Hash Tables

Data Structures for *Map ADT*

	Find	Insert	Delete
Unsorted Array	$\theta(n)$	$\theta(n)$	$\theta(n)$
Unsorted Linked List	$\theta(n)$	$\theta(n)$	$\theta(n)$
Sorted Array	$\theta(\log(n))$	$\theta(n)$	$\theta(n)$
Sorted Linked List	$\theta(n)$	$\theta(n)$	$\theta(n)$
Binary Search Tree	$\theta(h), \theta(n)$	$\theta(h), \theta(n)$	$\theta(h), \theta(n)$
AVL Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$

Data Structures for *Map ADT*

	Find	Insert	Delete
Unsorted Array	$\theta(n)$	$\theta(n)$	$\theta(n)$
Unsorted Linked List	$\theta(n)$	$\theta(n)$	$\theta(n)$
Sorted Array	$\theta(\log(n))$	$\theta(n)$	$\theta(n)$
Sorted Linked List	$\theta(n)$	$\theta(n)$	$\theta(n)$
Binary Search Tree	$\theta(h), \theta(n)$	$\theta(h), \theta(n)$	$\theta(h), \theta(n)$
AVL Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$
Bucket Array			

Data Structures for *Map ADT*

	Find	Insert	Delete
Unsorted Array	$\theta(n)$	$\theta(n)$	$\theta(n)$
Unsorted Linked List	$\theta(n)$	$\theta(n)$	$\theta(n)$
Sorted Array	$\theta(\log(n))$	$\theta(n)$	$\theta(n)$
Sorted Linked List	$\theta(n)$	$\theta(n)$	$\theta(n)$
Binary Search Tree	$\theta(h), \theta(n)$	$\theta(h), \theta(n)$	$\theta(h), \theta(n)$
AVL Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$
Bucket Array	$\theta(1)$	$\theta(1)$	$\theta(1)$

Data Structures for *Map ADT*

	Find	Insert	Delete	Space
Unsorted Array	$\theta(n)$	$\theta(n)$	$\theta(n)$	
Unsorted Linked List	$\theta(n)$	$\theta(n)$	$\theta(n)$	
Sorted Array	$\theta(\log(n))$	$\theta(n)$	$\theta(n)$	
Sorted Linked List	$\theta(n)$	$\theta(n)$	$\theta(n)$	
Binary Search Tree	$\theta(h), \theta(n)$	$\theta(h), \theta(n)$	$\theta(h), \theta(n)$	
AVL Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	
Bucket Array	$\theta(1)$	$\theta(1)$	$\theta(1)$	

Data Structures for *Map ADT*

	Find	Insert	Delete	Space
Unsorted Array	$\theta(n)$	$\theta(n)$	$\theta(n)$	
Unsorted Linked List	$\theta(n)$	$\theta(n)$	$\theta(n)$	
Sorted Array	$\theta(\log(n))$	$\theta(n)$	$\theta(n)$	
Sorted Linked List	$\theta(n)$	$\theta(n)$	$\theta(n)$	
Binary Search Tree	$\theta(h), \theta(n)$	$\theta(h), \theta(n)$	$\theta(h), \theta(n)$	
AVL Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	
Bucket Array	$\theta(1)$	$\theta(1)$	$\theta(1)$	Could be large relative to n

Data Structures for *Map ADT*

	Find	Insert	Delete	Space
Unsorted Array	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Unsorted Linked List	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Sorted Array	$\theta(\log(n))$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Sorted Linked List	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Binary Search Tree	$\theta(h), \theta(n)$	$\theta(h), \theta(n)$	$\theta(h), \theta(n)$	$\theta(\mathrm{n})$
AVL Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\mathrm{n})$
Bucket Array	$\theta(1)$	$\theta(1)$	$\theta(1)$	Could be large relative to n

Data Structures for *Map ADT*

	Find	Insert	Delete	Space
Unsorted Array	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Unsorted Linked List	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Sorted Array	$\theta(\log(n))$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Sorted Linked List	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Binary Search Tree	$\theta(h), \theta(n)$	$\theta(h), \theta(n)$	$\theta(h), \theta(n)$	$\theta(\mathrm{n})$
AVL Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\mathrm{n})$
Bucket Array	$\theta(1)$	$\theta(1)$	$\theta(1)$	Could be large relative to n
Hash Table				

Data Structures for *Map ADT*

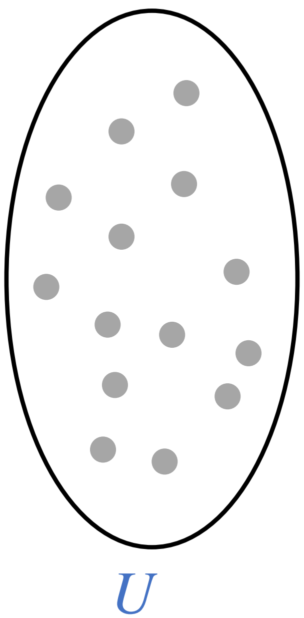
	Find	Insert	Delete	Space
Unsorted Array	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Unsorted Linked List	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Sorted Array	$\theta(\log(n))$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Sorted Linked List	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Binary Search Tree	$\theta(h), \theta(n)$	$\theta(h), \theta(n)$	$\theta(h), \theta(n)$	$\theta(\mathrm{n})$
AVL Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\mathrm{n})$
Bucket Array	$\theta(1)$	$\theta(1)$	$\theta(1)$	Could be large relative to n
Hash Table				$\theta(\mathrm{n})$

Data Structures for *Map ADT*

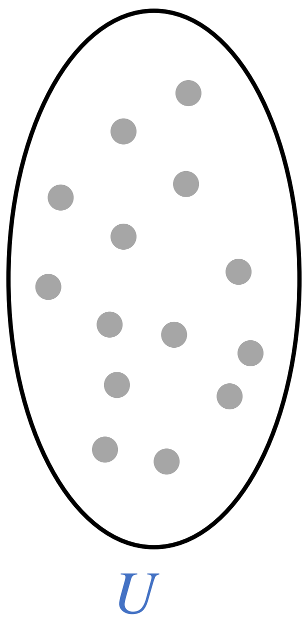
	Find	Insert	Delete	Space
Unsorted Array	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Unsorted Linked List	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Sorted Array	$\theta(\log(n))$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Sorted Linked List	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Binary Search Tree	$\theta(h), \theta(n)$	$\theta(h), \theta(n)$	$\theta(h), \theta(n)$	$\theta(\mathrm{n})$
AVL Tree	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\mathrm{n})$
Bucket Array	$\theta(1)$	$\theta(1)$	$\theta(1)$	Could be large relative to n
Hash Table	$\theta(1)$	$\theta(1)$	$\theta(1)$	$\theta(\mathrm{n})$

Hash Tables

Hash Tables

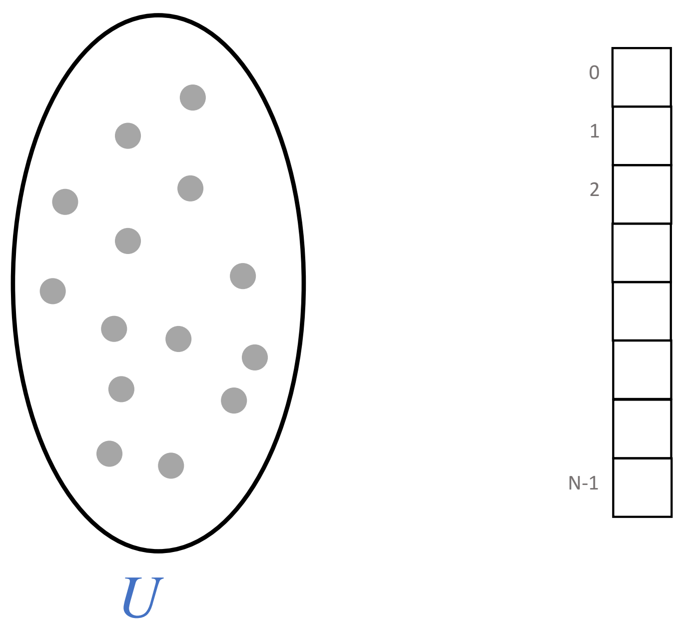


Hash Tables



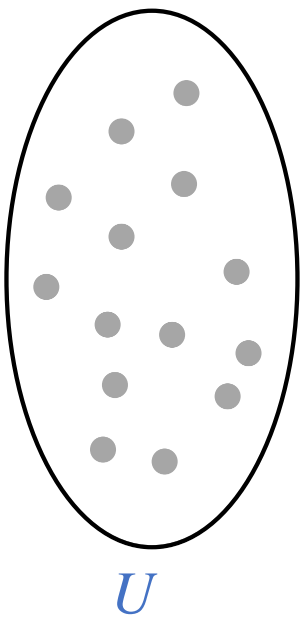
- U
  Universe from which the keys will be taken

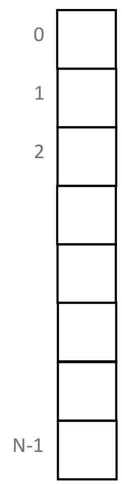
Hash Tables



- U
  Universe from which the keys will be taken

Hash Tables

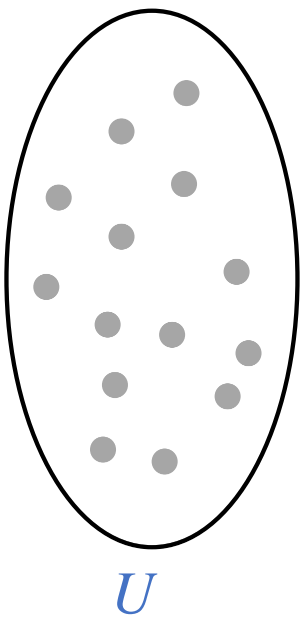


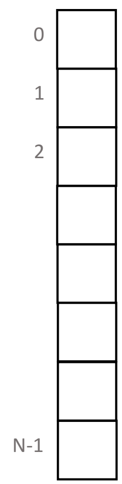


- U
  Universe from which the keys will be taken

- T = [None]*N

Hash Tables

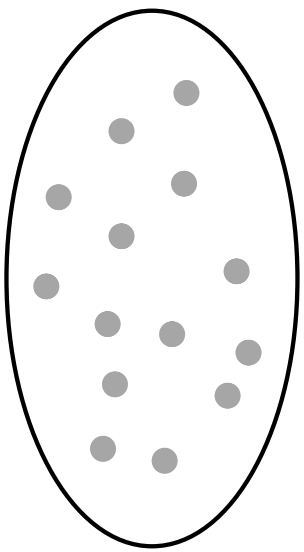




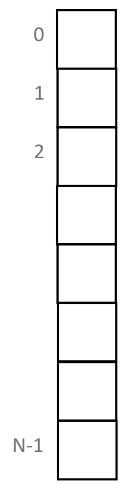
- U
  Universe from which the keys will be taken

- $T = [None]*N$
  Hash table of N slots, where the entries will be stored

Hash Tables



U

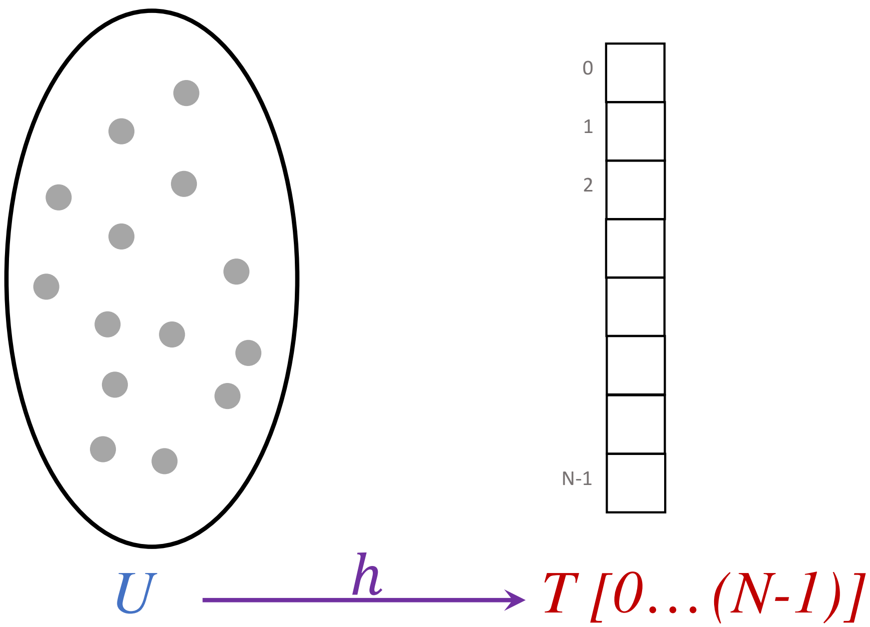


T [0…(N-1)]

- *U*
  Universe from which the keys will be taken

- *T = [None]*N*
  Hash table of N slots, where the entries will be stored

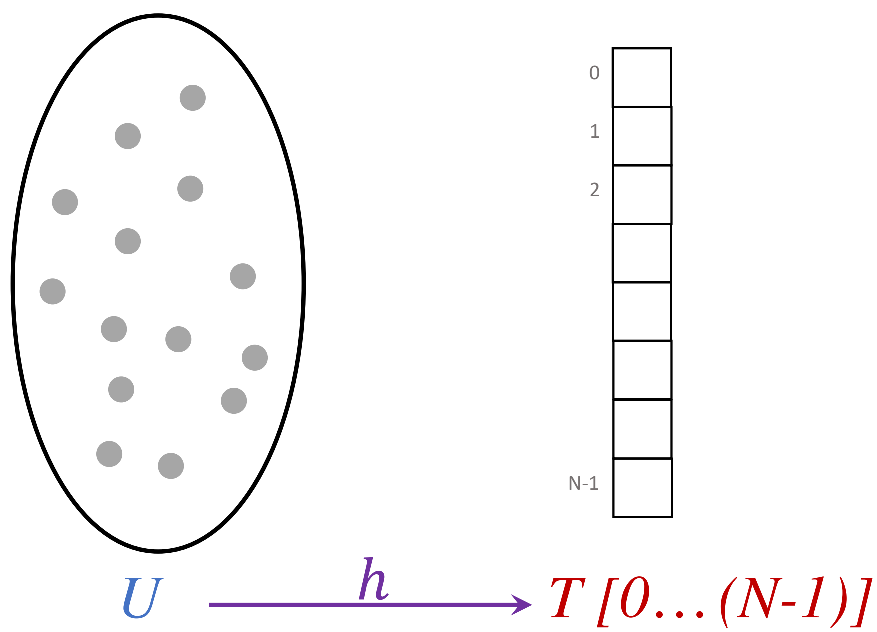
Hash Tables



- U
  Universe from which the keys will be taken

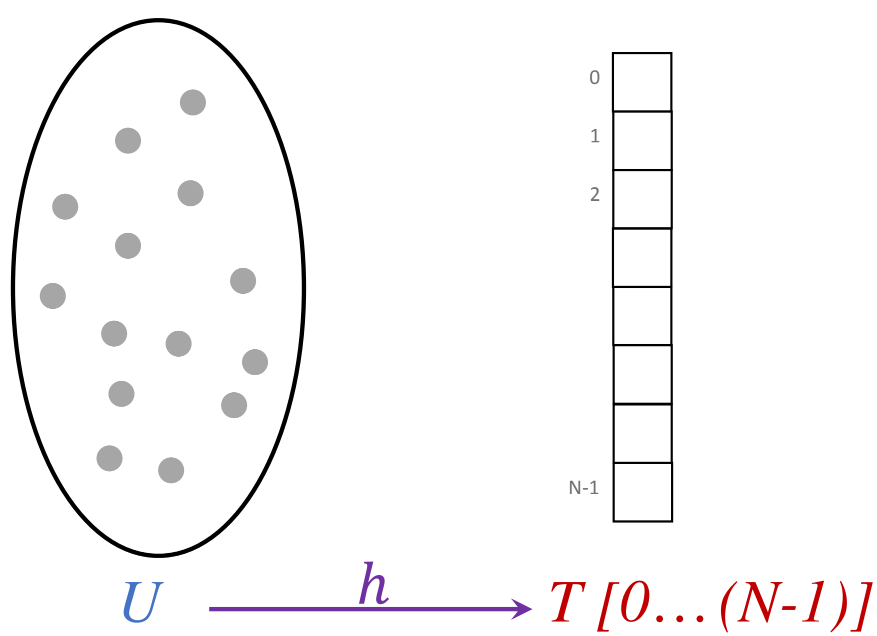
- $T = [None]*N$
  Hash table of N slots, where the entries will be stored

Hash Tables



- U
  Universe from which the keys will be taken
- $T = [None]*N$
  Hash table of N slots, where the entries will be stored
- $h: U \rightarrow \{0, 1, \ \ldots, (N-1)\}$

Hash Tables

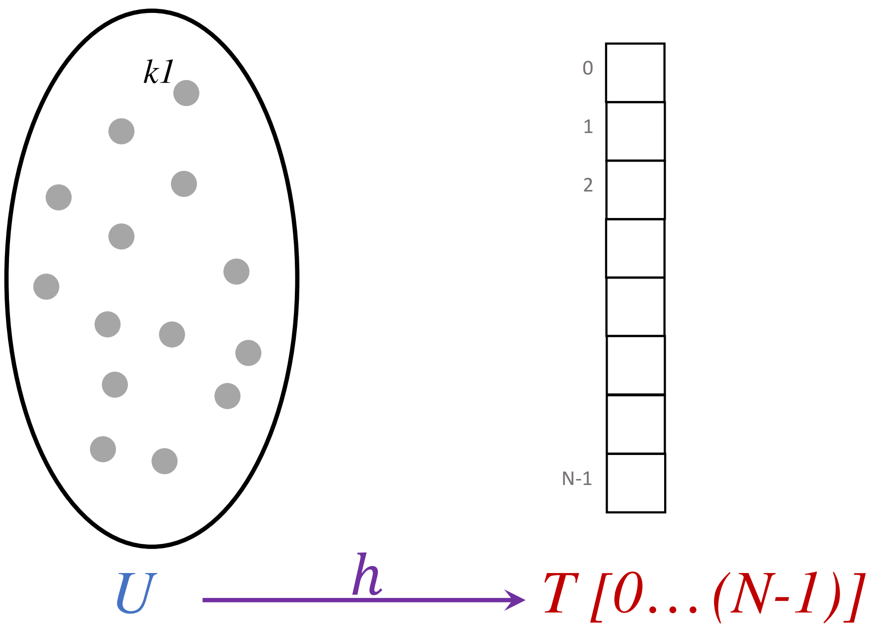


- U
  Universe from which the keys will be taken

- T = [None]*N
  Hash table of N slots, where the entries will be stored

- $h: U \rightarrow \{0, 1, \ \ldots, (N-1)\}$
  Hash function that maps keys form the universe to slots in the table

Hash Tables

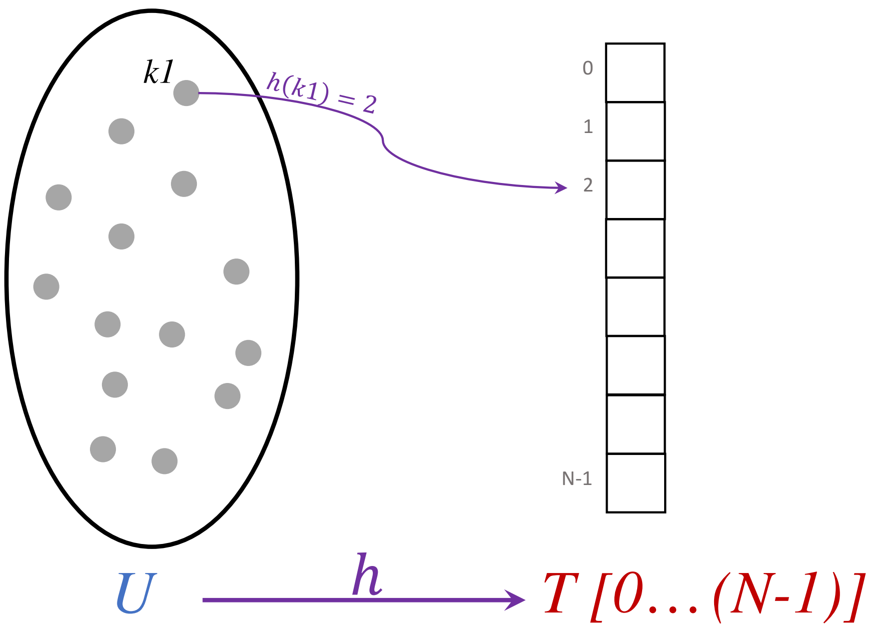


- U
  Universe from which the keys will be taken

- $T = [None]*N$
  Hash table of N slots, where the entries will be stored

- $h: U \rightarrow \{0, 1, \ \ldots, (N-1)\}$
  Hash function that maps keys form the universe to slots in the table

