Algorithms & Data Structures I
CSC 225

Ali Mashreghi

Fall 2018

Department of Computer Science, University of Victoria

Introduction

DICTIONARY

Dictionary is an ADT that supports the following operations on a set S of elements:

SEARCH(key): Search for an item with the given key in S.

INSERT(x): Add item x to S.

DELETE(x): Delete item x from S.

Motivation

- A linked list can also perform these operations but it may take $O(n)$ time to search for a key. (n is the size of the linked list)

- We can do these operations in time as fast as $O(1)$ if we implement dictionaries efficiently.

Motivation

- Example of using a dictionary:

1. In a **compiler** that looks up variables and their values
2. In a **spell correction** program that gives you suggestions
3. In **databases** when you want to fetch a student’s records

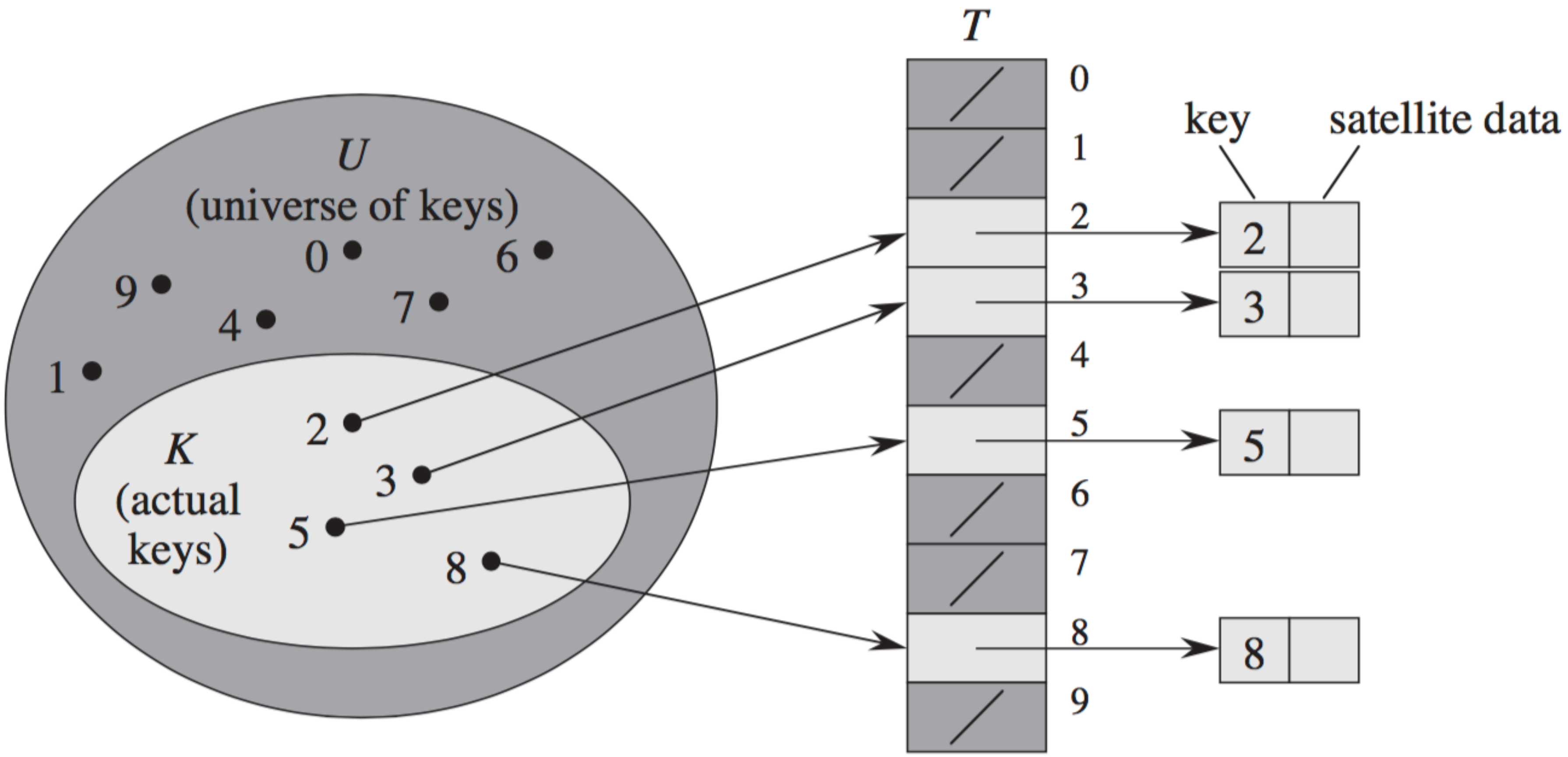
Basic assumptions

- Assume that all elements have **unique keys**.
- We need a data structure that stores the keys in a way that we can **easily** *look up*, *add*, and *remove* keys in that table.

