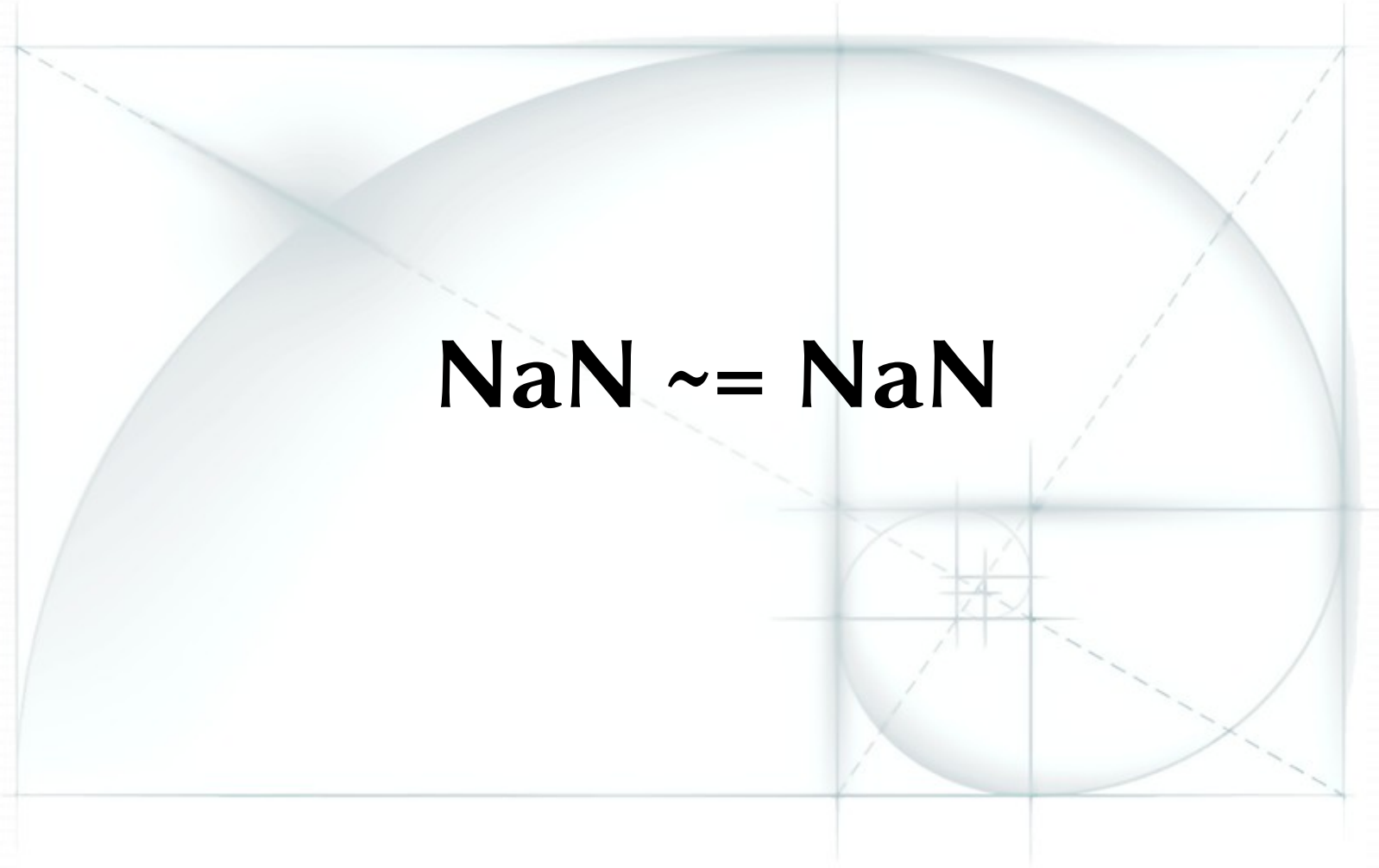


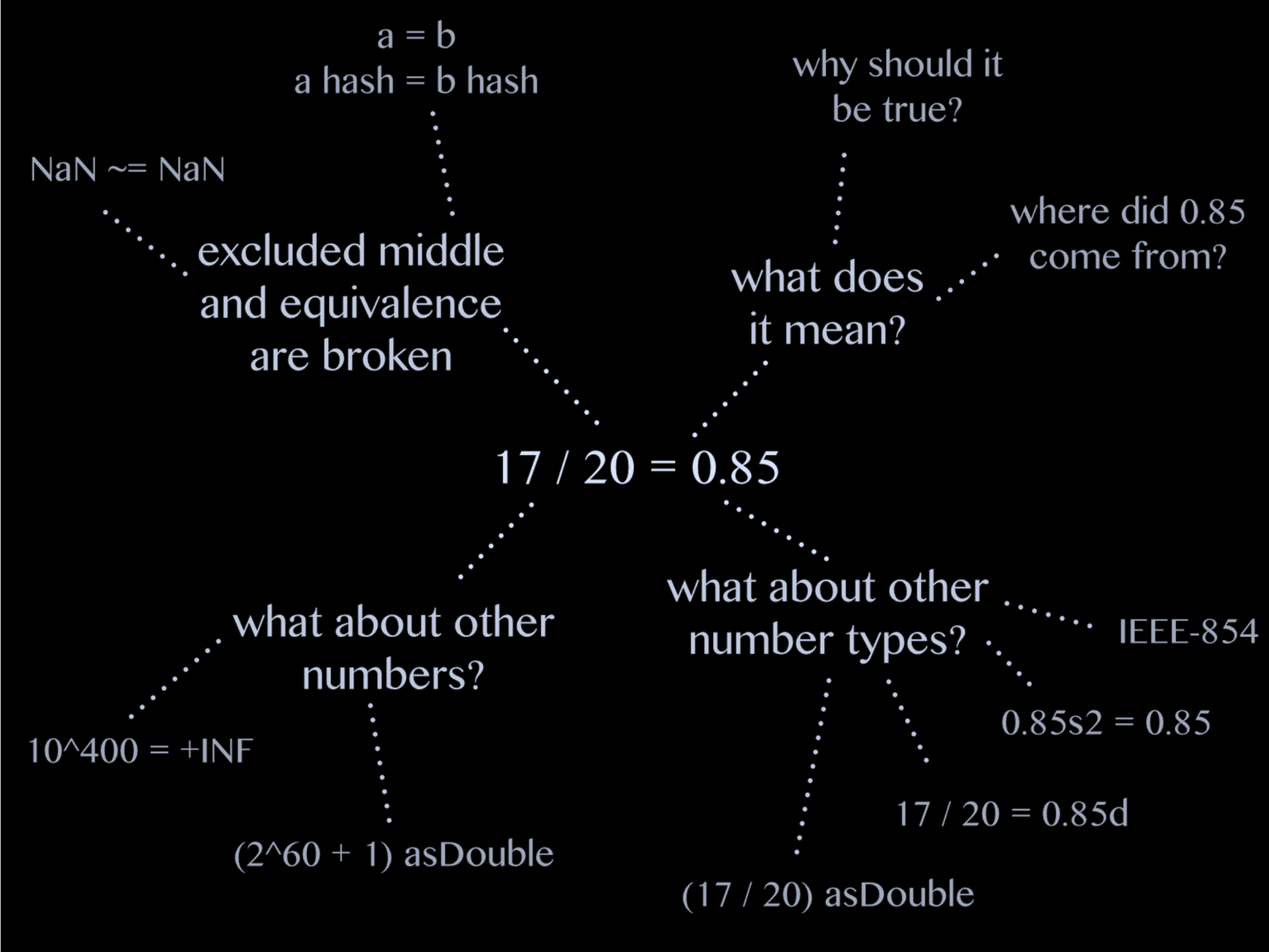
```
#(1 2 3) = (1 to: 3)
```

Funny equal methods

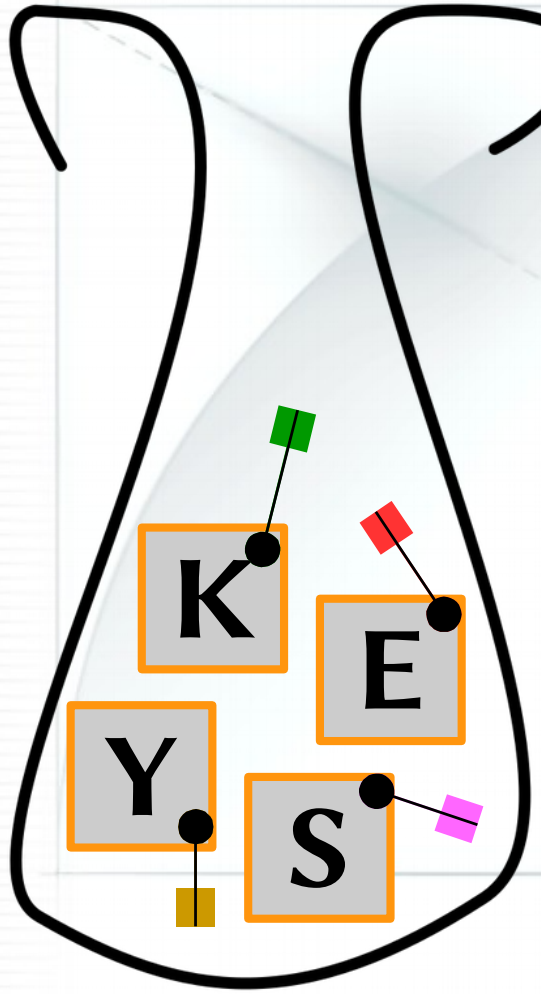


Funny equal methods




a = b
a hash = b hash
NaN ~= NaN
excluded middle
and equivalence
are broken
why should it
be true?
where did 0.85
come from?
what does
it mean?
17 / 20 = 0.85
what about other
numbers?
10^400 = +INF
(2^60 + 1) asDouble
what about other
number types?
IEEE-854
0.85s2 = 0.85
17 / 20 = 0.85d
(17 / 20) asDouble