Hash Tables

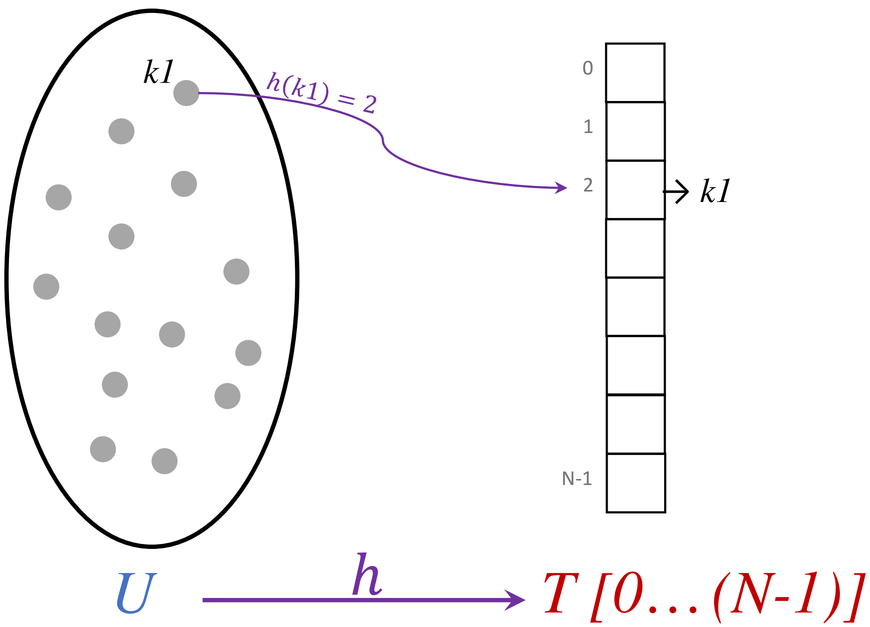


- U
  Universe from which the keys will be taken

- $T = [None]*N$
  Hash table of N slots, where the entries will be stored

- $h: U \rightarrow \{0, 1, \ \ldots, (N-1)\}$
  Hash function that maps keys form the universe to slots in the table

Hash Tables

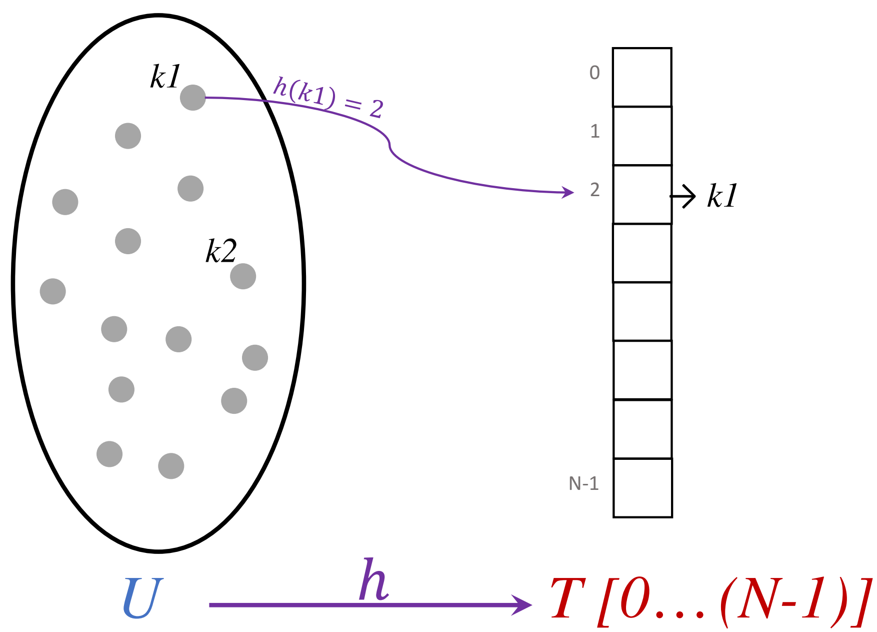


- U
  Universe from which the keys will be taken

- T = [None]*N
  Hash table of N slots, where the entries will be stored

- $h: U \rightarrow \{0, 1, \ \ldots, (N-1)\}$
  Hash function that maps keys form the universe to slots in the table

Hash Tables

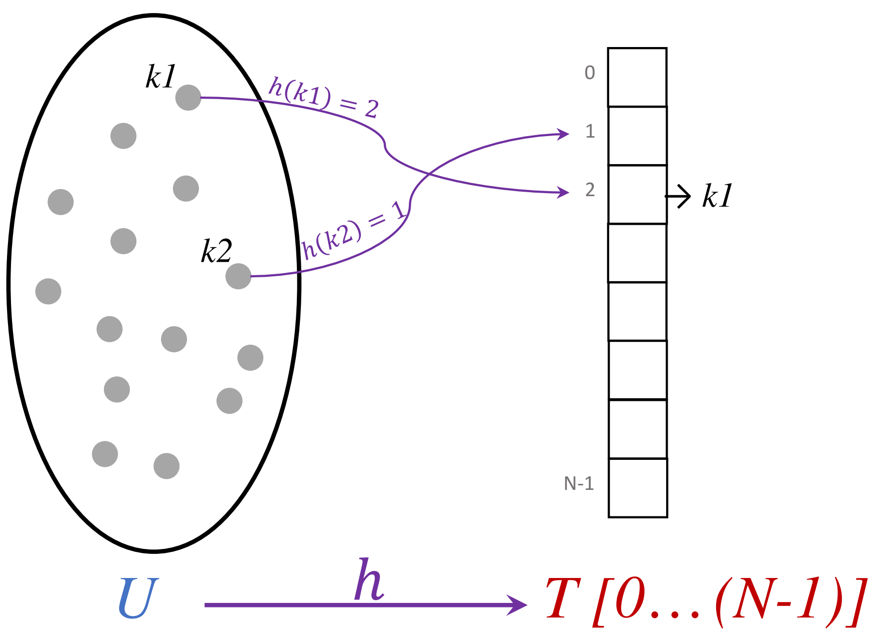


- U
  Universe from which the keys will be taken

- T = [None]*N
  Hash table of N slots, where the entries will be stored

- $h: U \rightarrow \{0, 1, \ \dots, (N-1)\}$
  Hash function that maps keys form the universe to slots in the table

Hash Tables

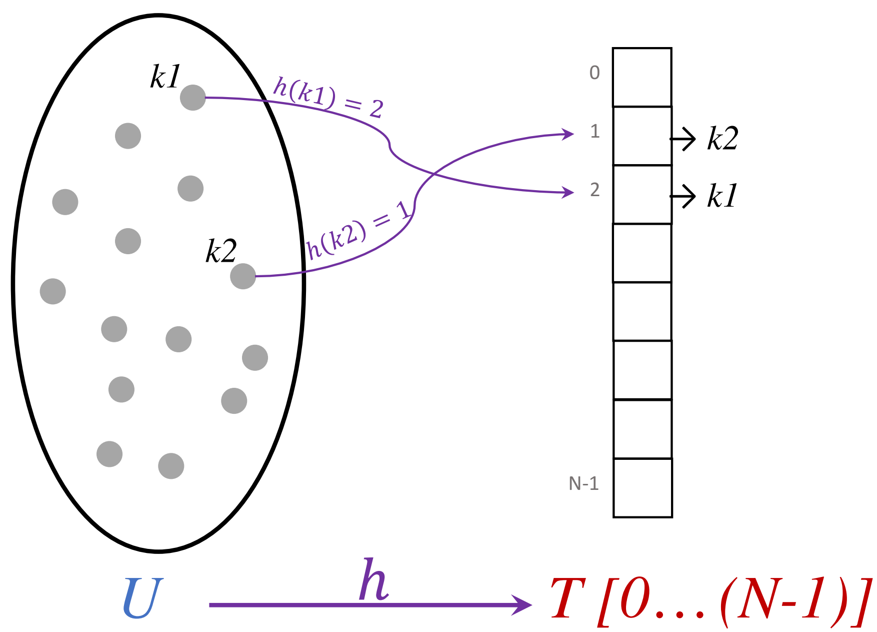


- U
  Universe from which the keys will be taken

- T = [None]*N
  Hash table of N slots, where the entries will be stored

- $h: U \rightarrow \{0, 1, \ \ldots, (N-1)\}$
  Hash function that maps keys form the universe to slots in the table

Hash Tables

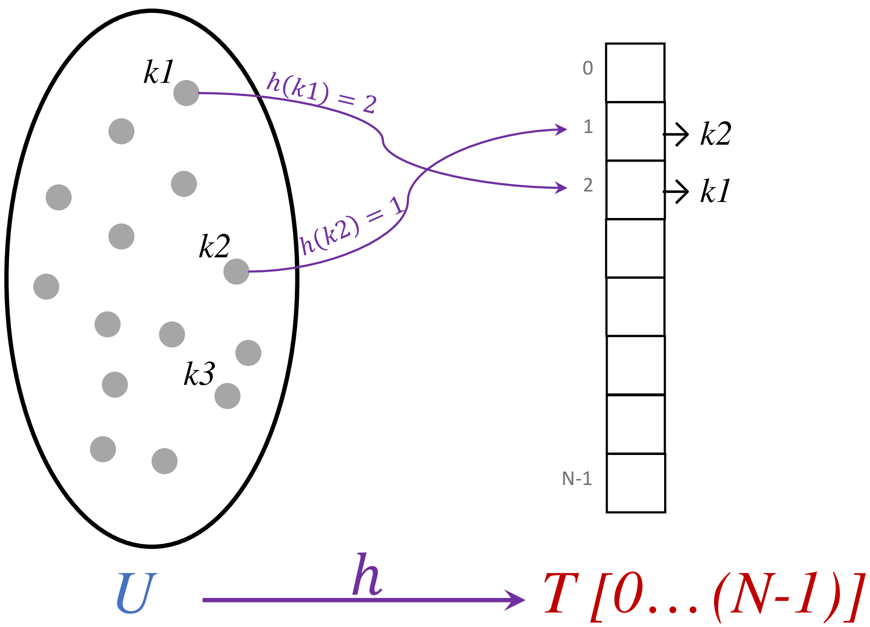


- U
  Universe from which the keys will be taken

- T = [None]*N
  Hash table of N slots, where the entries will be stored

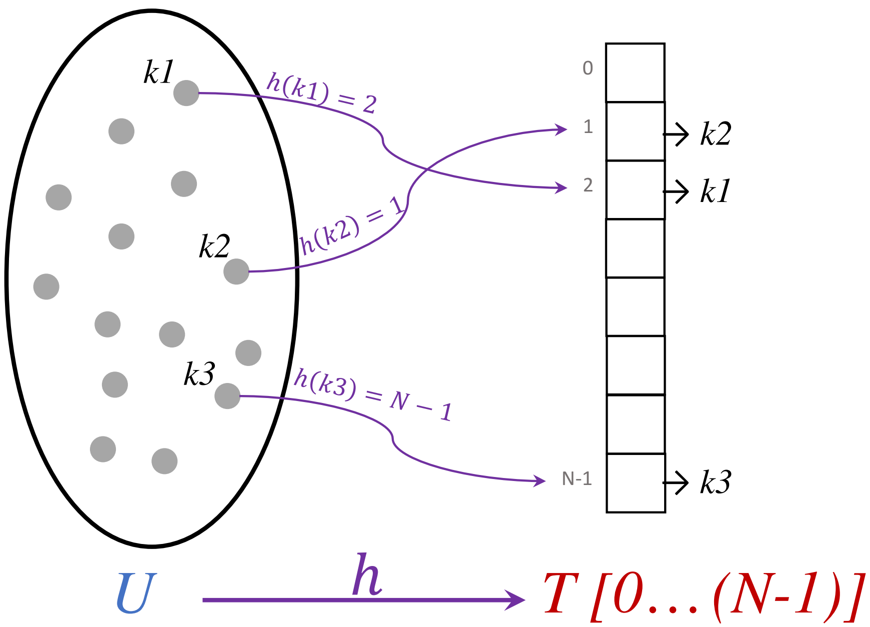
- $h: U \rightarrow \{0, 1, \ \ldots, (N-1)\}$
  Hash function that maps keys form the universe to slots in the table

Hash Tables



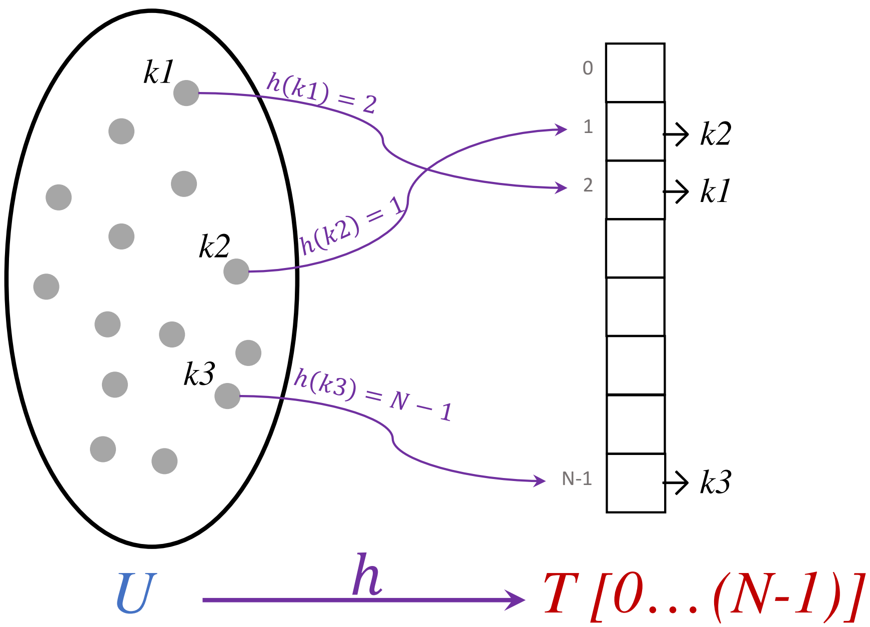
- U
  Universe from which the keys will be taken
- $T = [None]*N$
  Hash table of N slots, where the entries will be stored
- $h: U \rightarrow \{0, 1, \ \ldots, (N-1)\}$
  Hash function that maps keys form the universe to slots in the table

Hash Tables

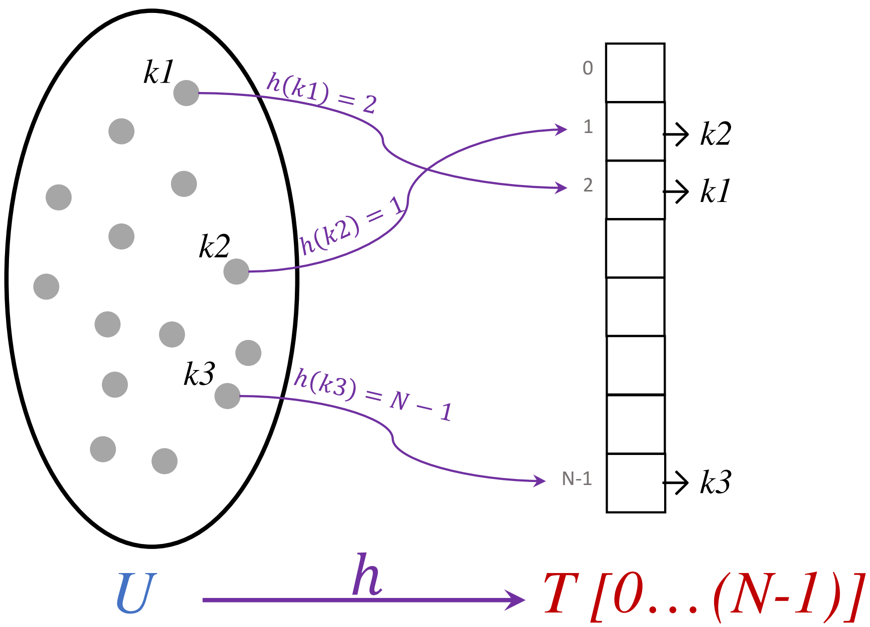


- *U*
  Universe from which the keys will be taken
- *T = [None]*N*
  Hash table of N slots, where the entries will be stored
- $h: U \rightarrow \{0, 1, \ \ldots, (N-1)\}$
  Hash function that maps keys form the universe to slots in the table

Hash Tables

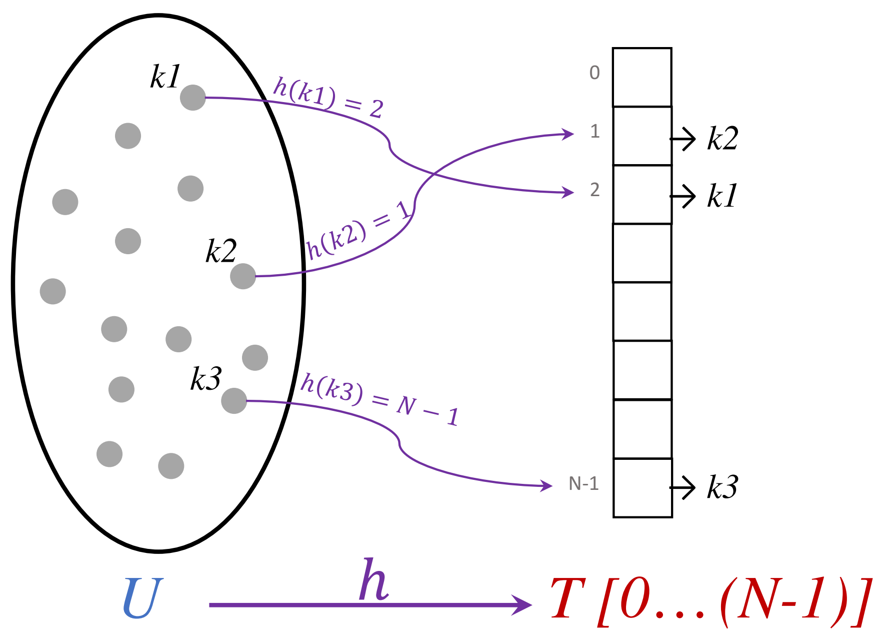


Hash Tables



insert(key, value):

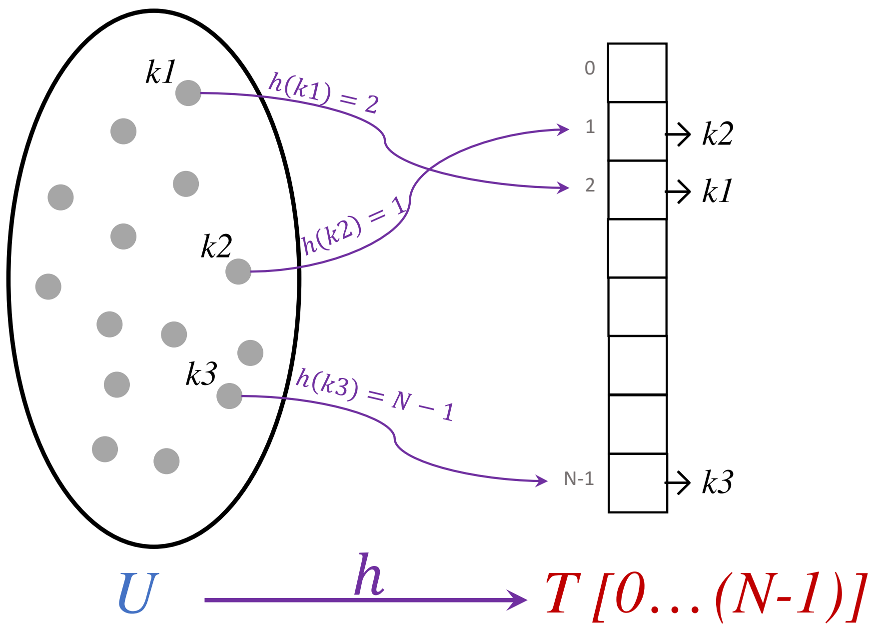
Hash Tables



insert(key, value):