- We have set $\boldsymbol{U}$ which is the **universe of keys**
- We have set $\boldsymbol{K}$ which is the **actual keys** we are working with. $\mathrm{K} \subseteq U$.
- We assume that the keys are integers, **for now**.

Simple solution: direct addressing

- $U = \{0, 1, 2, \ldots, 9\}, K = \{2, 3, 5, 8\}$



Direct-address tables

- In direct addressing we just allocate an array T (i.e. direct-address table) of size $|U|$, and slot k in the table points to the element whose key is k. The implementation is easy:

DIRECT-ADDRESS-SEARCH(T, k)

1 **return** $T[k]$

DIRECT-ADDRESS-INSERT(T, x)

1 $T[x.key] = x$

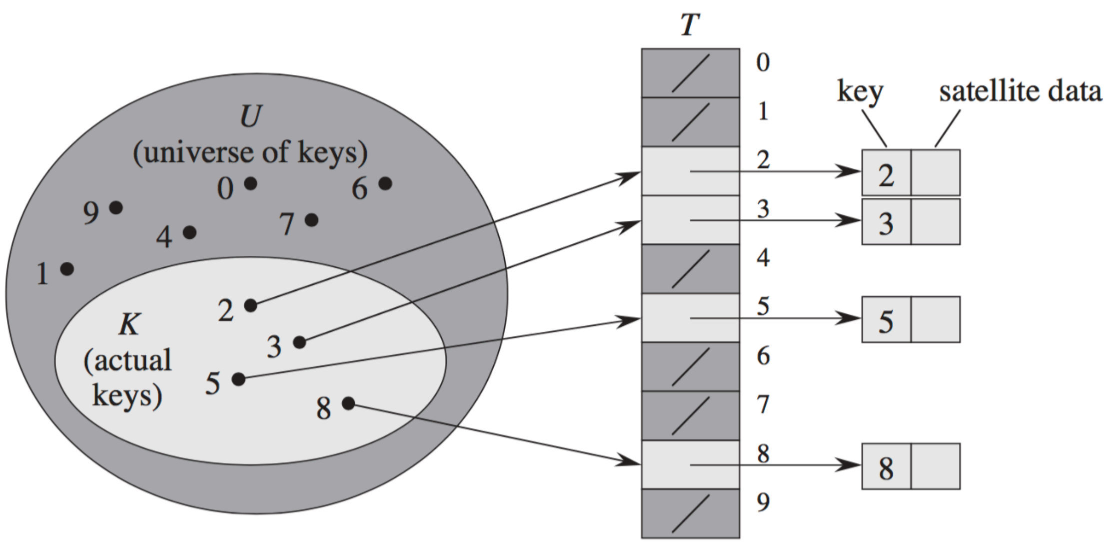
DIRECT-ADDRESS-DELETE(T, x)

1 $T[x.key] =$ NIL

All operations are worst-case $O(1)$ time.

Issues with direct accessing

- The main issue is that the size of universe might be huge.
- Imagine we choose 1000 keys from the set of all 64-bit integers: $|U| = 2^{64} \sim 10^{18}$, but $|K| = 1000$
  - **Issue 1:** It may not be possible to allocate this much memory
  - **Issue 2:** Even if we do, a lot of it is wasted.



Hashing

Chopping and mixing around

✔

Hashish, Hashashin

❌

Hashing

- The idea is to use a **hash function** $h: U \to \{0, 1, \dots, m-1\}$

- Instead of putting the item with key k in the slot k of the table, we put it in the slot $h(k)$ which is computed by the hash function.