Reverse hashing



keyAtValue:

Dictionary

Final Thoughts

Hash Function: Quality over speed

Hash Function:
Quality ≠ complexity

Hash Method
is only half of
Hash Function

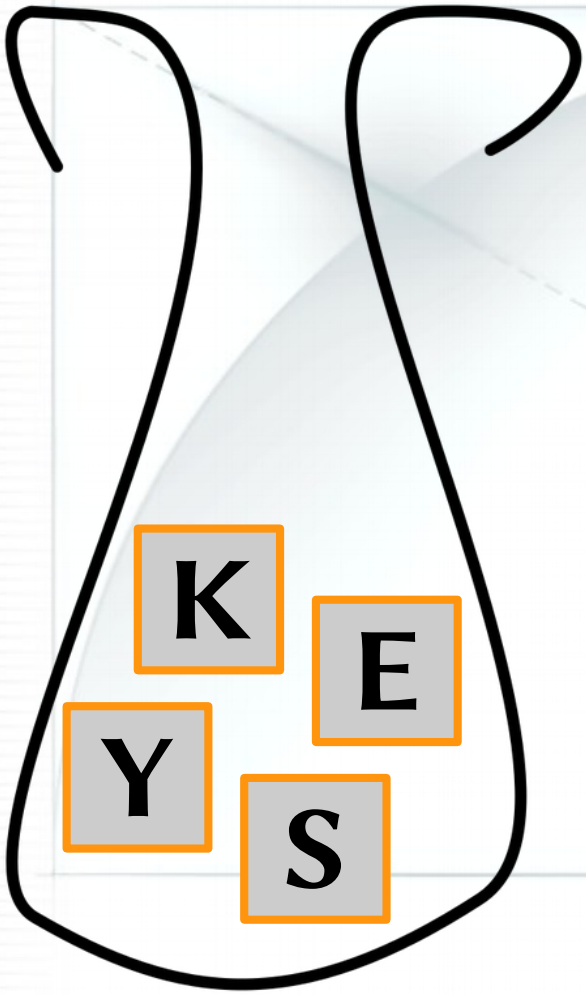
Test Profile TEST

Hashed Collections

...it julienne fries!

Martin McClure

Andrés Valloud



Set

#like: behavior

S

S

JDK 1.6

```
/**
 * Applies a supplemental hash function to a given hashCode, which
 * defends against poor quality hash functions.  This is critical
 * because HashMap uses power-of-two length hash tables, that
 * otherwise encounter collisions for hashCodes that do not differ
 * in lower bits. Note: Null keys always map to hash 0, thus index 0.
 */
static int hash(int h) {
    // This function ensures that hashCodes that differ only by
    // constant multiples at each bit position have a bounded
    // number of collisions (approximately 8 at default load factor).
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
```

Multiplicative hash function

Magic	Hash function
16,777,619 1, 099, 511, 628, 211 309, 485, 009, 821, 345, 068, 724, 781, 401 374, 144, 419, 156, 711, 147, 060, 143, 317, 175, 368, 453, 031, 918, 731, 002, 211 35, 835, 915, 874, 844, 867, 368, 919, 076, 489, 095, 108, 449, 946, 327, 955, 754, 392 558, 399, 825, 615, 420, 669, 938, 882, 575, 126, 094, 039, 892, 345, 713, 852, 759 5, 016, 456, 510, 113, 118, 655, 434, 598, 811, 035, 278, 955, 030, 765, 345, 404, 790 744, 303, 017, 523, 831, 112, 055, 108, 147, 451, 509, 157, 692, 220, 295, 382, 716 162, 651, 878, 526, 895, 249, 385, 292, 291, 816, 524, 375, 083, 746, 691, 371, 804 094, 271, 873, 160, 484, 737, 966, 720, 260, 389, 217, 684, 476, 157, 468, 082, 573	Fowler, Noll, Vo

Multiplicative hash function

Magic	Hash function
0	Black hole
1	Identity
3.14	Full circle
256	Byte oriented
2018	ESUG 2018 Conference
1234567890	Diversity
9876543210	Diversity the Great
10^10^100	googleplex