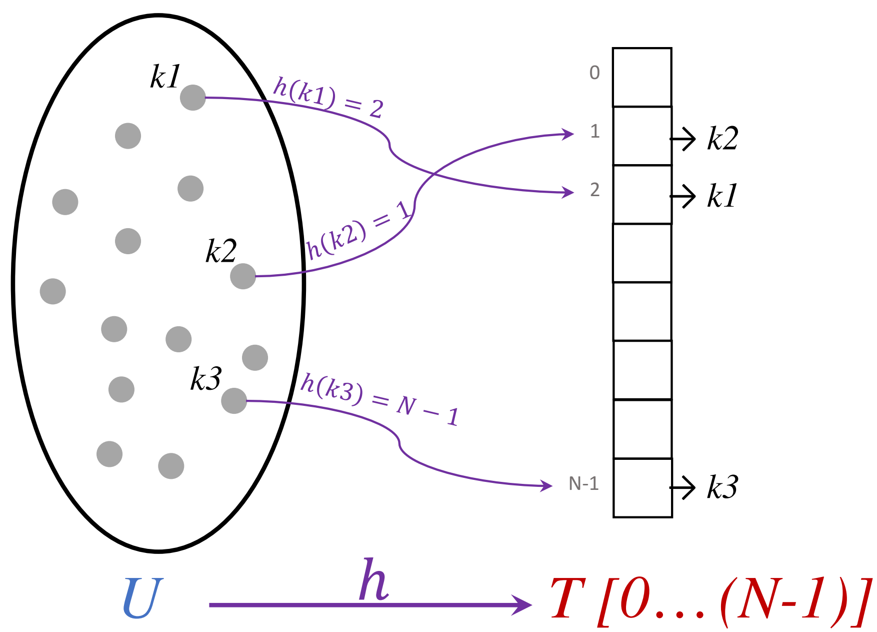
$i = h(key)$

Hash Tables



insert(key, value):
i = h(key)
T[i] = value

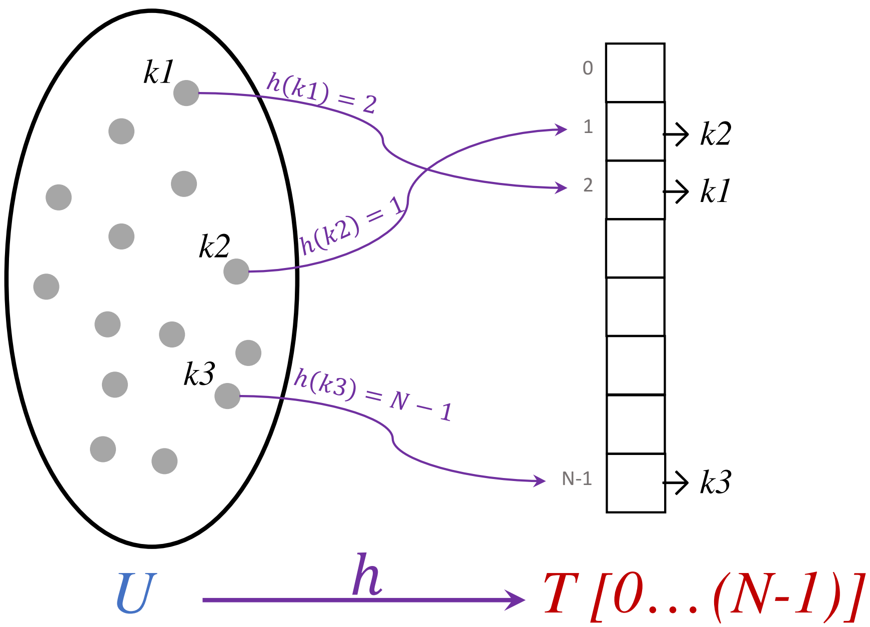
Hash Tables



insert(key, value):
i = h(key)
T[i] = value

find(key):

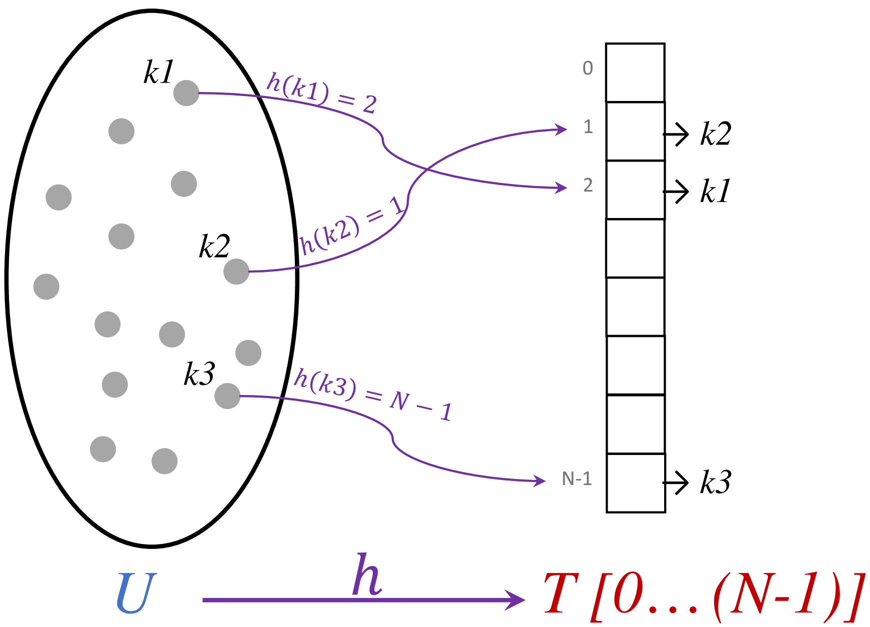
Hash Tables



insert(key, value):
i = h(key)
T[i] = value

find(key):
i = h(key)

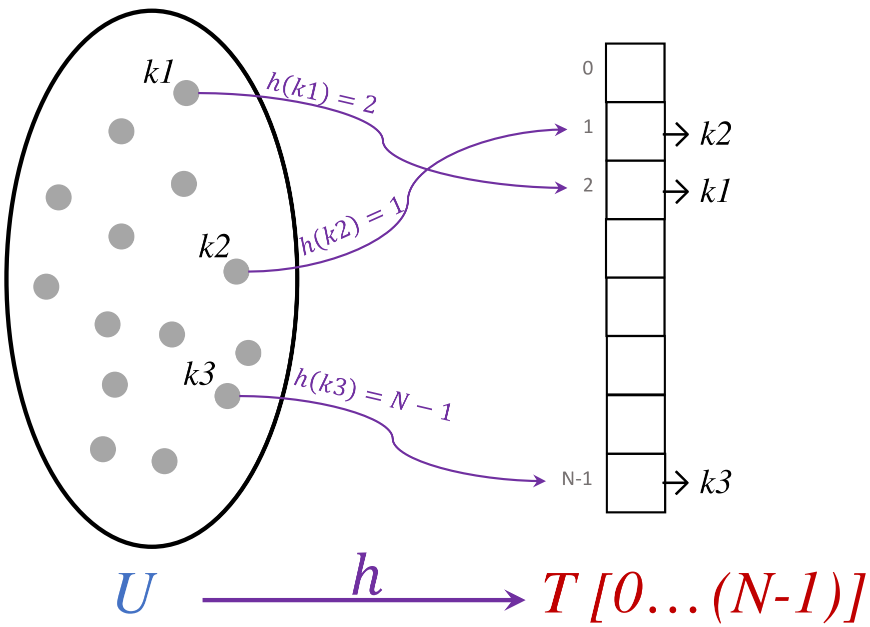
Hash Tables



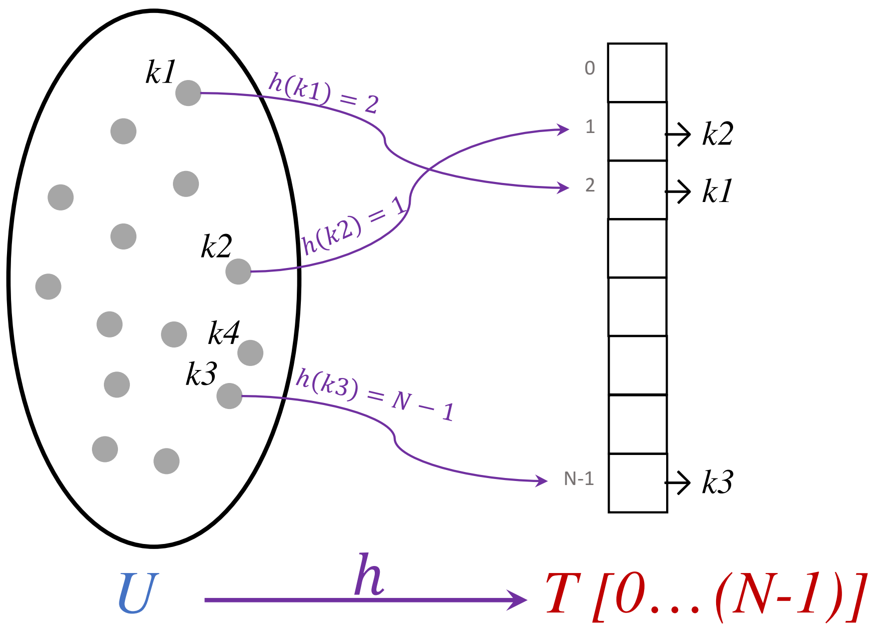
insert(key, value)
i = h(key)
T[i] = value

find(key)
i = h(key)
return T[i]

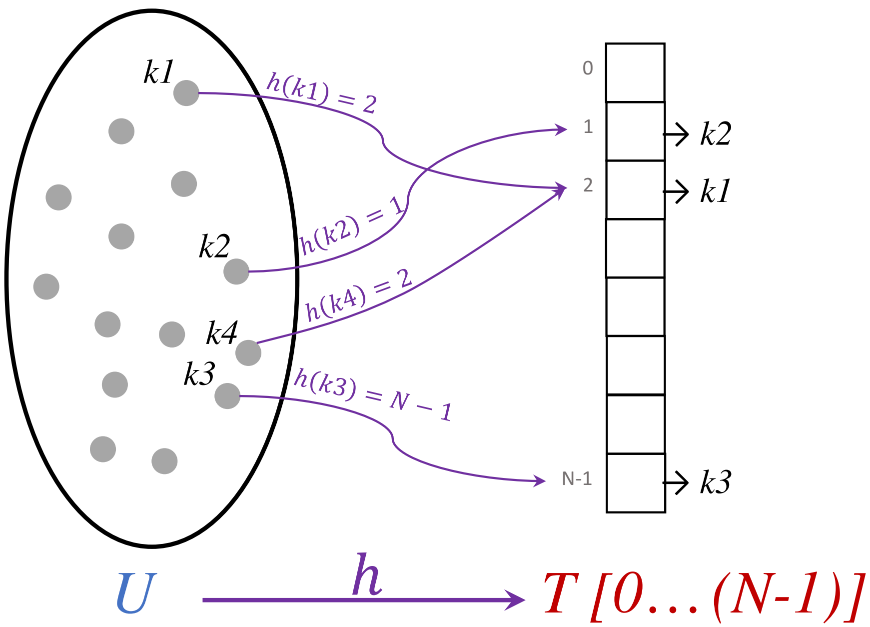
Hash Tables



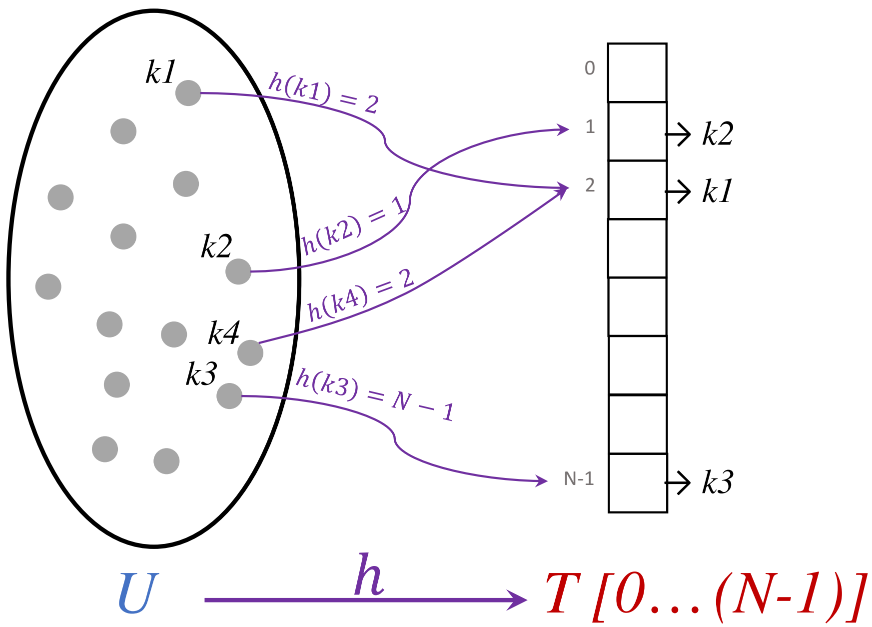
Hash Tables



Hash Tables

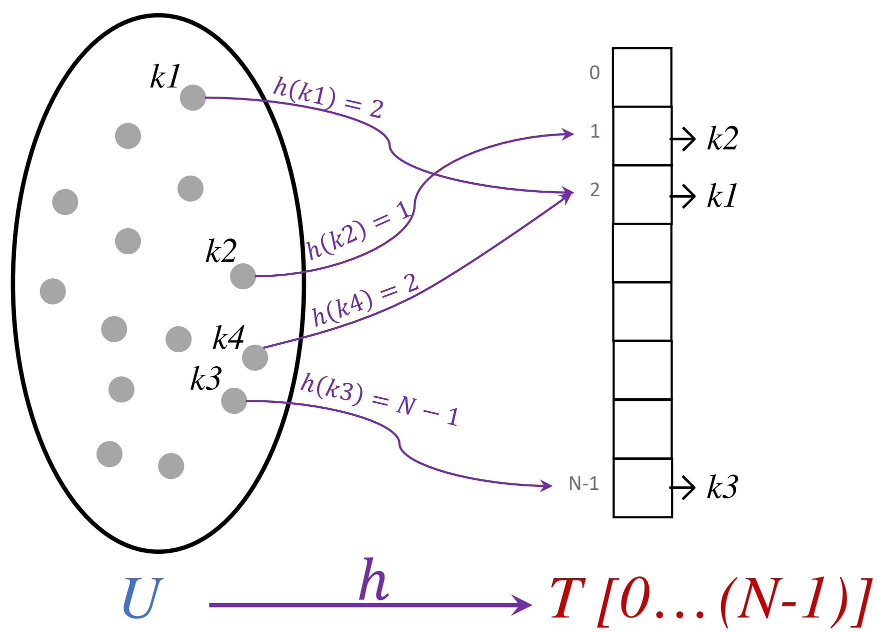


Hash Tables



The Collisions Problem:
multiple keys could be mapped to the same slot

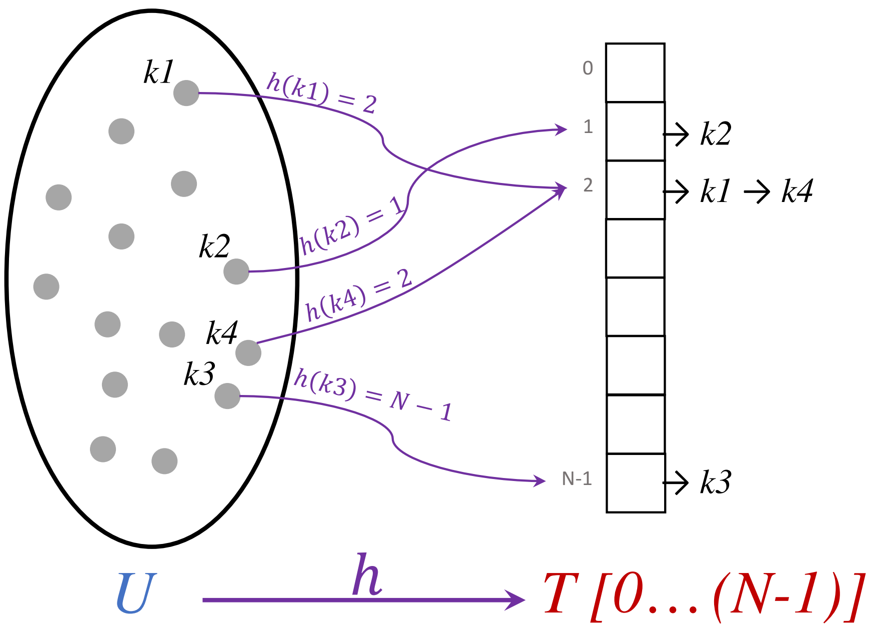
Hash Tables



<u>The Collisions Problem</u>:
multiple keys could be mapped to the same slot

<u>An easy solution</u>:
Chaining – store all entries that are mapped to the same slot, in some secondary collection

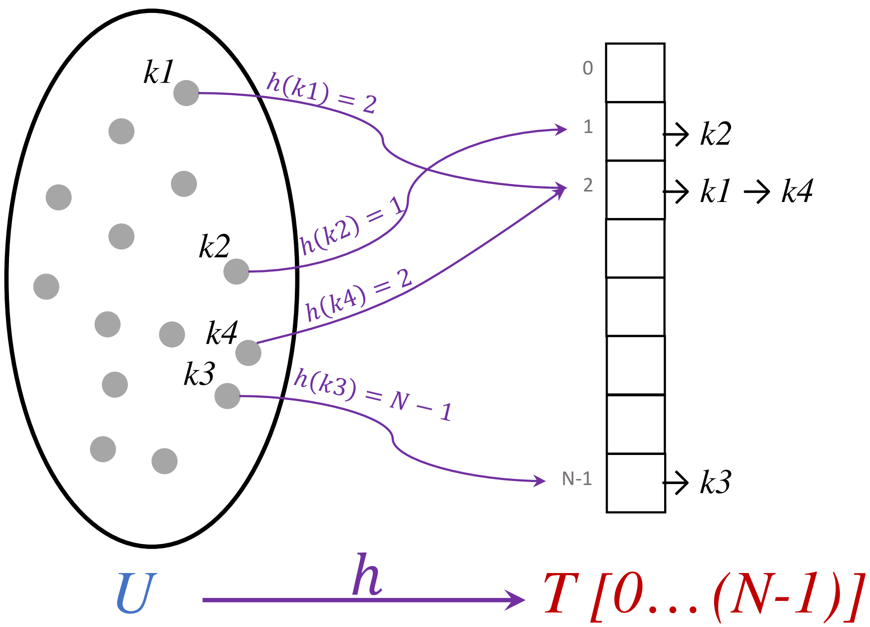
Hash Tables



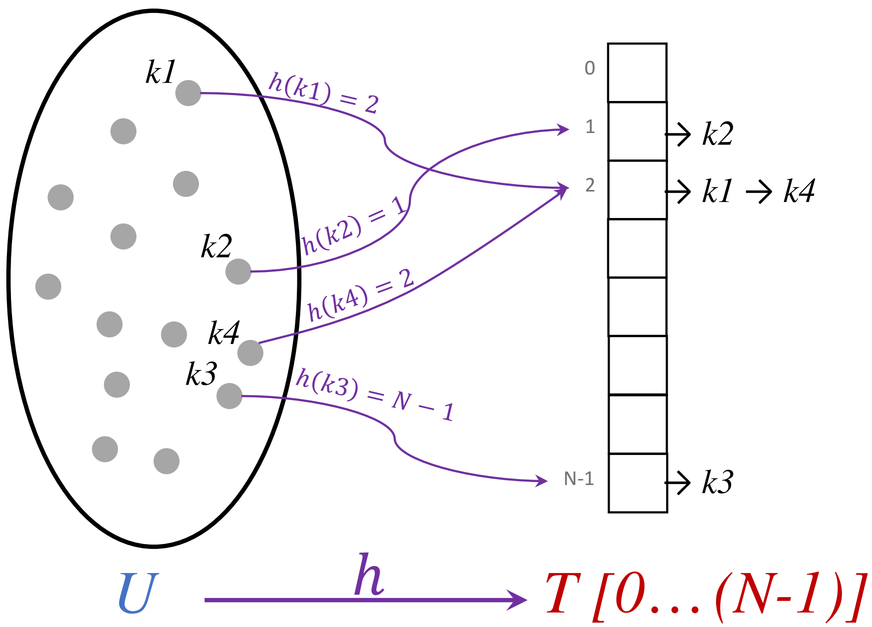
The Collisions Problem:
multiple keys could be mapped to the same slot

An easy solution:
Chaining – store all entries that are mapped to the same slot, in some secondary collection

Hash Tables

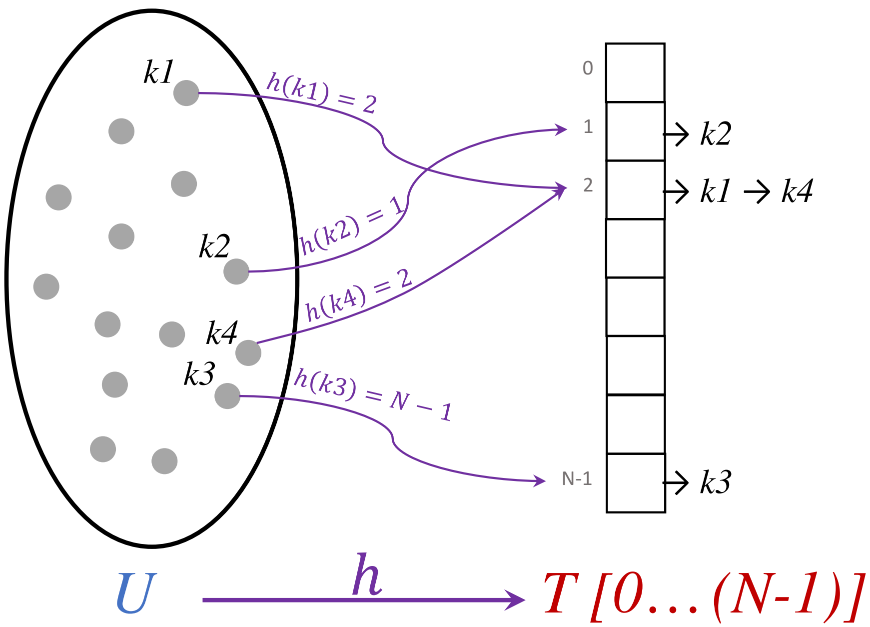


Hash Tables



Problem: When using chaining, a lot of keys could end up being stored at the same slot $\Rightarrow$ bad performance