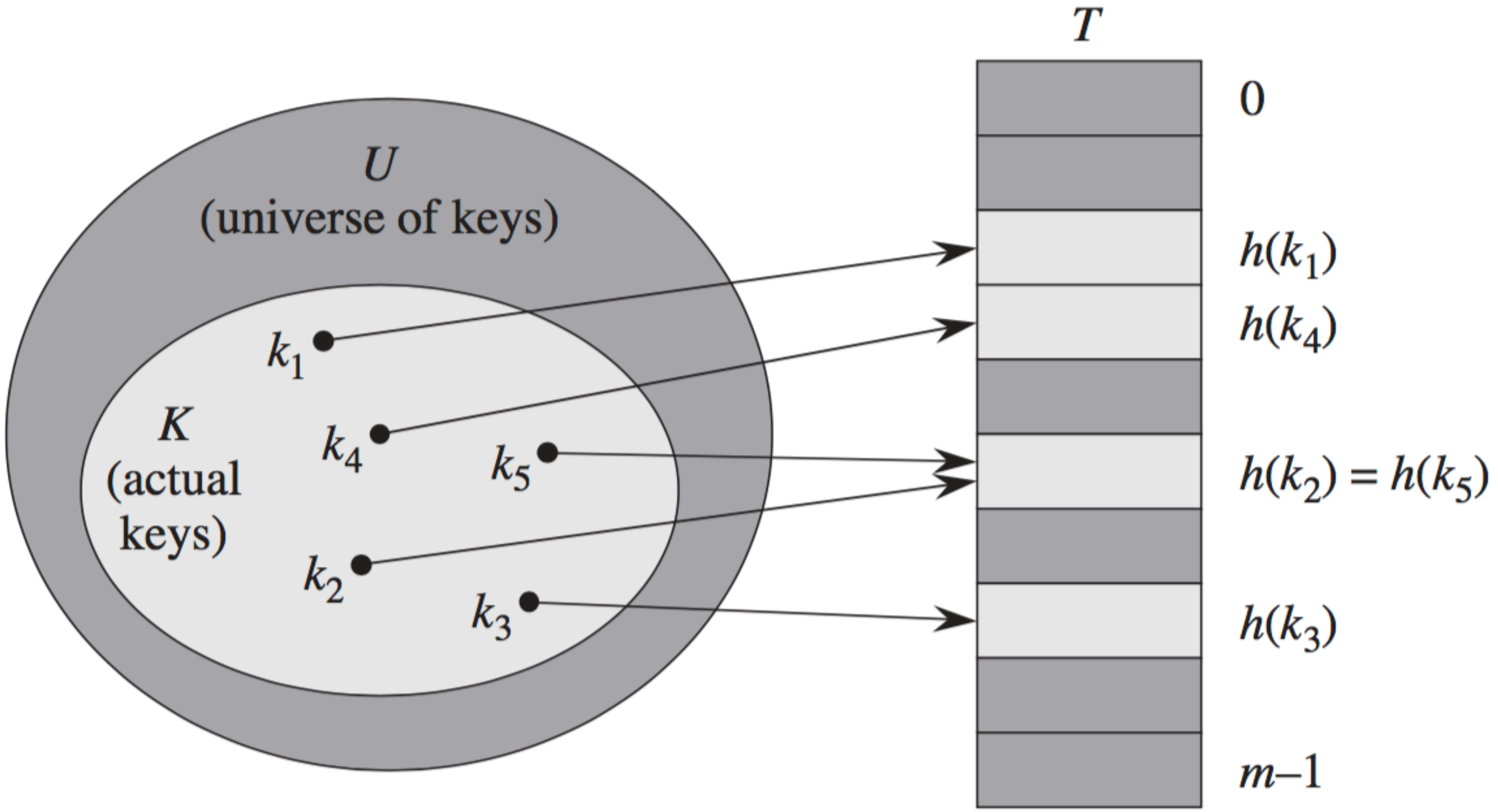
- As a result, we will only need $O(|K|) = O(m)$ space for the table T.

Hashing

Some terminology:

- T is a **hash-table**.
- $h(k)$ is the **hash value** of k.
- Key k **hashes** to $h(k)$.

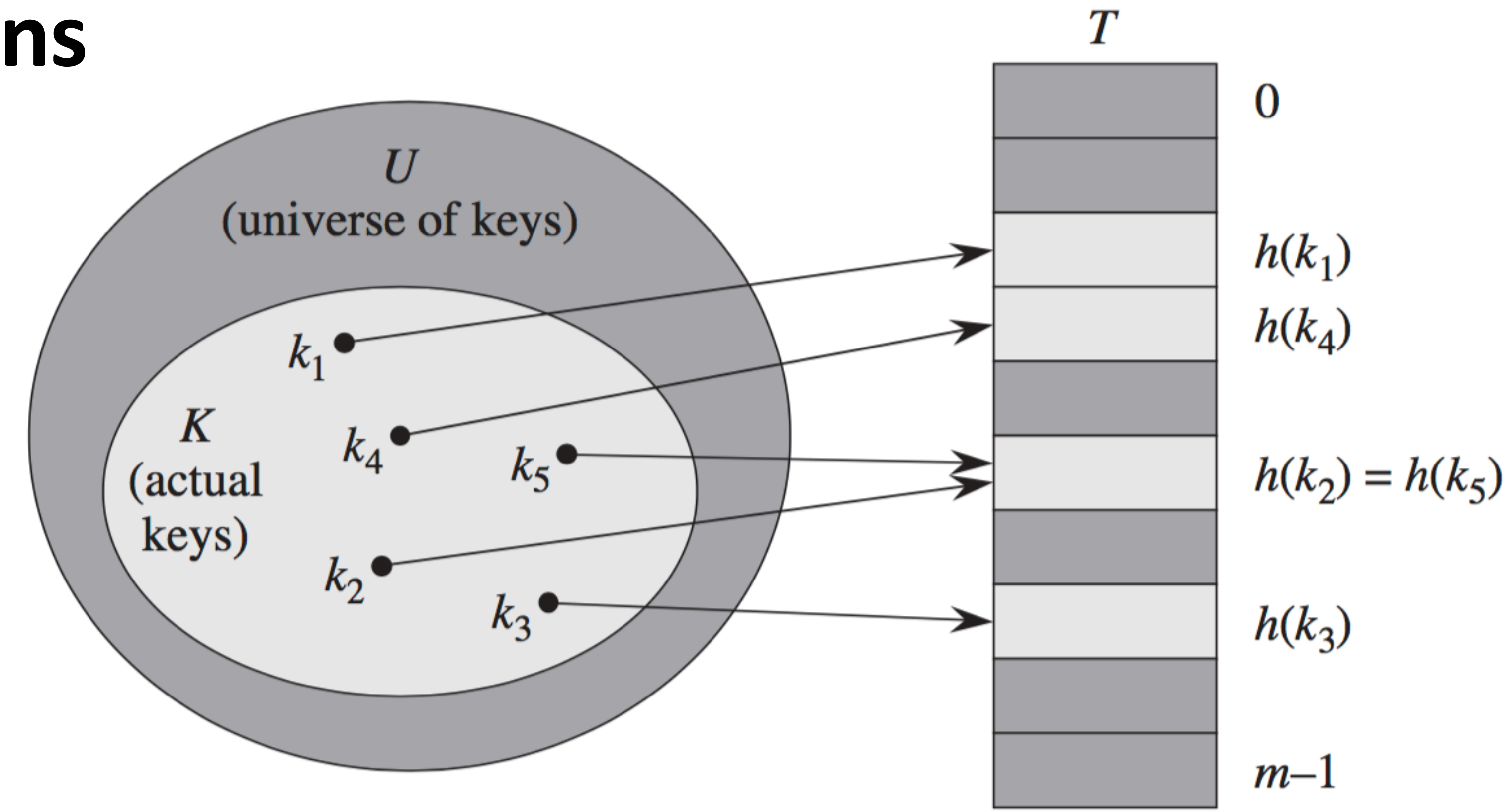
Hashing



U
(universe of keys)
K
(actual keys)
k1
k4
k5
k2
k3
T
0
h(k1)
h(k4)
h(k2) = h(k5)
h(k3)
m–1

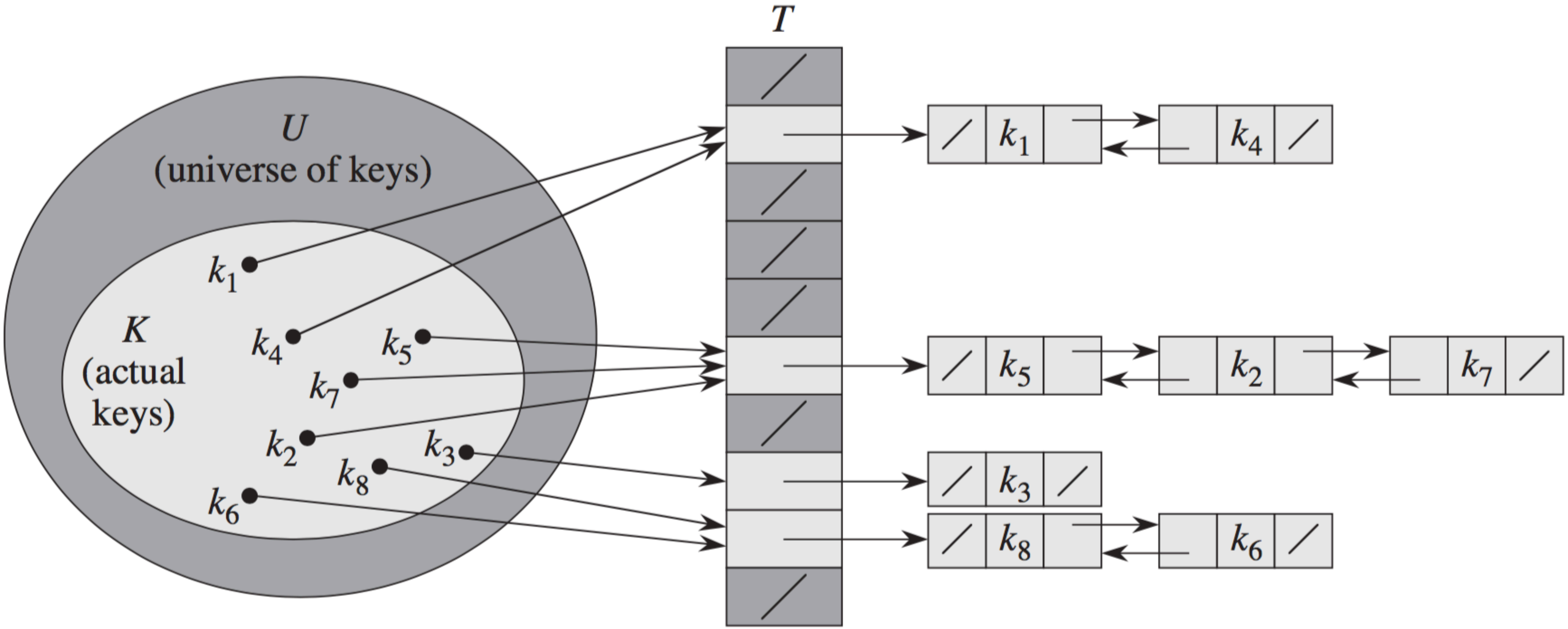
Problems to solve

1. **Resolving collisions** (i.e. when two keys are hashed to the same slot)
2. **Find hash functions**



Collision resolution - chaining

- In ***chaining***, we place all the elements that hash to the same slot into the same linked list.