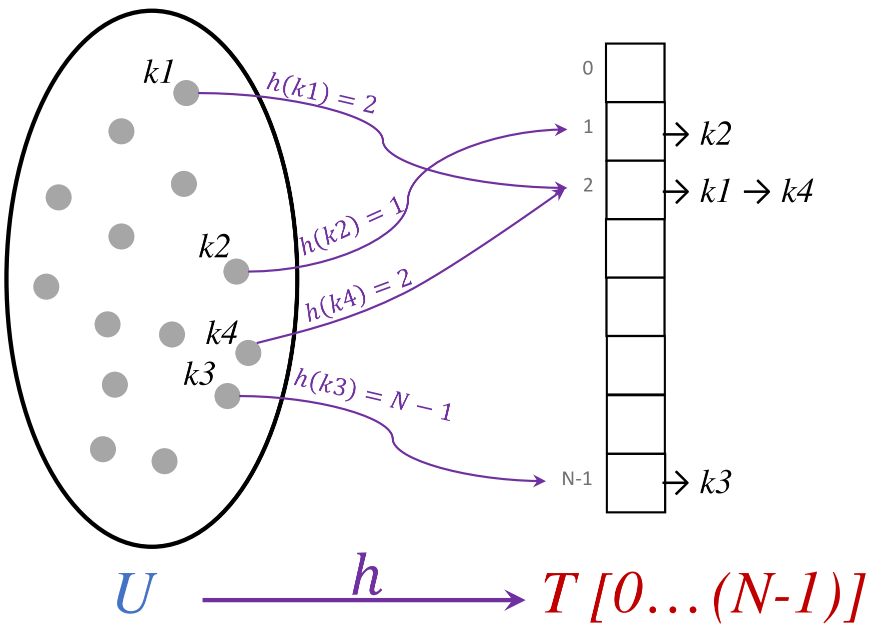
Hash Tables



Problem: When using chaining, a lot of keys could end up being stored at the same slot $\Rightarrow$ bad performance

Solution: Use a "good" hash function (a function that evens out the collisions)

Hash Tables



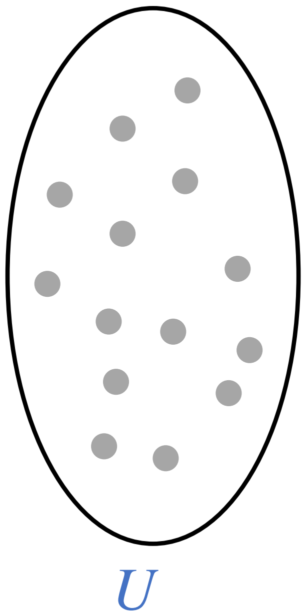
Problem: When using chaining, a lot of keys could end up being stored at the same slot $\Rightarrow$ bad performance

Solution: Use a "good" hash function (a function that evens out the collisions)

Uniform Hashing Function:
A function that when given a randomly chosen key, it will be equally likely mapped to any of the N slots of T, independently of where any other key has hashed to