Chaining - implementation

CHAINED-HASH-INSERT(T, x)

1 insert x at the head of list $T[h(x.key)]$

CHAINED-HASH-SEARCH(T, k)

1 search for an element with key k in list $T[h(k)]$

CHAINED-HASH-DELETE(T, x)

1 delete x from the list $T[h(x.key)]$

Chaining - analysis

- Basically, we want to show that the **expected length** is $O(1)$.

- We do the analysis under an assumption called **simple uniform hashing**.

Chaining - analysis

- **Simple uniform hashing** is the assumption that each of the keys is **equally likely** to hash to any of the m cells independently of where other keys have hashed to.

Chaining - analysis

- That is, we assume that each of the keys is **equally likely** to hash to any of the m slots independently of where other keys have hashed to.

- **Important note:** The hash function itself **should not** be randomized since we want it to always hash the same key to the same slot; however, in the real world, good hash functions which guarantee low collisions, are **constructed randomly**. But their behaviour is deterministic.

Chaining - analysis

- Now, say we have a **hash table $\boldsymbol{T}$** with m slots that stores n keys.

- We define load factor as $\alpha = \frac{number\ of\ elem.in\ T}{number\ of\ slots\ in\ T} = \frac{n}{m}$.

- This ratio shows how much your table is loaded.

Chaining - analysis

- We prove that **under the simple uniform** hashing assumption, the expected length of any list is $\frac{n}{m}$.

- In fact, the expected length is exactly **equal to the load factor** of the table α.

- Say Y_i is the random variable representing the number of keys that **hash to position i**.

Chaining - analysis

- We define $X_{i,j} = \begin{cases} 1 & if\ h(k_j) = i \\ 0 & otherwise \end{cases}$
- A **0-1 random variable** is called an **indicator random variable**.
- If X is a random variable, then $E[X] = \Pr(X = 1)$
- Since $Y_i = \sum_{j=1}^{n} X_{i,j}$, we show that $E[Y_i] = \frac{n}{m}$

Chaining - analysis

- Therefore, by **picking $\boldsymbol{m \geq \frac{n}{c}}$**, where c is some constant, the **expected length will be $\boldsymbol{\leq c = O(1)}$** and all dictionary operations can be done in $O(1)$ time, as well.

- Later we will show an alternative method (not necessarily faster) to store all n elements **inside** the table and **without** using pointers.

Hash functions

- A good hash function is one that is **close to satisfying** the assumption of **simple uniform hashing**.
- We present three methods for constructing hash functions:

1. The division method
2. The multiplication method
3. Universal hash functions (this construction is **random**)

Converting keys to integers

- Since hash functions work with a **natural number** ($\in \{0, 1, 2, \ldots.\}$) as input we need a way to interpret keys as natural numbers.
- A very common is that if we are dealing with strings as keys, we can interpret the string as an integer in base 256. For example **UVic**

256^3	256^2	256	1
U	V	i	c

$=$

256^3	256^2	256	1
085	086	105	99

$= \ldots$

Converting keys to integers

- Programming languages usually use an object's physical address in memory (which is an integer) as a unique integer representing that object.

- Since the address of the object doesn't change **during the execution**, this number can be used as the key.

Converting keys to integers

- Java's **hashCode** method is an example that returns a unique integer for a given object. (Used in Java's **HashMap**)

- Note that **two separate executions** of a program may result in different keys, and hence different hash values.

The division method

- The hash function is

$$h(k) = k \,\%\, m$$

The division method

- The hash function is

$$h(k) = k \,\%\, m$$

- This method is very bad!
- **Question:** Say $m = 8$, provide an example of a collision?

The division method

- The hash function is
$$h(k) = k \,\%\, m$$
- This method is very bad!
- **Question:** Say $m = 8$, provide an example of a collision?
- **Answer:** $k_1 = 8, k_2 = 16$, and $h(k_1) = h(k_2) = 0$

The division method

- The hash function is

$$h(k) = k \,\%\, m$$

- This method is very bad!

- Certain values should be avoided for m. For example, if $m = 2^p$ (is a power of 2), then $k \,\%\, m$ is equal to the p lowest-order bits of key.

					p	...	1	0
k	1	0	1	0	1	...	1	0

The division method

- Usually, **the best choice** for m is a **prime close to** $\boldsymbol{n}$ (number of keys) but not close to a power of 2.

- It's because a **prime has only two divisors** (1 and itself) which are less likely to be in common with the divisors of the keys.

- Also, a prime **prevents** to some extent **patterns** in the hash values.