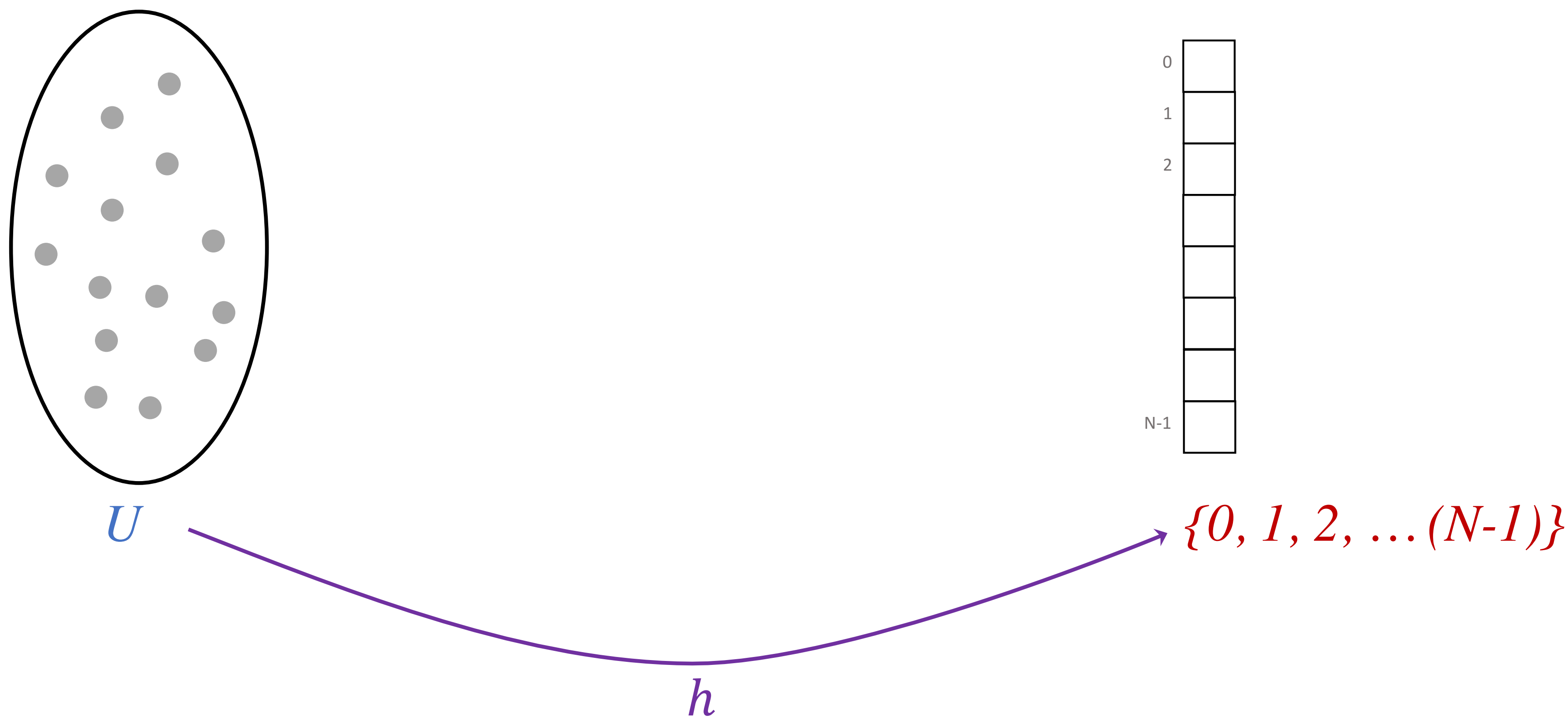
Hash Functions

Hash Functions

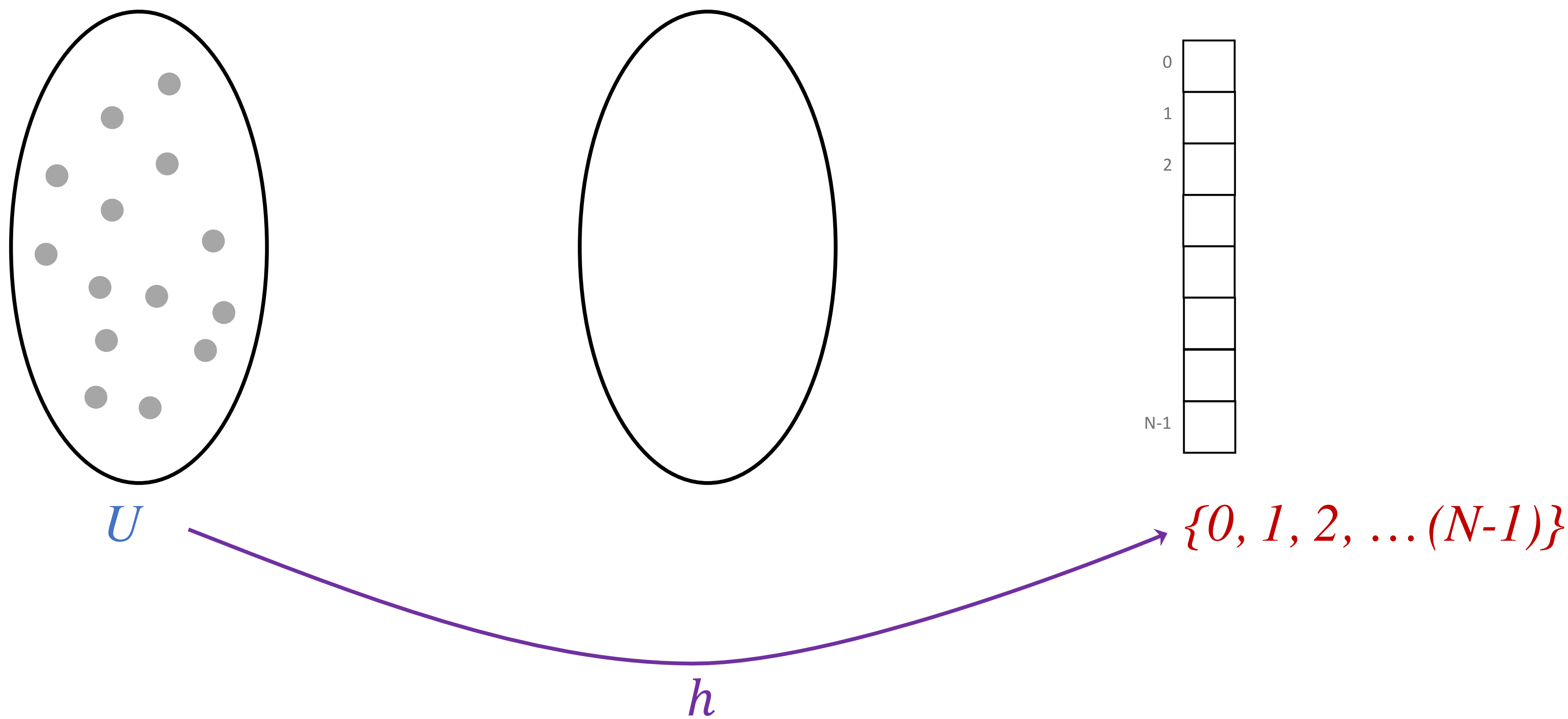


Hash Functions

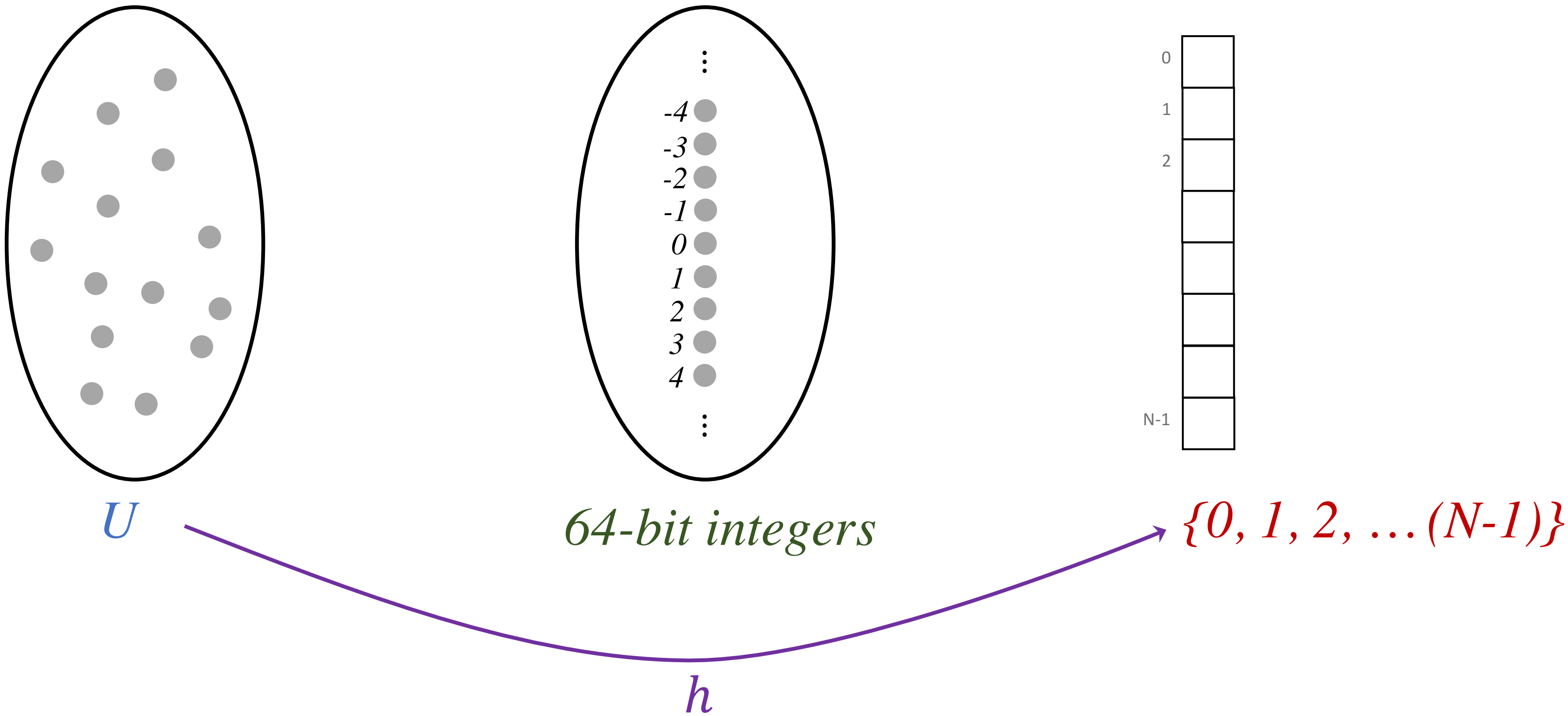
Hash Functions



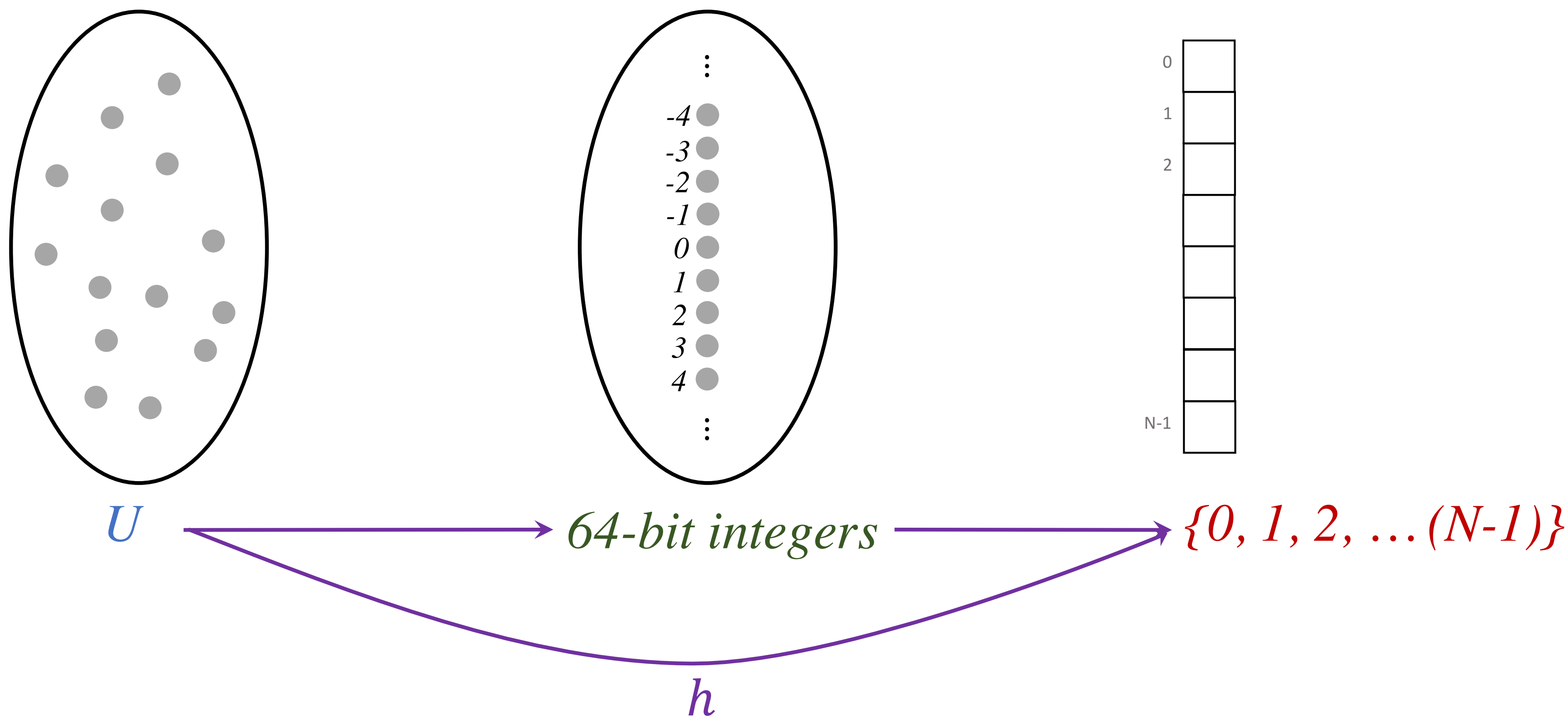
Hash Functions



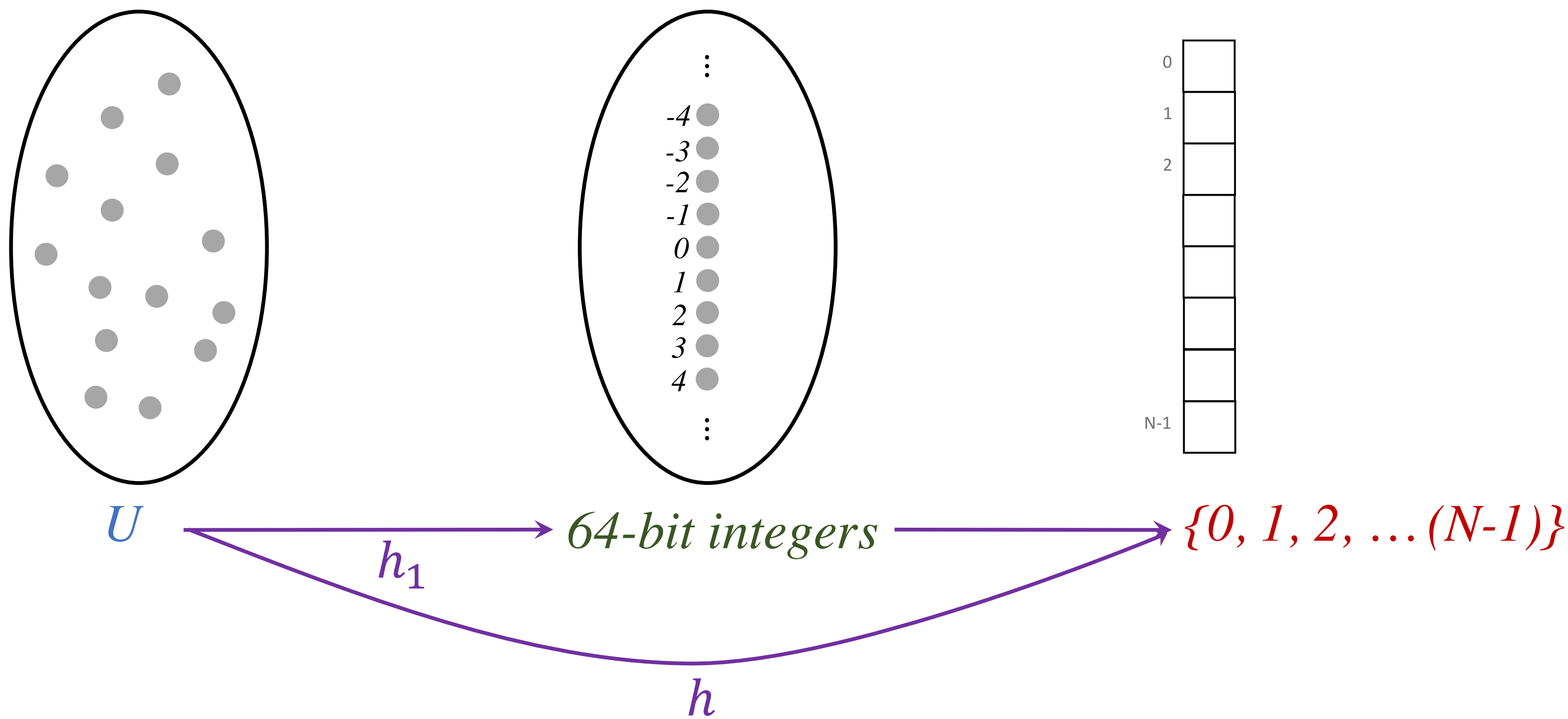
Hash Functions



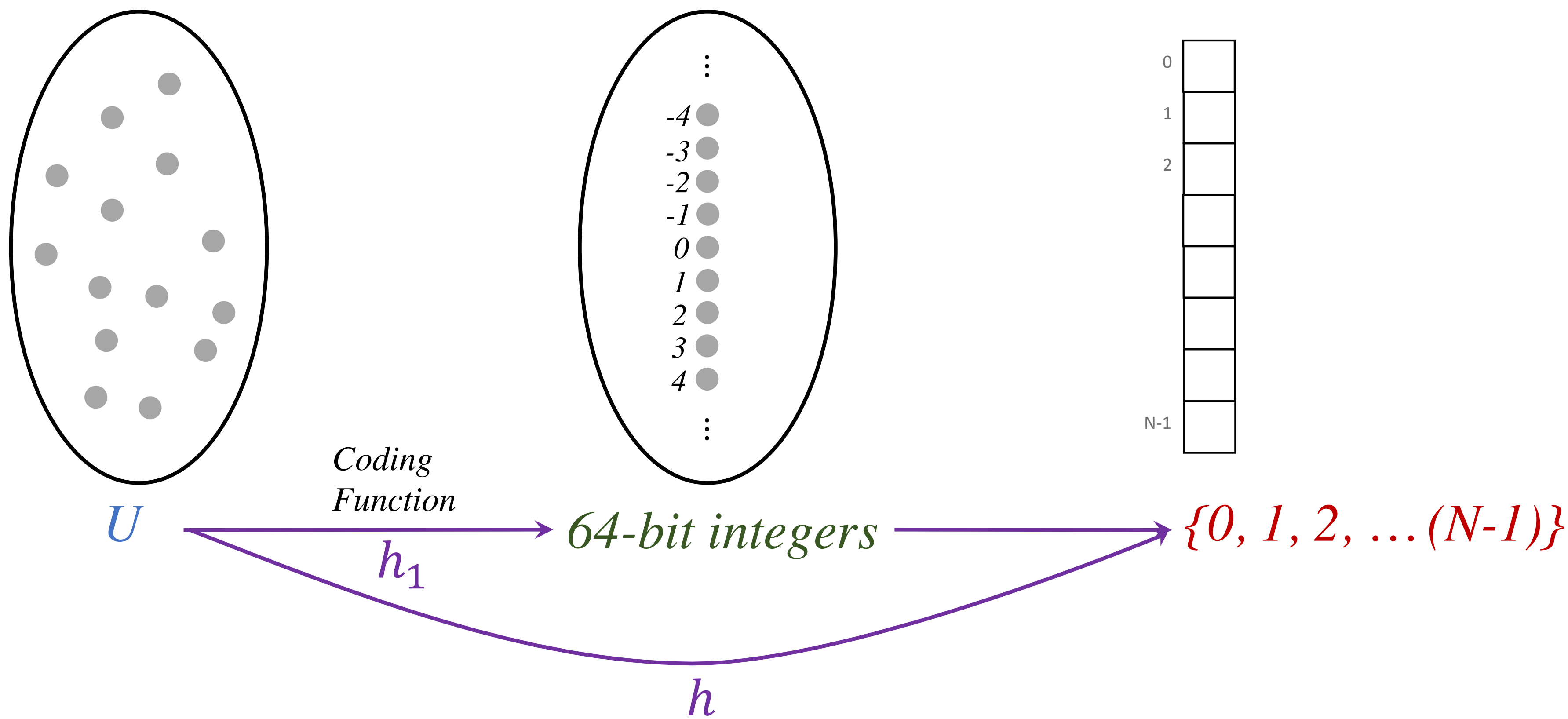
Hash Functions



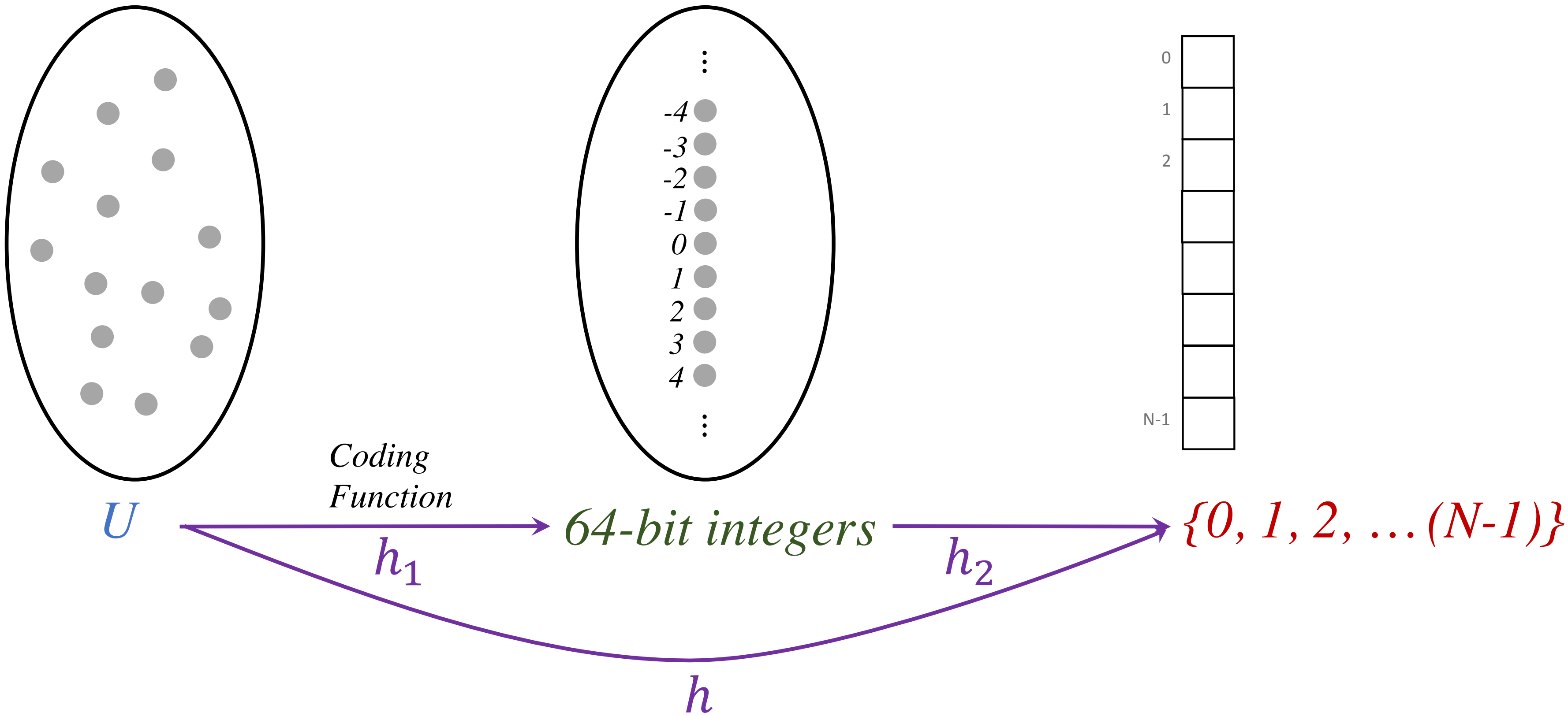
Hash Functions



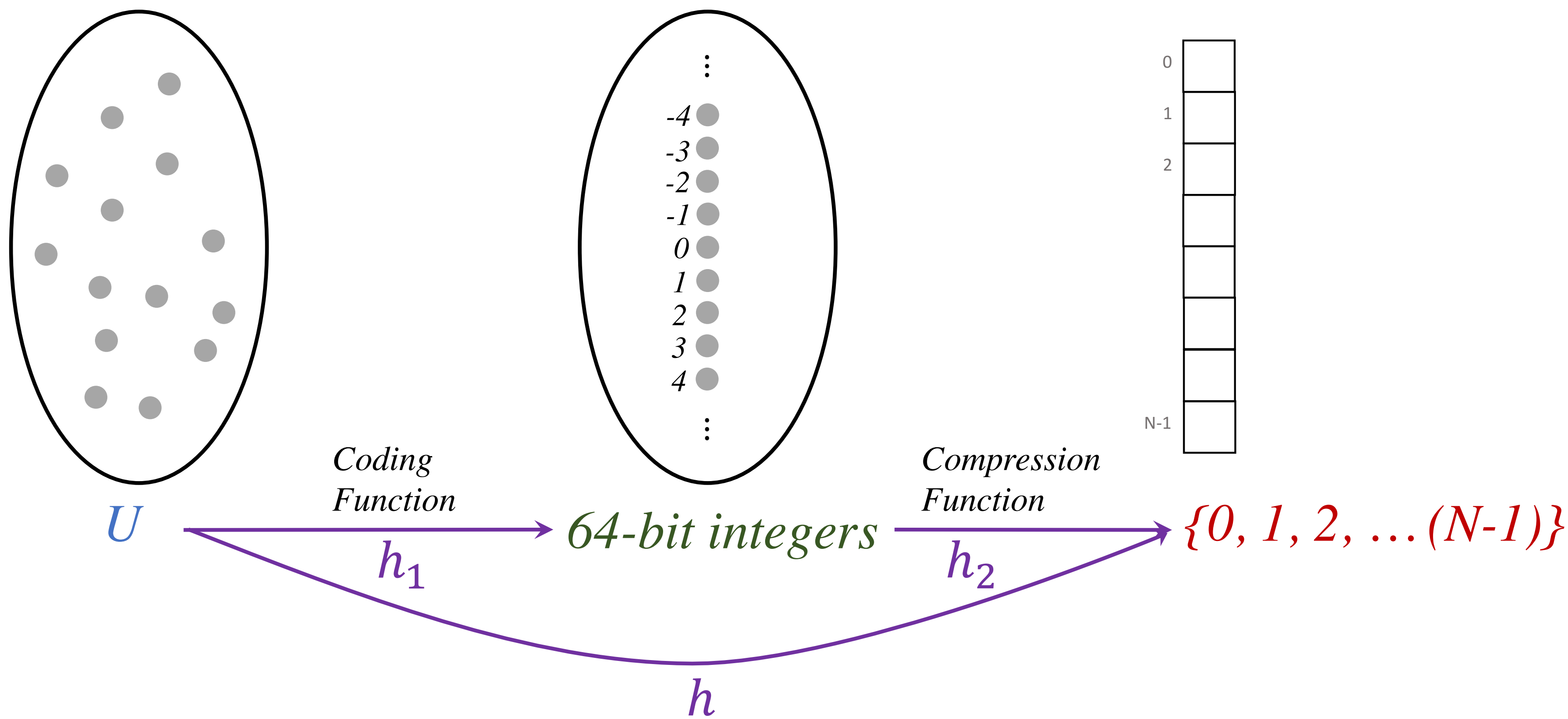
Hash Functions



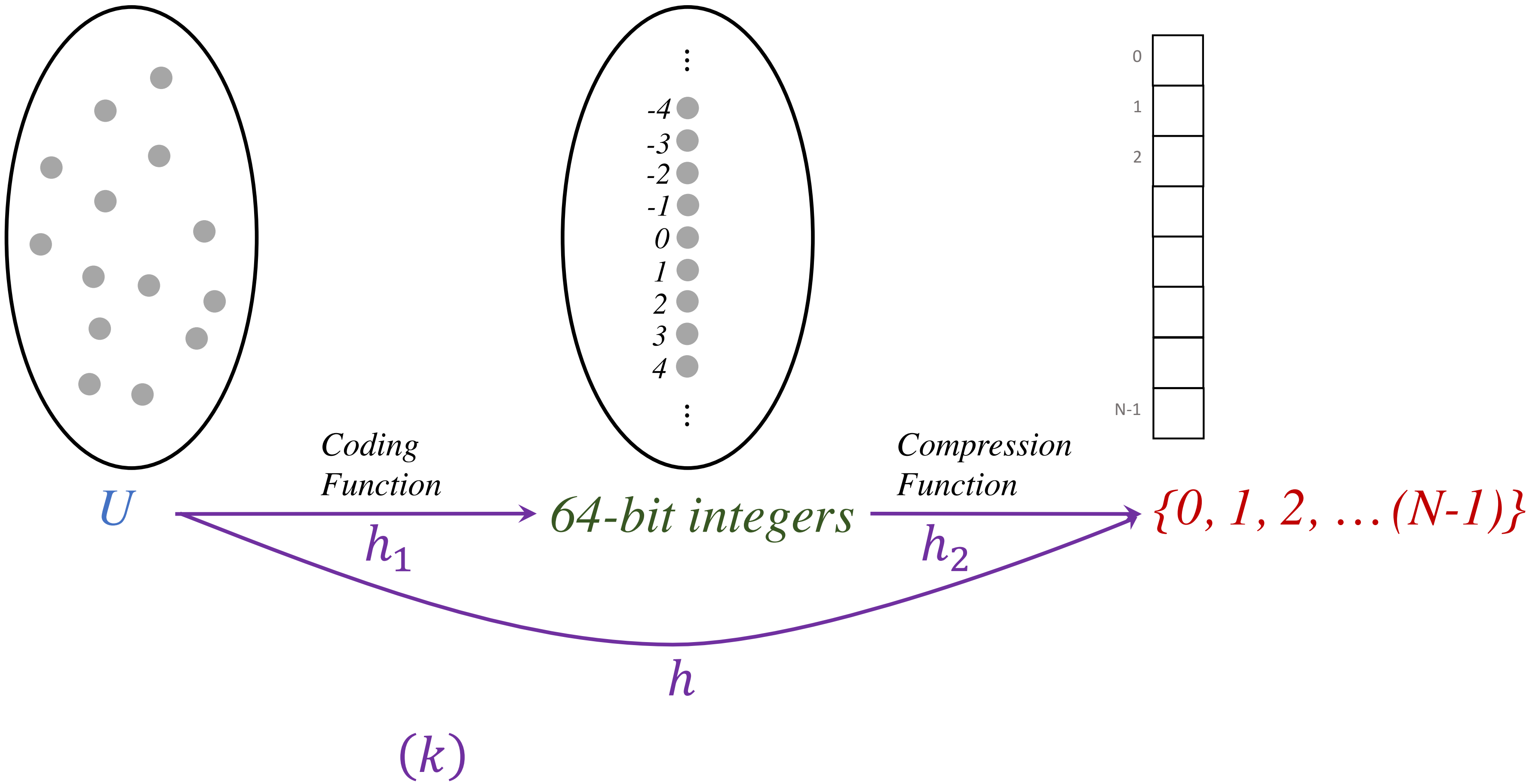
Hash Functions



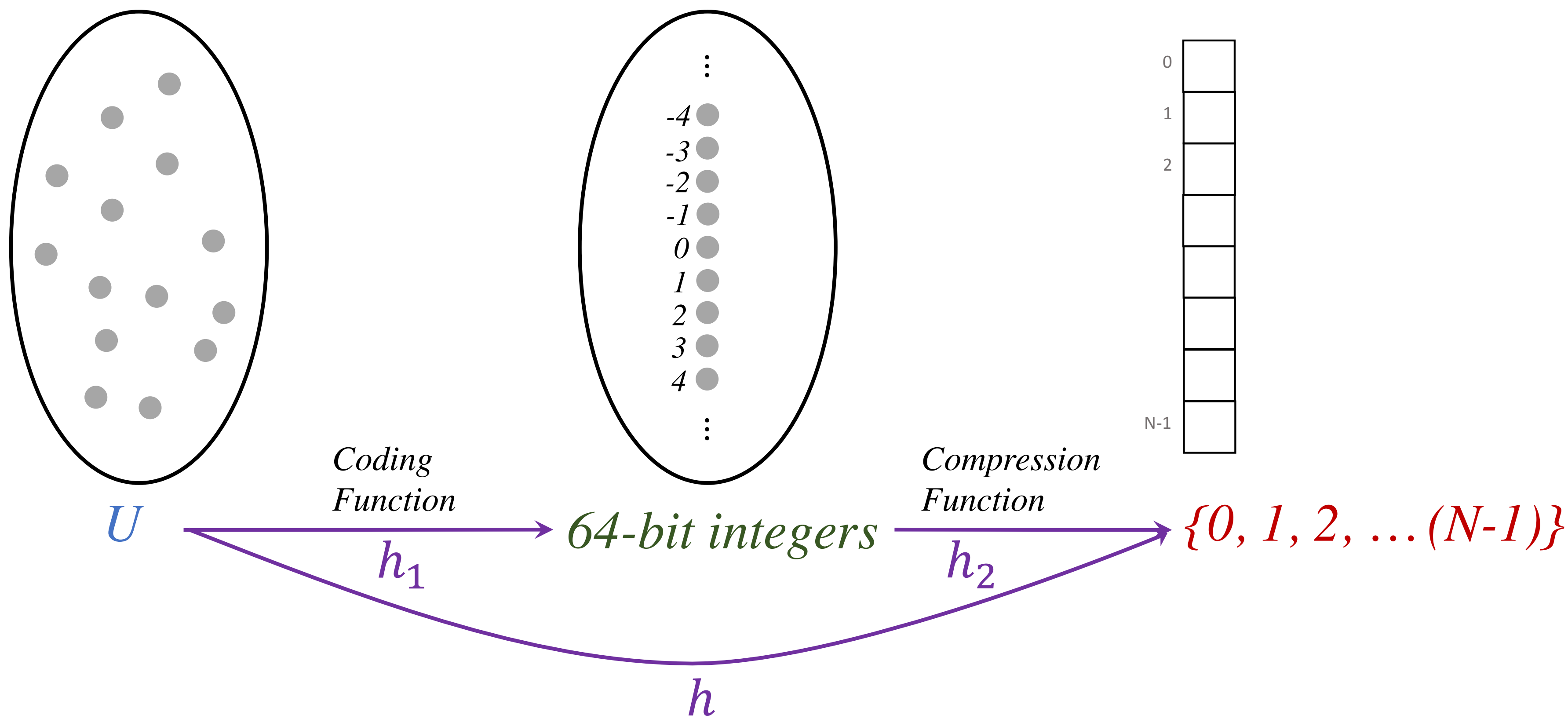
Hash Functions



Hash Functions

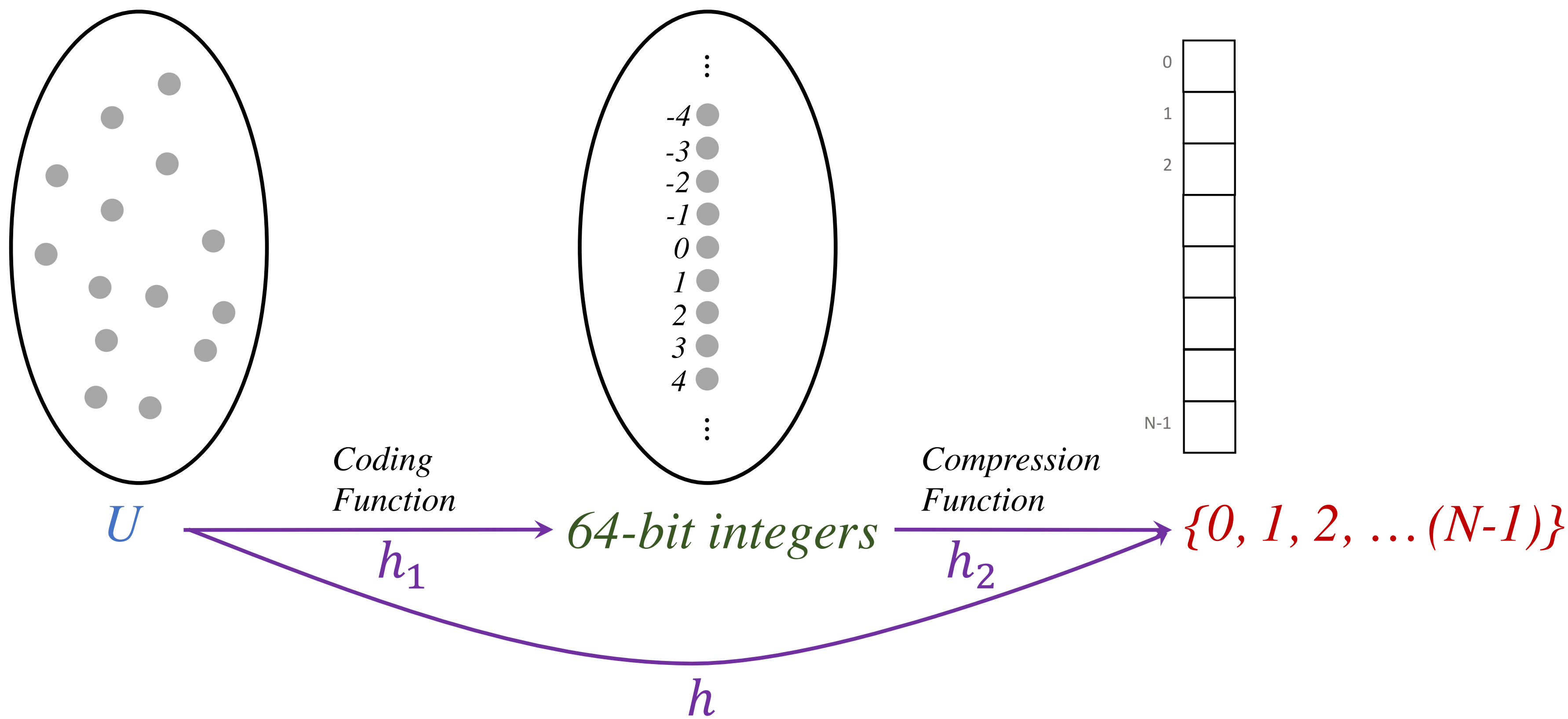


Hash Functions



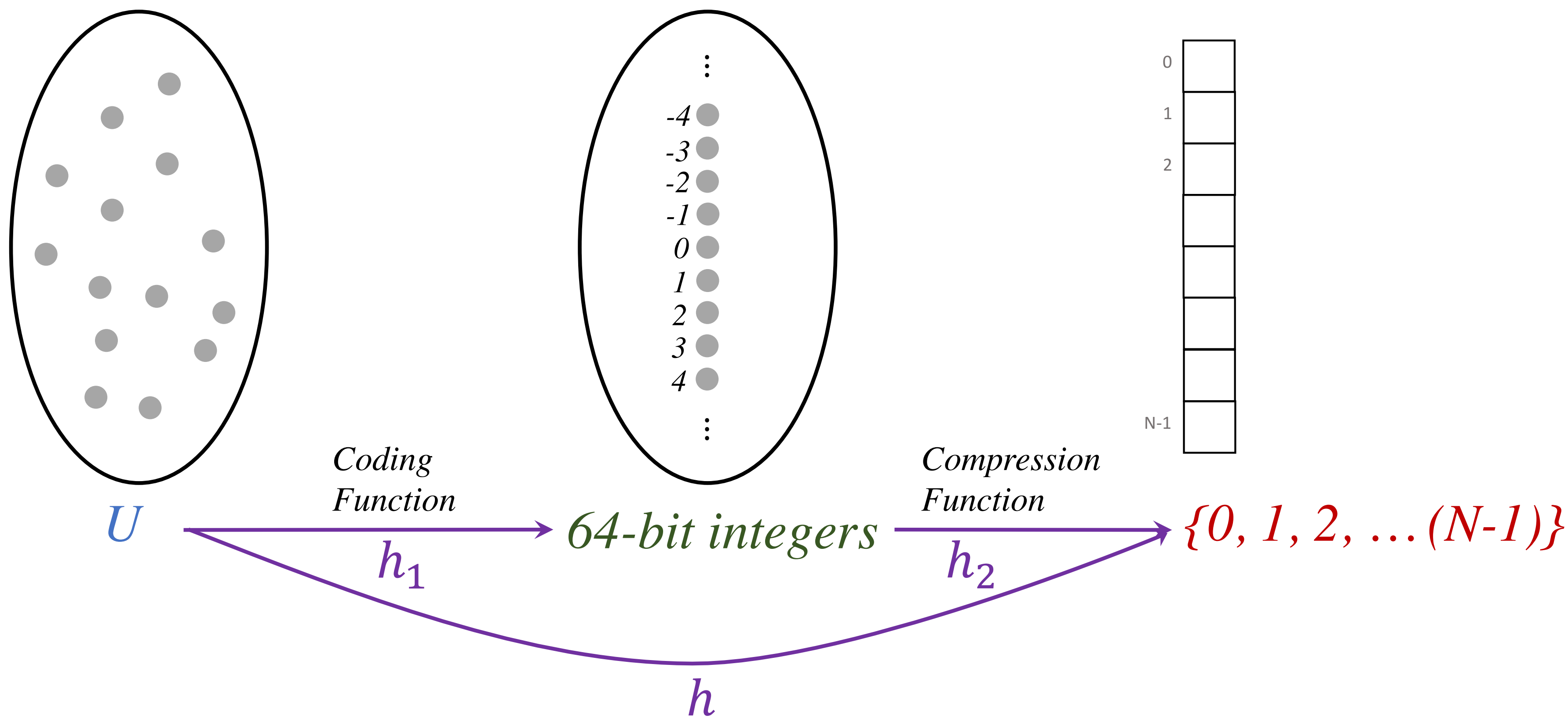
$h_1(k)$

Hash Functions



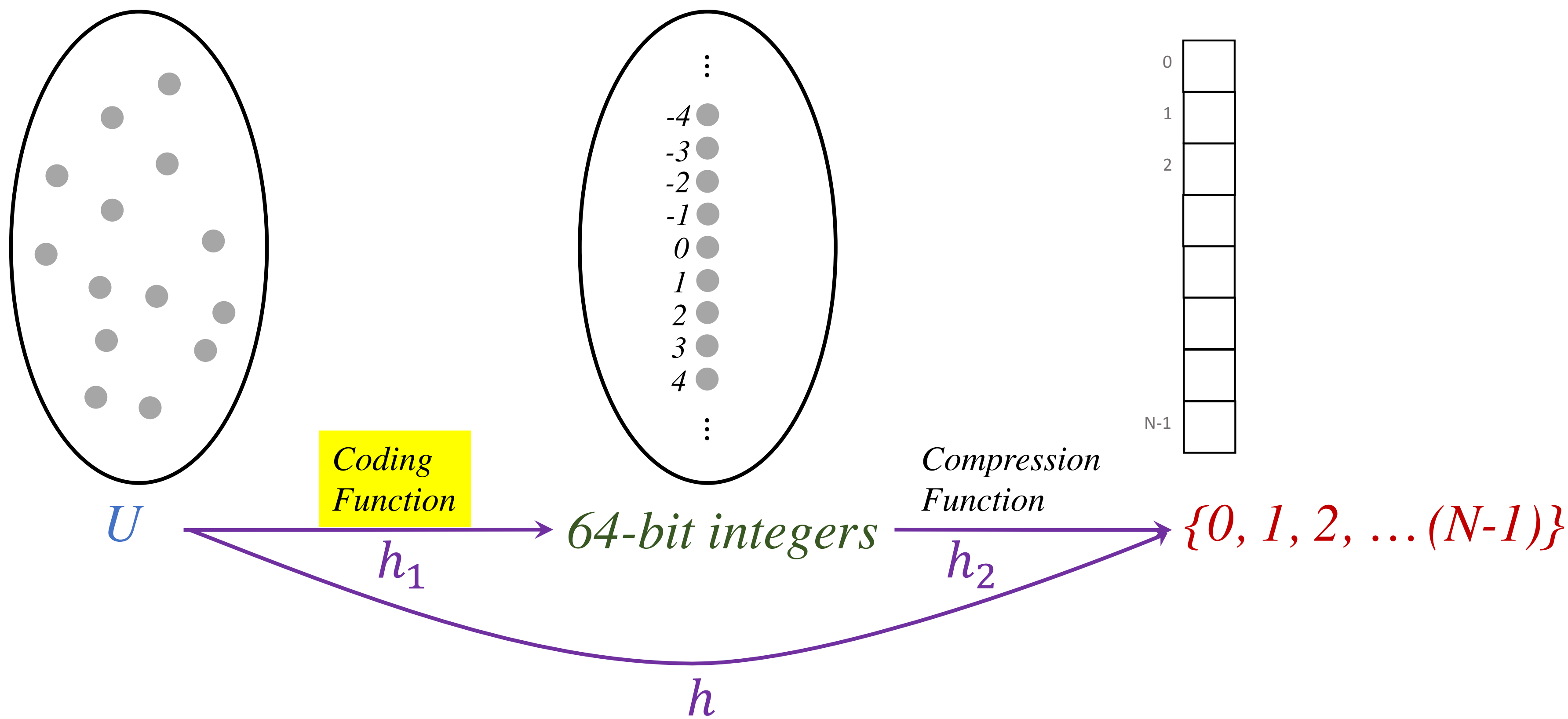
$$h_2(h_1(k))$$

Hash Functions



$$h(k) = h_2(h_1(k))$$

Hash Functions



$$h(k) = h_2(h_1(k))$$

Coding Functions

$h_1 : U \rightarrow (64\text{–}bit\ integers)$

Coding Functions

$h_1 : U \rightarrow (64\text{-}bit\ integers)$

i. <u>Common Approaches</u>:

Coding Functions

$h_1 : U \rightarrow (64\text{–}bit\ integers)$

i. <u>Common Approaches</u>:
   - Integer Casting

Coding Functions

$h_1 : U \rightarrow (64\text{–}bit\ integers)$

i. Common Approaches:

- Integer Casting

  Look at the binary representation of key,

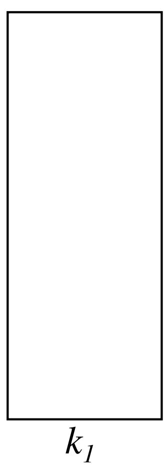
Coding Functions

$h_1 : U \rightarrow (64\text{–}bit\ integers)$

i. <u>Common Approaches</u>:

- Integer Casting

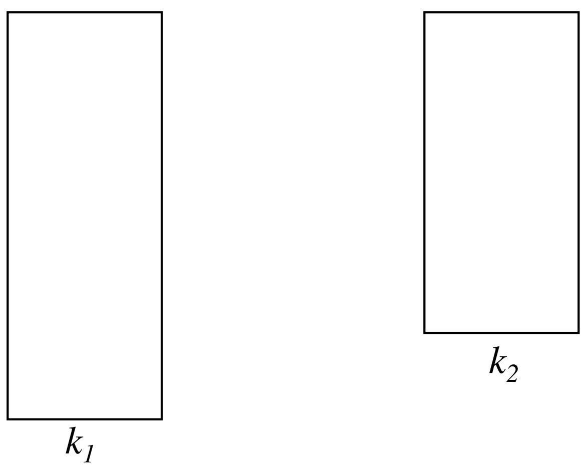
  Look at the binary representation of *key*,



Coding Functions

$h_1 : U \rightarrow (64\text{–}bit\ integers)$

i. <u>Common Approaches</u>:
- Integer Casting
  Look at the binary representation of *key*,

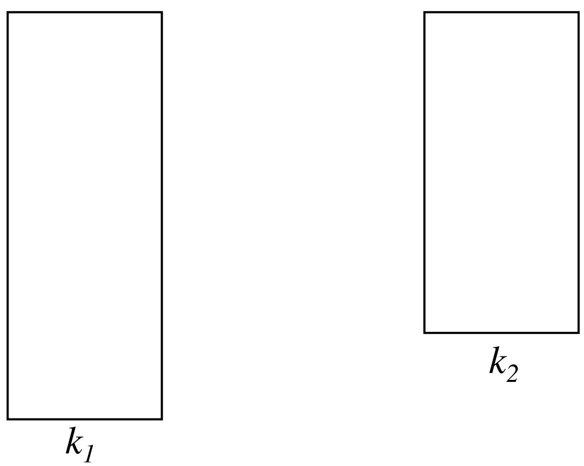


Coding Functions

$h_1 : U \rightarrow (64\text{–}bit\ integers)$

i. <u>Common Approaches</u>:

- Integer Casting
  Look at the binary representation of key,
  Take the 8 least significant bytes, and interpret it as a 64-bit 2's complement number

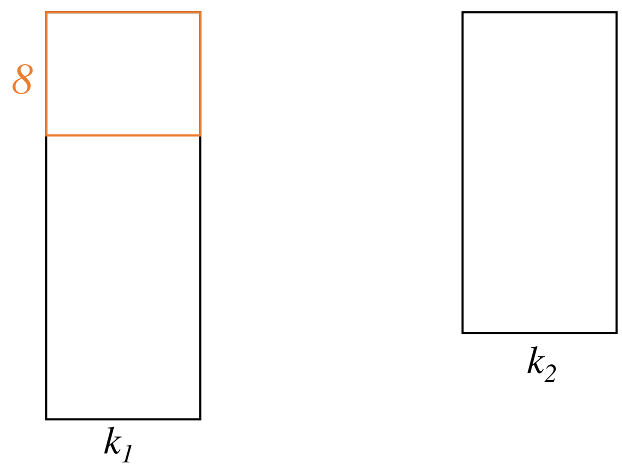


Coding Functions

$h_1 : U \rightarrow (64\text{–}bit\ integers)$

i. <u>Common Approaches</u>:

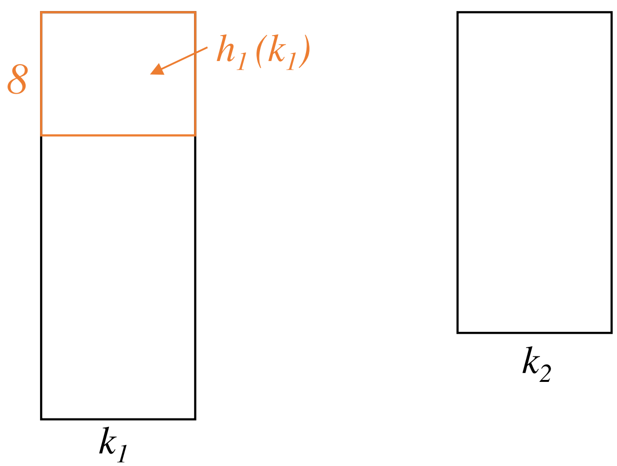
- Integer Casting
  Look at the binary representation of key,
  Take the 8 least significant bytes, and interpret it as a 64-bit 2's complement number



Coding Functions

$h_1 : U \rightarrow (64\text{−}bit\ integers)$

i. <u>Common Approaches</u>:
   - Integer Casting
     Look at the binary representation of key,
     Take the 8 least significant bytes, and interpret it as a 64-bit 2's complement number

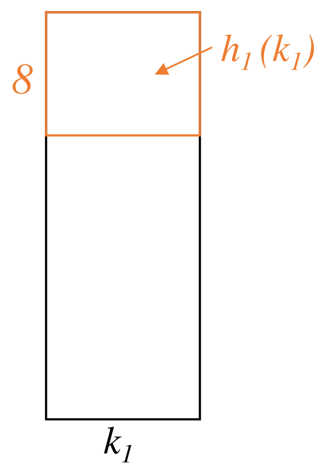


Coding Functions

$h_1 : U \rightarrow (64\text{–}bit\ integers)$

i. <u>Common Approaches</u>:

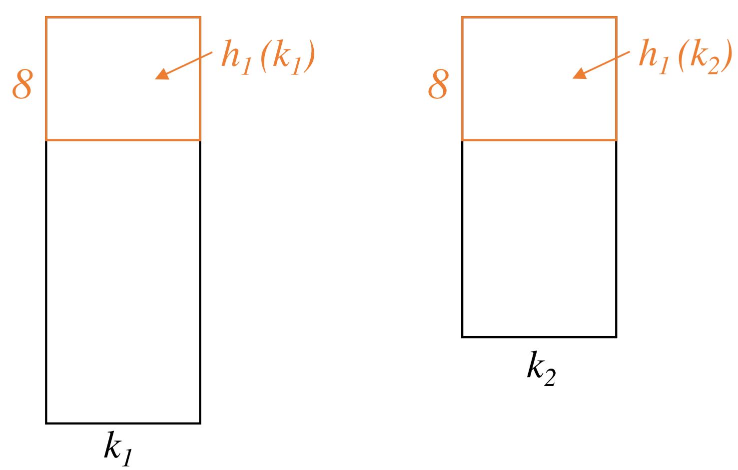
- Integer Casting
  Look at the binary representation of key,
  Take the 8 least significant bytes, and interpret it as a 64-bit 2's complement number



Coding Functions

$h_1: U \rightarrow (64\text{-}bit\ integers)$

i. <u>Common Approaches</u>:
- Integer Casting
  Look at the binary representation of key,
  Take the 8 least significant bytes, and interpret it as a 64-bit 2's complement number

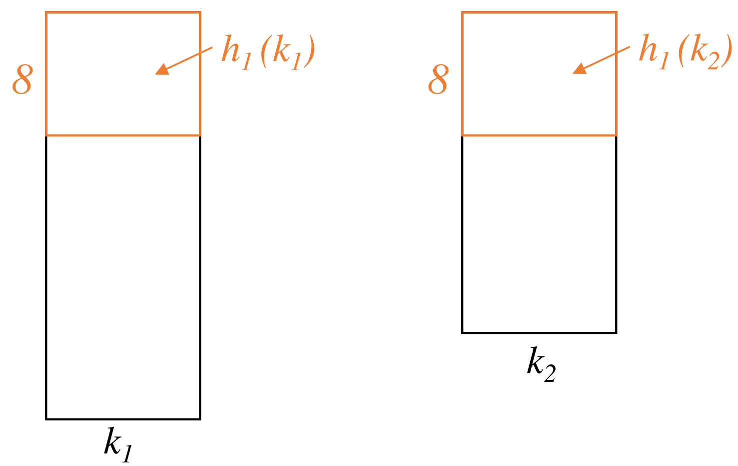


Coding Functions

$h_1 : U \rightarrow (64\text{–}bit\ integers)$

i. <u>Common Approaches</u>:

- Integer Casting
  Look at the binary representation of key,
  Take the 8 least significant bytes, and interpret it as a 64-bit 2's complement number



<u>Problem</u>:
This approach ignores part of the data.
In a biased set of keys, these parts could be where the keys differ

Coding Functions

$h_1 : U \rightarrow (64\text{–}bit\ integers)$

i. Common Approaches:
   - Integer Casting

Coding Functions

$h_1 : U \rightarrow (64\text{−}bit\ integers)$

i. <u>Common Approaches</u>:
   - Integer Casting

   - Component Sum

Coding Functions

$h_1 : U \rightarrow \text{(64−bit integers)}$

i. <u>Common Approaches</u>:
- Integer Casting

- Component Sum
  Break key to its components: $key = (k_0, k_1, k_2, \ldots, k_{m-1})$.

Coding Functions

$h_1 : U \rightarrow (64\text{–}bit\ integers)$

i. Common Approaches:
- Integer Casting

- Component Sum
  Break key to its components: $key = (k_0, k_1, k_2, \dots, k_{m-1})$.
  The coding function h_1 would add all the components of key.

Coding Functions

$h_1 : U \rightarrow (64\text{—}bit\ integers)$

i. <u>Common Approaches</u>:

- Integer Casting

- Component Sum
  Break key to its components: $key = (k_0, k_1, k_2, \dots, k_{m-1})$.
  The coding function h_1 would add all the components of key.
  That is: $h_1(key) = k_0 + k_1 + \ \dots + k_{m-1}$

Coding Functions

$h_1: U \rightarrow (64\text{–}bit\ integers)$

i. Common Approaches:

- Integer Casting

- Component Sum
  Break key to its components: $key = (k_0, k_1, k_2, \ldots, k_{m-1})$.
  The coding function h_1 would add all the components of key.
  That is: $h_1(key) = k_0 + k_1 + \ \ldots + k_{m-1}$

"stop"

Coding Functions

$h_1: U \rightarrow (64\!-\!bit\ integers)$

i. <u>Common Approaches</u>:

- Integer Casting

- Component Sum
  Break key to its components: $key = (k_0, k_1, k_2, \ldots, k_{m-1})$.
  The coding function h_1 would add all the components of key.
  That is: $h_1(key) = k_0 + k_1 + \ \ldots + k_{m-1}$

"stop"
"tops"

Coding Functions

$h_1 : U \rightarrow (64\text{–}bit\ integers)$

i. Common Approaches:

- Integer Casting

- Component Sum
  Break key to its components: $key = (k_0, k_1, k_2, \dots, k_{m-1})$.
  The coding function h_1 would add all the components of key.
  That is: $h_1(key) = k_0 + k_1 + \ \dots + k_{m-1}$

"stop"
"tops"
"pots"

Coding Functions

$h_1 : U \rightarrow (64\text{−}bit\ integers)$

i. Common Approaches:

- Integer Casting

- Component Sum
  Break key to its components: $key = (k_0, k_1, k_2, \dots, k_{m-1})$.
  The coding function h_1 would add all the components of key.
  That is: $h_1(key) = k_0 + k_1 + \ \dots + k_{m-1}$

"stop"
"tops"
"pots"
"spot"

Coding Functions

$h_1 : U \rightarrow (64\text{—}bit\ integers)$

i. <u>Common Approaches</u>:

- Integer Casting

- Component Sum
  Break key to its components: $key = (k_0, k_1, k_2, \dots, k_{m-1})$.
  The coding function h_1 would add all the components of key.
  That is: $h_1(key) = k_0 + k_1 + \ \dots + k_{m-1}$

"stop"
"tops"
"pots"
"spot"
"post"

Coding Functions

$h_1: U \rightarrow (64\text{−}bit\ integers)$

i. <u>Common Approaches</u>:

- Integer Casting

- Component Sum
  Break key to its components: $key = (k_0, k_1, k_2, \dots, k_{m-1})$.
  The coding function h_1 would add all the components of key.
  That is: $h_1(key) = k_0 + k_1 + \ \dots + k_{m-1}$

"stop"
"tops"
"pots"
"spot"
"post"

all coded to the same value

Coding Functions

$h_1: U \rightarrow (64\text{-}bit\ integers)$

i. Common Approaches:

- Integer Casting

- Component Sum
  Break key to its components: $key = (k_0, k_1, k_2, \dots, k_{m-1})$.
  The coding function h_1 would add all the components of key.
  That is: $h_1(key) = k_0 + k_1 + \dots + k_{m-1}$

“stop”
“tops”
“pots”
“spot”
“post”

all coded to the same value

Problem:
This approach doesn't take the positions of the components into account

Coding Functions

$h_1 : U \rightarrow (64\text{−}bit\ integers)$

i. <u>Common Approaches</u>:
  - Integer Casting
  - Component Sum

Coding Functions

$h_1 : U \rightarrow (64\text{−}bit\ integers)$

i. <u>Common Approaches</u>:

- Integer Casting
- Component Sum
- Polynomial Accumulation

Coding Functions

$h_1 : U \rightarrow (64\text{–}bit\ integers)$

i. Common Approaches:
- Integer Casting
- Component Sum
- Polynomial Accumulation
  Let z be an integer ≥ 2

Coding Functions

$h_1 : U \rightarrow (64\text{–}bit\ integers)$

i. Common Approaches:

- Integer Casting

- Component Sum

- Polynomial Accumulation
  Let z be an integer ≥ 2
  To code key,

Coding Functions

$h_1 : U \rightarrow (64\text{−}bit\ integers)$

i. Common Approaches:
   - Integer Casting

   - Component Sum

   - Polynomial Accumulation
     Let z be an integer ≥ 2
     To code key, break it to its components: $key = (k_0, k_1, k_2, \ldots, k_{m-1})$

Coding Functions

$h_1 : U \rightarrow (64\text{−}bit\ integers)$

i. <u>Common Approaches</u>:
   - Integer Casting

   - Component Sum

   - Polynomial Accumulation
     Let z be an integer ≥ 2
     To code key, break it to its components: $key = (k_0, k_1, k_2, \ldots, k_{m-1})$
     We define: $h_1(key) =$

Coding Functions

$h_1 : U \rightarrow (64\text{−}bit\ integers)$

i. <u>Common Approaches</u>:

- Integer Casting
- Component Sum
- Polynomial Accumulation
  Let z be an integer ≥ 2
  To code key, break it to its components: $key = (k_0, k_1, k_2, \ldots, k_{m-1})$
  We define: $h_1(key) = k_0 \cdot z^0 + k_1 \cdot z^1 + \ldots + k_{m-1} \cdot z^{m-1}$

Coding Functions

$h_1 : U \rightarrow (64\text{−}bit\ integers)$

i. Common Approaches:

- Integer Casting

- Component Sum

- Polynomial Accumulation
  Let z be an integer ≥ 2
  To code key, break it to its components: $key = (k_0, k_1, k_2, \ldots, k_{m-1})$
  We define: $h_1(key) = k_0 \cdot z^0 + k_1 \cdot z^1 + \ldots + k_{m-1} \cdot z^{m-1}$

  Fun Fact:

Coding Functions

$h_1 : U \rightarrow (64\text{−}bit\ integers)$

i. Common Approaches:
- Integer Casting

- Component Sum

- Polynomial Accumulation
  Let z be an integer ≥ 2
  To code key, break it to its components: $key = (k_0, k_1, k_2, \ldots, k_{m-1})$
  We define: $h_1(key) = k_0 \cdot z^0 + k_1 \cdot z^1 + \ \ldots + k_{m-1} \cdot z^{m-1}$

  Fun Fact: If we take $z = 33$

Coding Functions

$h_1 : U \rightarrow (64\text{−}bit\ integers)$

i. Common Approaches:

- Integer Casting

- Component Sum

- Polynomial Accumulation
  Let z be an integer ≥ 2
  To code key, break it to its components: $key = (k_0, k_1, k_2, \dots, k_{m-1})$
  We define: $h_1(key) = k_0 \cdot z^0 + k_1 \cdot z^1 + \dots + k_{m-1} \cdot z^{m-1}$

  Fun Fact: If we take $z = 33$, When coding 50,000 English words, have at most 6 collisions

Coding Functions

$h_1 : U \rightarrow (64\text{−}bit\ integers)$

i. <u>Common Approaches</u>:
- Integer Casting
- Component Sum
- Polynomial Accumulation

Coding Functions

$h_1: U \rightarrow (64\text{−}bit\ integers)$

i. <u>Common Approaches</u>:
   - Integer Casting

   - Component Sum

   - Polynomial Accumulation

ii. <u>Python's Built-in coding function</u>:

Coding Functions

$h_1 : U \rightarrow (64\text{−}bit\ integers)$

i. Common Approaches:
   - Integer Casting

   - Component Sum

   - Polynomial Accumulation

ii. Python's Built-in coding function: `hash(key)`

Coding Functions

$h_1 : U \rightarrow (64\text{−}bit\ integers)$

i. <u>Common Approaches</u>:
   - Integer Casting
   - Component Sum
   - Polynomial Accumulation

ii. <u>Python's Built-in coding function</u>: `hash(key)`
   - Can be called with built-in **immutable** types (`int`, `str`, `float`, `tuple`)

Coding Functions

$h_1 : U \rightarrow (64\text{−}bit\ integers)$

i. Common Approaches:
- Integer Casting
- Component Sum
- Polynomial Accumulation

ii. Python's Built-in coding function: `hash(key)`
- Can be called with built-in **immutable** types (`int`, `str`, `float`, `tuple`)
- User defined classes are unhashable, unless they overload the `__hash__` method

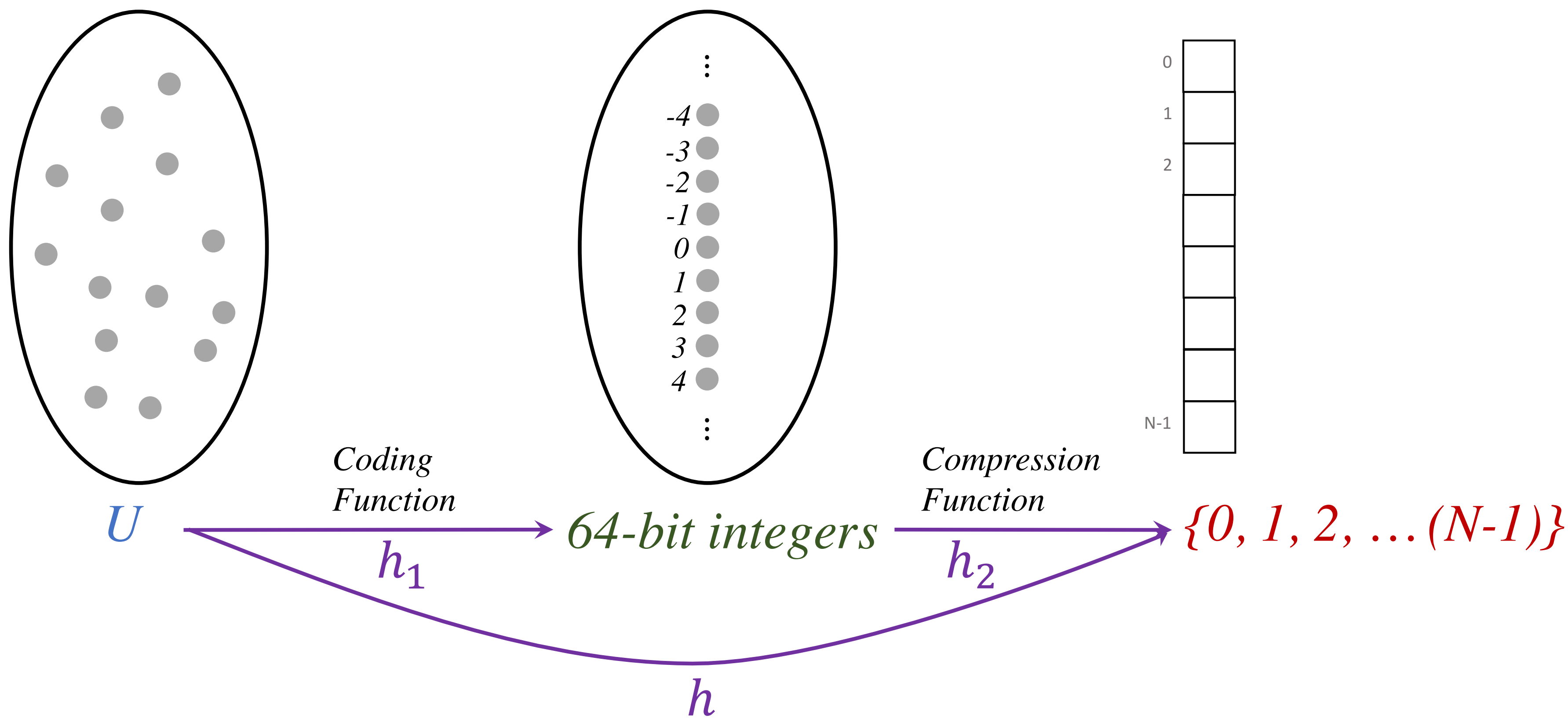
Coding Functions

$h_1 : U \rightarrow \textit{(64–bit integers)}$

i. Common Approaches:
- Integer Casting
- Component Sum
- Polynomial Accumulation

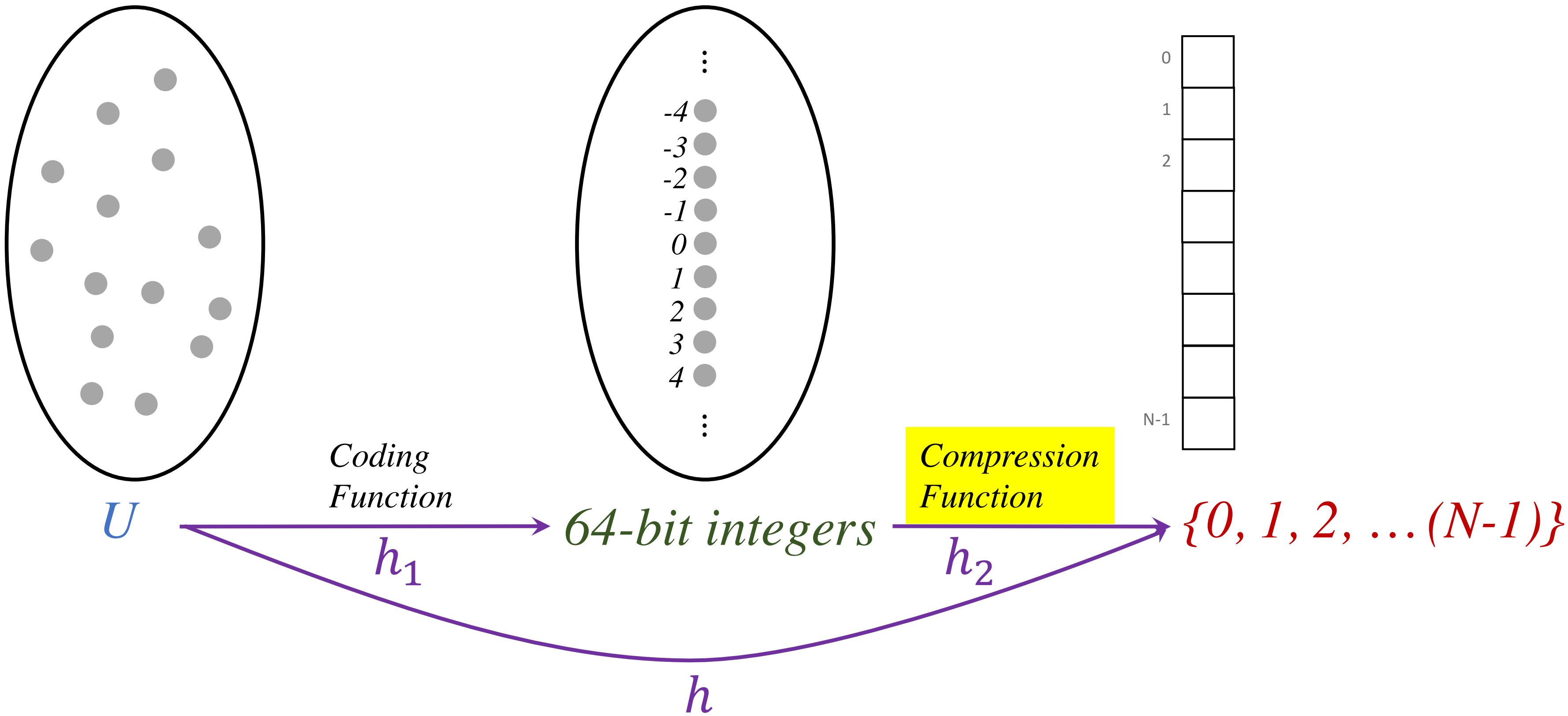
ii. Python’s Built-in coding function: `hash(key)`
- Can be called with built-in **immutable** types (`int`, `str`, `float`, `tuple`)
- User defined classes are unhashable, unless they overload the `__hash__` method
- Make sure that $(x = y) \rightarrow hash(x) = hash(y)$

Hash Functions



$$h(k) = h_2\big(h_1(k)\big)$$

Hash Functions



$$h(k) = h_2(h_1(k))$$

Compression Functions

h_2: $(64\text{−}bit\ integers) \rightarrow \{0, 1, \ldots, N - 1\}$

Compression Functions

h_2: $(64\text{−}bit\ integers) \rightarrow \{0, 1, \ldots, N - 1\}$

Common Approaches:

Compression Functions

h_2: *(64−bit integers)* $\rightarrow \{0, 1, \ldots, N-1\}$

Common Approaches:

- The Division Method

Compression Functions

h_2: *(64−bit integers)* $\rightarrow \{0, 1, \ldots, N-1\}$

<u>Common Approaches</u>:

- The Division Method

  We define: $h_2(k) =$

Compression Functions

h_2: *(64−bit integers)* $\rightarrow \{0, 1, \ldots, N-1\}$

<u>Common Approaches</u>:

- The Division Method

  We define: $h_2(k) = k \bmod N$

Compression Functions

h_2: *(64−bit integers)* $\rightarrow \{0, 1, \ldots, N-1\}$

Common Approaches:

- The Division Method

  We define: $h_2(k) = k \bmod N$

  - If keys are chosen randomly → satisfies the "Uniform Hashing" property

Compression Functions

h_2: $(64\text{−}bit\ integers) \rightarrow \{0, 1, \ldots, N-1\}$

<u>Common Approaches</u>:

- The Division Method

  We define: $h_2(k) = k \bmod N$

  - If keys are chosen randomly → satisfies the "Uniform Hashing" property
  - Long chains might be created, if keys are biased

Compression Functions

h_2: $(64\text{−}bit\ integers) \rightarrow \{0, 1, \ldots, N-1\}$

Common Approaches:

- The Division Method

  We define: $h_2(k) = k\ mod\ N$

  - If keys are chosen randomly → satisfies the "Uniform Hashing" property
  - Long chains might be created, if keys are biased

  Example: if our keys are: {0, 5, 10, 15, 20, 30, 35, 40, 45, 50, 55}:

Compression Functions

h_2: *(64−bit integers)* $\rightarrow \{0, 1, \ldots, N-1\}$

<u>Common Approaches</u>:

- The Division Method

  We define: $h_2(k) = k \ mod \ N$
  - If keys are chosen randomly → satisfies the "Uniform Hashing" property
  - Long chains might be created, if keys are biased

  Example: if our keys are: {0, 5, 10, 15, 20, 30, 35, 40, 45, 50, 55}:

  For N=10:

Compression Functions

h_2: $(64\text{–}bit\ integers) \rightarrow \{0, 1, \dots, N-1\}$

<u>Common Approaches</u>:

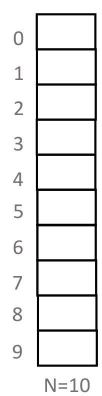
- The Division Method

  We define: $h_2(k) = k \bmod N$

  - If keys are chosen randomly → satisfies the "Uniform Hashing" property
  - Long chains might be created, if keys are biased

Example: if our keys are: {0, 5, 10, 15, 20, 30, 35, 40, 45, 50, 55}:

For N=10:



Compression Functions

h_2: *(64−bit integers)* $\rightarrow \{0, 1, \ldots, N-1\}$

<u>Common Approaches</u>:

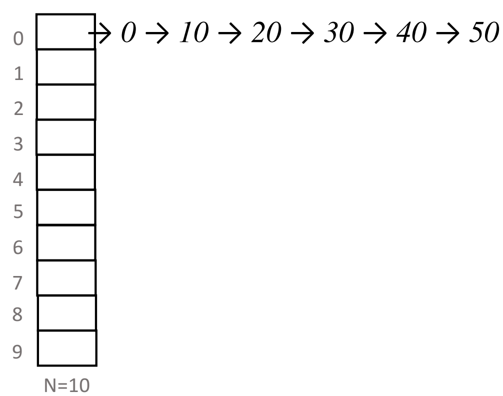
- The Division Method

  We define: $h_2(k) = k \bmod N$

  - If keys are chosen randomly → satisfies the "Uniform Hashing" property
  - Long chains might be created, if keys are biased

Example: if our keys are: {0, 5, 10, 15, 20, 30, 35, 40, 45, 50, 55}:

For N=10:



Compression Functions

h_2: *(64−bit integers)* $\rightarrow \{0, 1, \ldots, N-1\}$

<u>Common Approaches</u>:

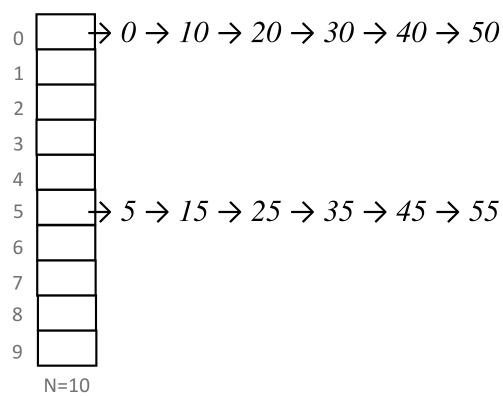
- The Division Method

  We define: $h_2(k) = k \bmod N$

  - If keys are chosen randomly → satisfies the "Uniform Hashing" property
  - Long chains might be created, if keys are biased

Example: if our keys are: {0, 5, 10, 15, 20, 30, 35, 40, 45, 50, 55}:

For N=10:



Compression Functions

h_2: (64−bit integers) $\rightarrow \{0, 1, \ldots, N-1\}$

<u>Common Approaches</u>:

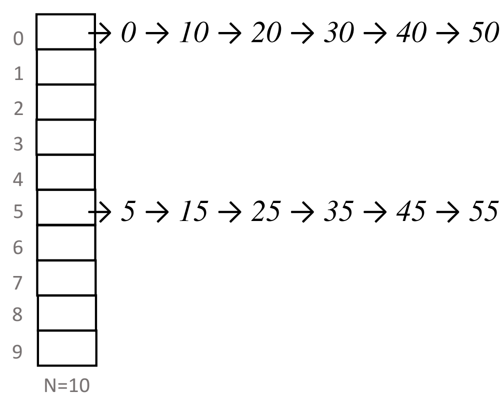
- The Division Method

  We define: $h_2(k) = k \bmod N$

  - If keys are chosen randomly → satisfies the "Uniform Hashing" property
  - Long chains might be created, if keys are biased

Example: if our keys are: {0, 5, 10, 15, 20, 30, 35, 40, 45, 50, 55}:

For N=10:



A common heuristic is to choose N to be prime.

Compression Functions

h_2: $(64\text{−}bit\ integers) \rightarrow \{0, 1, \ldots, N-1\}$

<u>Common Approaches</u>:

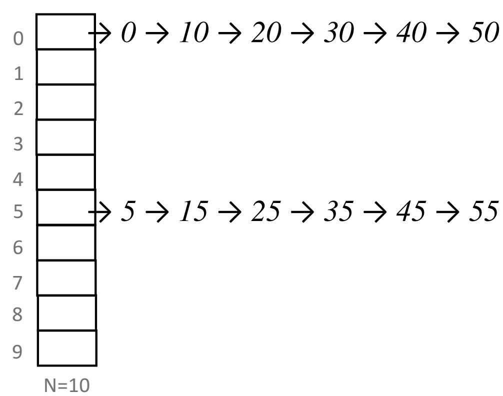
- The Division Method

  We define: $h_2(k) = k \bmod N$

  - If keys are chosen randomly → satisfies the "Uniform Hashing" property
  - Long chains might be created, if keys are biased

Example: if our keys are: {0, 5, 10, 15, 20, 30, 35, 40, 45, 50, 55}:

For N=10:



A common heuristic is to choose N to be prime.

For N=7:

Compression Functions

h_2: *(64−bit integers)* $\rightarrow \{0, 1, \dots, N-1\}$

<u>Common Approaches</u>:

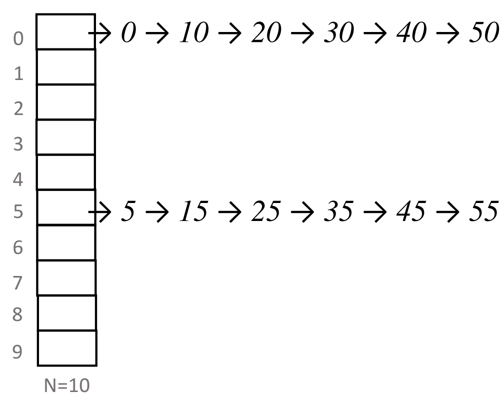
- The Division Method

  We define: $h_2(k) = k \bmod N$

  - If keys are chosen randomly → satisfies the "Uniform Hashing" property
  - Long chains might be created, if keys are biased

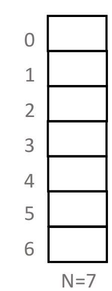
Example: if our keys are: {0, 5, 10, 15, 20, 30, 35, 40, 45, 50, 55}:

For N=10:



A common heuristic is to choose N to be prime.