The division method

- For example, say we have **10 keys** which are multiples of 14. We pick $m = 12$ (not a prime).

14 mod 12 = 2
28 mod 12 = 4
42 mod 12 = 6
56 mod 12 = 8
70 mod 12 = 10

84 mod 12 = 0
98 mod 12 = 2
112 mod 12 = 4
126 mod 12 = 6
140 mod 12 = 8

- A pattern appears because 14 and 12 have 2 as a common divisor.

The division method

- For example, say we have **10 keys** which are multiples of 14. We pick $m = 12$ (not a prime).

14 mod 12 = 2	84 mod 12 = 0
28 mod 12 = 4	98 mod 12 = 2
42 mod 12 = 6	112 mod 12 = 4
56 mod 12 = 8	126 mod 12 = 6
70 mod 12 = 10	140 mod 12 = 8

- So, technically we are using half the capacity of table.

The division method

- But if we $m = 11$ (a prime not close to 8 or 16):

14 mod 11 = 3
28 mod 11 = 6
42 mod 11 = 9
56 mod 11 = 1
70 mod 11 = 4

84 mod 11 = 7
98 mod 11 = 10
112 mod 11 = 2
126 mod 11 = 5
140 mod 11 = 8

The multiplication method

- The hash function is

$$h(k) = [(k \cdot s) \% 2^w] \gg (w - p)$$

- First we **multiply** the key **by s**.
- Then we take the result **mod 2^w**.
- Finally we **shift right** the value by **$(w - p)$ bits**.

The multiplication method

- The hash function is

$$h(k) = [(k \cdot s) \,\%\, 2^w] \gg (w - p)$$

- We assume that the machine works with ***w*-bit** words. (64-bits for example) – w **is known to us** beforehand.

- **Choice of $\boldsymbol{p}$**:

We take $\boldsymbol{m = 2^p}$, such that $p < w$. So, $2^p < 2^w$.

Again, m should be near $\frac{n}{c}$ so that the load factor becomes constant.

The multiplication method

- The hash function is

$$h(k) = [(k \cdot s) \,\%\, 2^w] \gg (w - p)$$

- **Choice of s:**
- s is an **integer** in range $(0, 2^w)$
- So, we can say that $s = A.2^w$ where A is a **real number** in range $(0, 1)$

The multiplication method

- The hash function is

$$h(k) = [(k \cdot s) \,\%\, 2^w] \gg (w - p)$$

- **Choice of s:**
- s is an **integer** in range $(0, 2^w)$
- So, we can say that $s = A.2^w$ where A is a **real number** in range $(0, 1)$
- In fact, what matters is the choice of A.
- Knuth says picking $A \approx \frac{\sqrt{5}-1}{2}$ (golden ratio), works well!

The multiplication method

$$h(k) = [(k \cdot s)\ \%\ 2^w] \gg (w - p)$$

- % is the **mod** operation.
- >> is the **shift right** operation. (C/C++, Java, ...)

0	0	**1**	**0**	**1**	**0**	**1**	**0**	42
0	0	0	**1**	**0**	**1**	**0**	**1**	21

The multiplication method

$$h(k) = [(k \cdot s)\ \%\ 2^w] \gg (w - p)$$

- % is the **mod** operation.
- >> is the **shift right** operation. (C/C++, Java, …)

0	0	**1**	**0**	**1**	**0**	**1**	**0**	42
0	0	0	**1**	**0**	**1**	**0**	**1**	21
0	0	0	0	**1**	**0**	**1**	**0**	10