For N=7:



Compression Functions

h_2: $(64\text{−}bit\ integers) \rightarrow \{0, 1, \ldots, N-1\}$

<u>Common Approaches</u>:

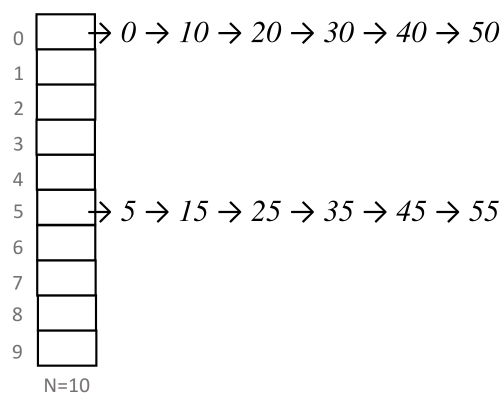
- The Division Method

  We define: $h_2(k) = k \bmod N$

  - If keys are chosen randomly → satisfies the "Uniform Hashing" property
  - Long chains might be created, if keys are biased

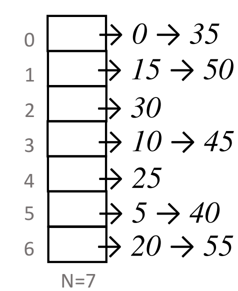
Example: if our keys are: {0, 5, 10, 15, 20, 30, 35, 40, 45, 50, 55}:

For N=10:



A common heuristic is to choose N to be prime.

For N=7:



Compression Functions

h_2: $(64\text{-}bit\ integers) \rightarrow \{0, 1, \ldots, N-1\}$

Common Approaches:

- The Division Method

Compression Functions

h_2: $(64\text{–}bit\ integers) \to \{0, 1, \dots, N-1\}$

Common Approaches:

- The Division Method
- The Multiply-Add-and-Divide (MAD) Method

Compression Functions

h_2: $(64\text{−}bit\ integers) \rightarrow \{0, 1, \ldots, N-1\}$

Common Approaches:

- The Division Method
- The Multiply-Add-and-Divide (MAD) Method

  Let p be a prime number, $p > |U|$

Compression Functions

h_2: $(64\text{−}bit\ integers) \to \{0, 1, \ldots, N-1\}$

Common Approaches:

- The Division Method
- The Multiply-Add-and-Divide (MAD) Method

  Let p be a prime number, $p>|U|$

  Let a be a random number from $\{1, 2, \ldots, p\text{-}1\}$

Compression Functions

h_2: $(64\text{−}bit\ integers) \rightarrow \{0, 1, \ldots, N-1\}$

Common Approaches:

- The Division Method
- The Multiply-Add-and-Divide (MAD) Method

  Let p be a prime number, $p>|U|$

  Let a be a random number from $\{1, 2, \ldots, p\text{-}1\}$

  Let b be a random number from $\{0, 1, 2, \ldots, p\text{-}1\}$

Compression Functions

h_2: *(64−bit integers)* $\rightarrow \{0, 1, \ldots, N-1\}$

<u>Common Approaches</u>:

- The Division Method
- The Multiply-Add-and-Divide (MAD) Method

  Let p be a prime number, $p>|U|$

  Let a be a random number from $\{1, 2, \ldots, p\text{-}1\}$

  Let b be a random number from $\{0, 1, 2, \ldots, p\text{-}1\}$

  <u>We define</u>: $h_2(k) =$

Compression Functions

h_2: *(64−bit integers)* $\rightarrow \{0, 1, \ldots, N-1\}$

<u>Common Approaches</u>:

- The Division Method
- The Multiply-Add-and-Divide (MAD) Method

  Let p be a prime number, $p>|U|$

  Let a be a random number from $\{1, 2, \ldots, p\text{-}1\}$

  Let b be a random number from $\{0, 1, 2, \ldots, p\text{-}1\}$

  <u>We define</u>: $h_2(k) = [\quad k \qquad\qquad\qquad]\ mod\ N$

Compression Functions

h_2: *(64−bit integers)* $\rightarrow \{0, 1, \ldots, N-1\}$

<u>Common Approaches</u>:

- The Division Method
- The Multiply-Add-and-Divide (MAD) Method

  Let p be a prime number, $p > |U|$

  Let a be a random number from $\{1, 2, \ldots, p\text{-}1\}$

  Let b be a random number from $\{0, 1, 2, \ldots, p\text{-}1\}$

  <u>We define</u>: $h_2(k) = [\ ak \qquad]\ mod\ N$

Compression Functions

h_2: $(64\text{–}bit\ integers) \rightarrow \{0, 1, \dots, N-1\}$

Common Approaches:

- The Division Method
- The Multiply-Add-and-Divide (MAD) Method

  Let p be a prime number, $p>|U|$

  Let a be a random number from $\{1, 2, \dots, p\text{-}1\}$

  Let b be a random number from $\{0, 1, 2, \dots, p\text{-}1\}$

  We define: $h_2(k) = [\ ak + b\qquad\qquad]\ mod\ N$

Compression Functions

h_2: *(64−bit integers)* $\rightarrow \{0, 1, \ldots, N-1\}$

Common Approaches:

- The Division Method
- The Multiply-Add-and-Divide (MAD) Method

  Let p be a prime number, $p > |U|$

  Let a be a random number from $\{1, 2, \ldots, p\text{-}1\}$

  Let b be a random number from $\{0, 1, 2, \ldots, p\text{-}1\}$

  We define: $h_2(k) = [(ak + b)\ mod\ p]\ mod\ N$

Compression Functions

h_2: *(64−bit integers)* $\rightarrow \{0, 1, \dots, N-1\}$

Common Approaches:

- The Division Method
- The Multiply-Add-and-Divide (MAD) Method

  Let p be a prime number, $p>|U|$

  Let a be a random number from $\{1, 2, \dots, p\text{-}1\}$

  Let b be a random number from $\{0, 1, 2, \dots, p\text{-}1\}$

  We define: $h_2(k) = [(ak + b)\ mod\ p]\ mod\ N$

Example: if $|U|=60$

Compression Functions

h_2: *(64−bit integers)* $\rightarrow \{0, 1, \ldots, N-1\}$

Common Approaches:

- The Division Method
- The Multiply-Add-and-Divide (MAD) Method

  Let p be a prime number, $p>|U|$

  Let a be a random number from $\{1, 2, \ldots, p\text{-}1\}$

  Let b be a random number from $\{0, 1, 2, \ldots, p\text{-}1\}$

  We define: $h_2(k) = [(ak + b) \; mod \; p] \; mod \; N$

Example: if $|U|=60$

the keys are:

Compression Functions

h_2: $(64\text{–}bit\ integers) \rightarrow \{0, 1, \ldots, N-1\}$

Common Approaches:

- The Division Method
- The Multiply-Add-and-Divide (MAD) Method

  Let p be a prime number, $p>|U|$

  Let a be a random number from $\{1, 2, \ldots, p\text{-}1\}$

  Let b be a random number from $\{0, 1, 2, \ldots, p\text{-}1\}$

  We define: $h_2(k) = [(ak + b)\ mod\ p]\ mod\ N$

Example: if $|U|=60$

the keys are: {0, 5, 10, 15, 20, 30, 35, 40, 45, 50, 55}

Compression Functions

h_2: *(64–bit integers)* $\rightarrow \{0, 1, \dots, N-1\}$

Common Approaches:

- The Division Method
- The Multiply-Add-and-Divide (MAD) Method

  Let p be a prime number, $p>|U|$

  Let a be a random number from $\{1, 2, \dots, p\text{-}1\}$

  Let b be a random number from $\{0, 1, 2, \dots, p\text{-}1\}$

  We define: $h_2(k) = [(ak + b)\ mod\ p]\ mod\ N$

Example: if $|U|=60$

the keys are: {0, 5, 10, 15, 20, 30, 35, 40, 45, 50, 55}

we choose:

Compression Functions

h_2: $(64\text{−}bit\ integers) \rightarrow \{0, 1, \ldots, N-1\}$

Common Approaches:

- The Division Method
- The Multiply-Add-and-Divide (MAD) Method

  Let p be a prime number, $p>|U|$

  Let a be a random number from $\{1, 2, \ldots, p\text{-}1\}$

  Let b be a random number from $\{0, 1, 2, \ldots, p\text{-}1\}$

  We define: $h_2(k) = [(ak + b)\ mod\ p]\ mod\ N$

Example: if $|U|=60$

the keys are: {0, 5, 10, 15, 20, 30, 35, 40, 45, 50, 55}

we choose: $p=101$ $a=30$ $b=6$

Compression Functions

h_2: $(64\text{−}bit\ integers) \rightarrow \{0, 1, \ldots, N-1\}$

Common Approaches:

- The Division Method
- The Multiply-Add-and-Divide (MAD) Method

  Let p be a prime number, $p>|U|$

  Let a be a random number from $\{1, 2, \ldots, p\text{-}1\}$

  Let b be a random number from $\{0, 1, 2, \ldots, p\text{-}1\}$

  We define: $h_2(k) = [(ak + b)\ mod\ p]\ mod\ N$

Example: if $|U|=60$

the keys are: {0, 5, 10, 15, 20, 30, 35, 40, 45, 50, 55}

we choose: $p=101$ $a=30$ $b=6$

For a table of size N=10:

Compression Functions

h_2: *(64−bit integers)* $\rightarrow \{0, 1, \dots, N-1\}$

<u>Common Approaches</u>:

- The Division Method
- The Multiply-Add-and-Divide (MAD) Method

  Let p be a prime number, $p>|U|$

  Let a be a random number from $\{1, 2, \dots, p\text{-}1\}$

  Let b be a random number from $\{0, 1, 2, \dots, p\text{-}1\}$

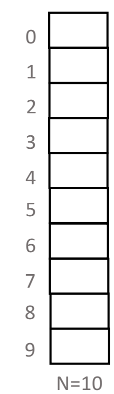
  <u>We define</u>: $h_2(k) = [(ak + b)\ mod\ p]\ mod\ N$

Example: if $|U|=60$

the keys are: {0, 5, 10, 15, 20, 30, 35, 40, 45, 50, 55}

we choose: $p=101 \quad a=30 \quad b=6$

For a table of size N=10:



Compression Functions

h_2: *(64−bit integers)* $\rightarrow \{0, 1, \ldots, N-1\}$

Common Approaches:

- The Division Method
- The Multiply-Add-and-Divide (MAD) Method

  Let p be a prime number, $p > |U|$

  Let a be a random number from $\{1, 2, \ldots, p\text{-}1\}$

  Let b be a random number from $\{0, 1, 2, \ldots, p\text{-}1\}$

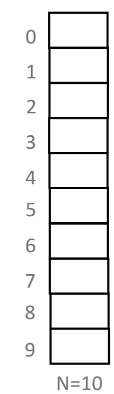
  We define: $h_2(k) = [(ak + b) \; mod \; p] \; mod \; N$

Example: if $|U| = 60$

the keys are: {0, 5, 10, 15, 20, 30, 35, 40, 45, 50, 55}

we choose: $p=101 \quad a=30 \quad b=6$

For a table of size N=10: $h_2(k) = [(30k + 6) \; mod \; 101] \; mod \; 10$



Compression Functions

h_2: *(64−bit integers)* $\rightarrow \{0, 1, \ldots, N-1\}$

<u>Common Approaches</u>:

- The Division Method
- The Multiply-Add-and-Divide (MAD) Method

  Let p be a prime number, $p>|U|$

  Let a be a random number from $\{1, 2, \ldots, p\text{-}1\}$

  Let b be a random number from $\{0, 1, 2, \ldots, p\text{-}1\}$

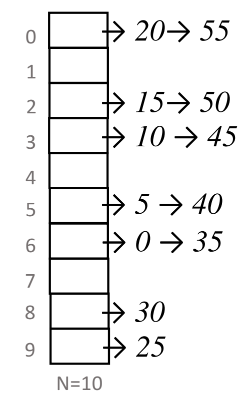
  <u>We define</u>: $h_2(k) = [(ak + b)\ mod\ p]\ mod\ N$

Example: if $|U|=60$

the keys are: {0, 5, 10, 15, 20, 30, 35, 40, 45, 50, 55}

we choose: $p=101 \quad a=30 \quad b=6$

For a table of size N=10: $h_2(k) = [(30k + 6)\ mod\ 101]\ mod\ 10$



Time Analysis of *Find*

Time Analysis of *Find*

Worst-Case:

Time Analysis of *Find*

Worst-Case: all keys are mapped to the same slot, it takes $\theta(n)$ to scan that chain

Time Analysis of $\boldsymbol{Find}$

Worst-Case: all keys are mapped to the same slot, it takes $\theta(n)$ to scan that chain

Expected-Time:

Time Analysis of $Find$

Worst-Case: all keys are mapped to the same slot, it takes $\theta(n)$ to scan that chain

Expected-Time:

I. We assume that our hash function satisfies the "Uniform Hashing" property

Time Analysis of *Find*

Worst-Case: all keys are mapped to the same slot, it takes $\theta(n)$ to scan that chain

Expected-Time:

I. We assume that our hash function satisfies the "Uniform Hashing" property

II. Let $\propto$ be the load-factor of the table

Time Analysis of $Find$

Worst-Case: all keys are mapped to the same slot, it takes $\theta(n)$ to scan that chain

Expected-Time:

I. We assume that our hash function satisfies the "Uniform Hashing" property

II. Let $\propto$ be the load-factor of the table

That is: $\propto = \frac{n}{N}$

Time Analysis of $\boldsymbol{Find}$

Worst-Case: all keys are mapped to the same slot, it takes $\theta(n)$ to scan that chain

Expected-Time:

I. We assume that our hash function satisfies the “Uniform Hashing” property

II. Let $\propto$ be the load-factor of the table

That is: $\propto = \frac{n}{N}$

$\Downarrow$

$\propto =$

Time Analysis of $\boldsymbol{Find}$

Worst-Case: all keys are mapped to the same slot, it takes $\theta(n)$ to scan that chain

Expected-Time:

I. We assume that our hash function satisfies the "Uniform Hashing" property

II. Let $\propto$ be the load-factor of the table

That is: $\propto = \frac{n}{N}$

$\Downarrow$

$\propto = \text{expected length of each chain}$

Time Analysis of $Find$

Worst-Case: all keys are mapped to the same slot, it takes $\theta(n)$ to scan that chain

Expected-Time:

I. We assume that our hash function satisfies the "Uniform Hashing" property

II. Let $\propto$ be the load-factor of the table

That is: $\propto = \frac{n}{N}$

$\Downarrow$

$\propto =$ expected length of each chain

$\Downarrow$

Expected time for $Find$ =

Time Analysis of $Find$

Worst-Case: all keys are mapped to the same slot, it takes $\theta(n)$ to scan that chain

Expected-Time:

I. We assume that our hash function satisfies the "Uniform Hashing" property

II. Let $\propto$ be the load-factor of the table

That is: $\propto = \frac{n}{N}$

$\Downarrow$

$\propto =$ expected length of each chain

$\Downarrow$

Expected time for $Find$ = $\theta(1 + \propto)$

Time Analysis of $Find$

Worst-Case: all keys are mapped to the same slot, it takes $\theta(n)$ to scan that chain

Expected-Time:

I. We assume that our hash function satisfies the "Uniform Hashing" property

II. Let $\propto$ be the load-factor of the table

That is: $\propto = \frac{n}{N}$

$\Downarrow$

$\propto =$ expected length of each chain

$\Downarrow$

Expected time for $Find$ = $\theta(1 + \propto)$

Calculate the hash function and Access slot

Time Analysis of $Find$

Worst-Case: all keys are mapped to the same slot, it takes $\theta(n)$ to scan that chain

Expected-Time:

I. We assume that our hash function satisfies the “Uniform Hashing” property

II. Let $\propto$ be the load-factor of the table

That is: $\propto = \frac{n}{N}$

$\Downarrow$

$\propto =$ expected length of each chain

$\Downarrow$

Scan the chain

Expected time for $Find$ = $\theta(1 + \propto)$

Calculate the hash function and Access slot

Time Analysis of *Find*

Worst-Case: all keys are mapped to the same slot, it takes $\theta(n)$ to scan that chain

Expected-Time:

I. We assume that our hash function satisfies the "Uniform Hashing" property

II. Let $\propto$ be the load-factor of the table

That is: $\propto = \frac{n}{N}$

$\Downarrow$

$\propto =$ expected length of each chain

$\Downarrow$

Scan the chain

Expected time for *Find* = $\theta(1 + \propto)$

Calculate the hash function and Access slot

$\Downarrow$

Time Analysis of $Find$

Worst-Case: all keys are mapped to the same slot, it takes $\theta(n)$ to scan that chain

Expected-Time:

I. We assume that our hash function satisfies the "Uniform Hashing" property

II. Let $\propto$ be the load-factor of the table

That is: $\propto = \frac{n}{N}$

$\Downarrow$

$\propto =$ expected length of each chain

$\Downarrow$

Scan the chain

Expected time for $Find$ = $\theta(1 + \propto)$

Calculate the hash function and Access slot

$\Downarrow$

If **we always maintain $\boldsymbol{n \leq N}$**

Time Analysis of $\boldsymbol{Find}$

Worst-Case: all keys are mapped to the same slot, it takes $\theta(n)$ to scan that chain

Expected-Time:

I. We assume that our hash function satisfies the "Uniform Hashing" property

II. Let $\propto$ be the load-factor of the table

That is: $\propto = \frac{n}{N}$

$\Downarrow$

$\propto =$ expected length of each chain

$\Downarrow$

Scan the chain

Expected time for $Find$ = $\theta(1 + \propto)$

Calculate the hash function and Access slot

$\Downarrow$

If **we always maintain** $\boldsymbol{n \leq N} \quad \Rightarrow$

Time Analysis of $\boldsymbol{Find}$

Worst-Case: all keys are mapped to the same slot, it takes $\theta(n)$ to scan that chain

Expected-Time:

I. We assume that our hash function satisfies the "Uniform Hashing" property

II. Let $\propto$ be the load-factor of the table

That is: $\propto = \frac{n}{N}$

$\Downarrow$

$\propto =$ expected length of each chain

$\Downarrow$

Scan the chain

Expected time for $Find$ = $\theta(1 + \propto)$

Calculate the hash function and Access slot

$\Downarrow$

If **we always maintain** $\boldsymbol{n \leq N} \quad \Rightarrow \quad \propto \leq 1$

Time Analysis of $Find$

Worst-Case: all keys are mapped to the same slot, it takes $\theta(n)$ to scan that chain

Expected-Time:

I. We assume that our hash function satisfies the "Uniform Hashing" property

II. Let $\propto$ be the load-factor of the table

That is: $\propto = \frac{n}{N}$

$\Downarrow$

$\propto =$ expected length of each chain

$\Downarrow$

Scan the chain

Expected time for $Find$ = $\theta(1 + \propto)$

Calculate the hash function and Access slot

$\Downarrow$

If **we always maintain** $\boldsymbol{n} \leq \boldsymbol{N} \quad \Rightarrow \quad \propto \leq 1 \quad \Rightarrow$

Time Analysis of $Find$

Worst-Case: all keys are mapped to the same slot, it takes $\theta(n)$ to scan that chain

Expected-Time:

I. We assume that our hash function satisfies the "Uniform Hashing" property

II. Let $\propto$ be the load-factor of the table

That is: $\propto = \frac{n}{N}$

$\Downarrow$

$\propto =$ expected length of each chain

$\Downarrow$

Scan the chain

Expected time for $Find$ = $\theta(1 + \propto)$

Calculate the hash function and Access slot

$\Downarrow$

If **we always maintain** $\boldsymbol{n} \leq \boldsymbol{N}$ $\Rightarrow$ $\propto \leq 1$ $\Rightarrow$ Expected time for $Find$ = $\theta(1)$

Implementation Points

-
-
-

Implementation Points

- Collision resolution:
-
-

Implementation Points

- <u>Collision resolution</u>: Chaining implemented as UnsortedArrayMap
-
-

Implementation Points

- Collision resolution: Chaining implemented as UnsortedArrayMap
- Hash function:
-

Implementation Points

- Collision resolution: Chaining implemented as UnsortedArrayMap
- Hash function: using build-in hash() function for coding + MAD method for compressing
-

Implementation Points

- Collision resolution: Chaining implemented as UnsortedArrayMap
- Hash function: using build-in hash() function for coding + MAD method for compressing
- Performance:

Implementation Points

- Collision resolution: Chaining implemented as UnsortedArrayMap
- Hash function: using build-in hash() function for coding + MAD method for compressing
- Performance: always keep $n<N$