The multiplication method

$$h(k) = [(k \cdot s)\ \%\ 2^w] \gg (w - p)$$

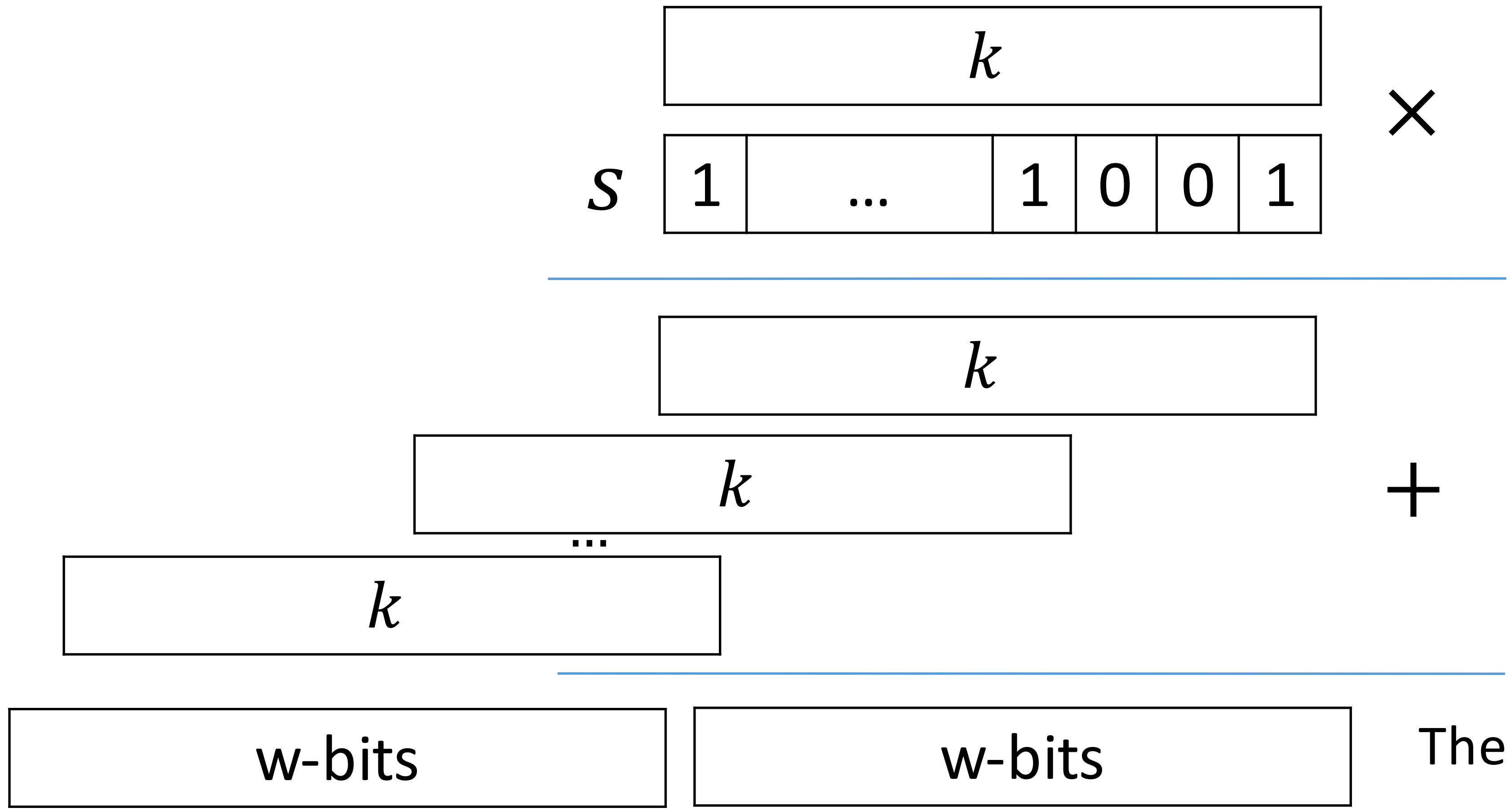
- % is the **mod** operation.
- >> is the **shift right** operation. (C/C++, Java, …)

0	0	**1**	**0**	**1**	**0**	**1**	**0**	42
0	0	0	**1**	**0**	**1**	**0**	**1**	21
0	0	0	0	**1**	**0**	**1**	**0**	10

Shift right is like **integer division** by 2. Similarly, we have shift left which is **multiplication** by 2. These are much faster than normal division and mult. operations.

The multiplication method

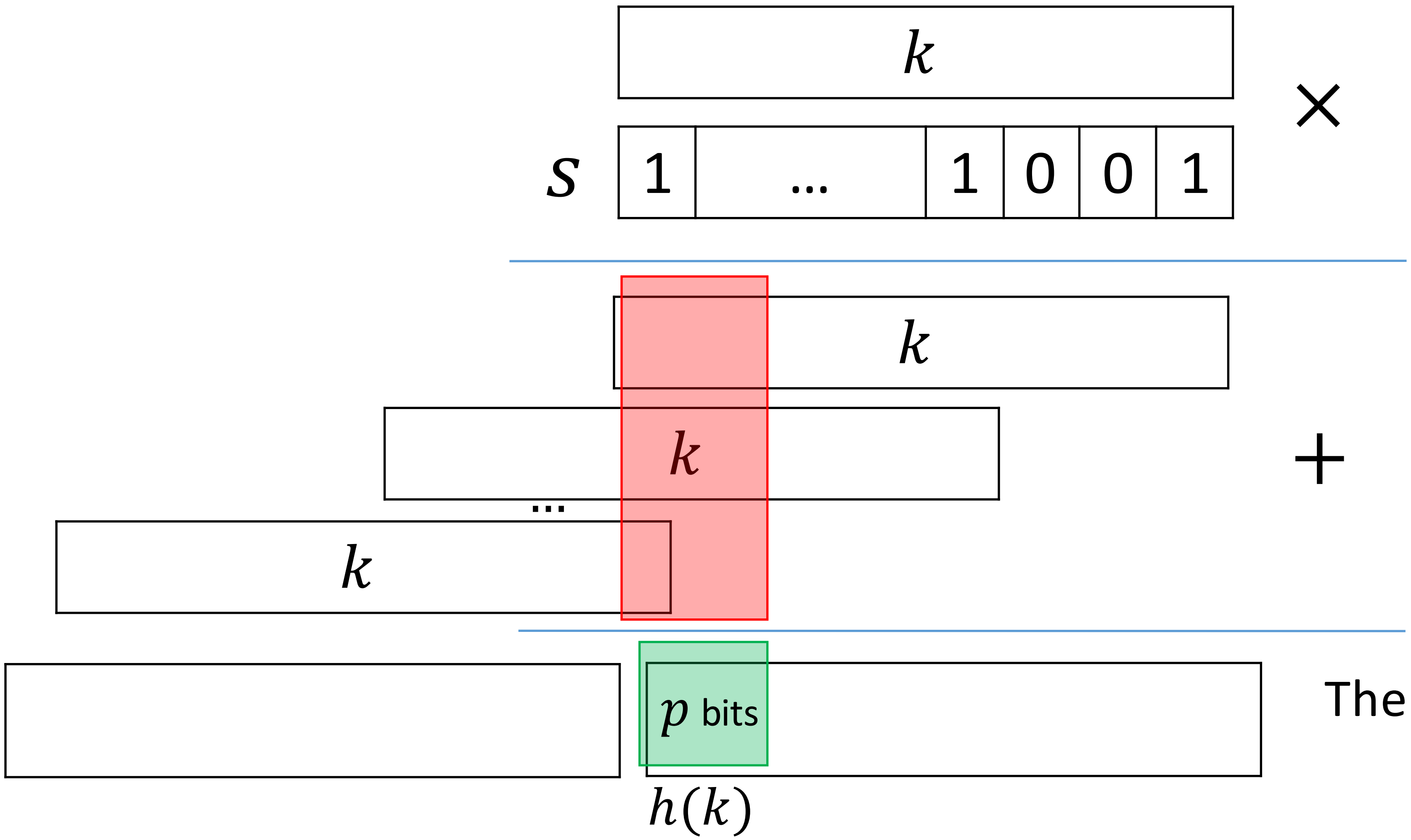
$$h(k) = [(k \cdot s) \ \% \ 2^w] \gg (w - p)$$



The result is a 2w-bit integer

The multiplication method

$$h(k) = [(k \cdot s)\ \%\ 2^w] \gg (w - p)$$



Intuitively, all copies of k, meet in the middle, and affect the p bits in the middle of the results. So, we **extract** the p highest-order bits of the right w-bits of the result!

The result is a 2w-bit integer

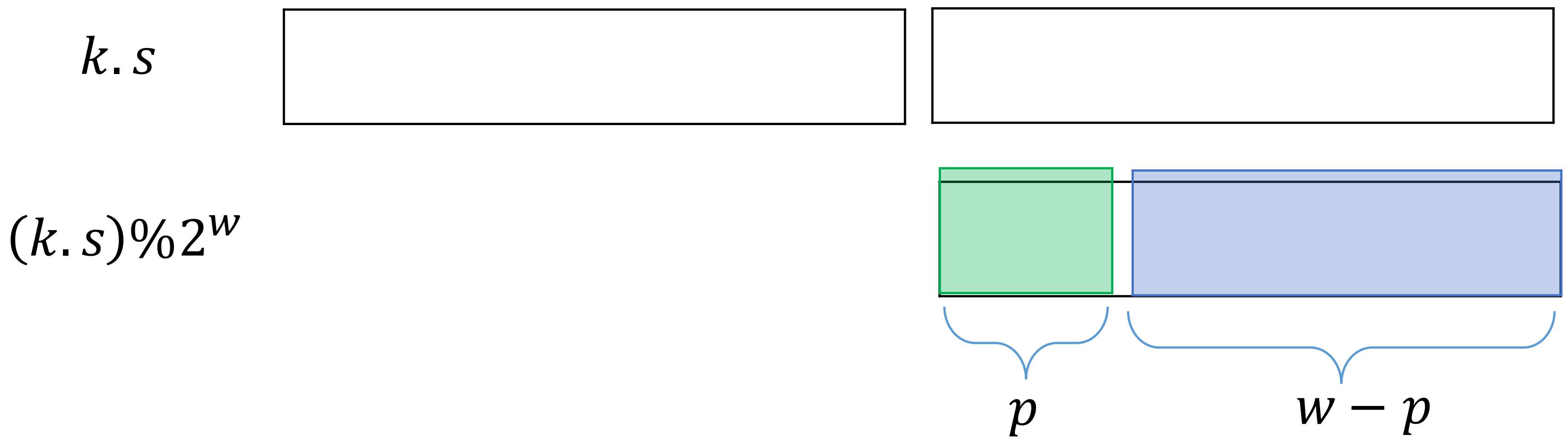
The multiplication method

$$h(k) = [(k \cdot s) \,\%\, 2^w] \gg (w - p)$$

$k.s$

The multiplication method

$$h(k) = [(k \cdot s) \ \% \ 2^w] \gg (w - p)$$



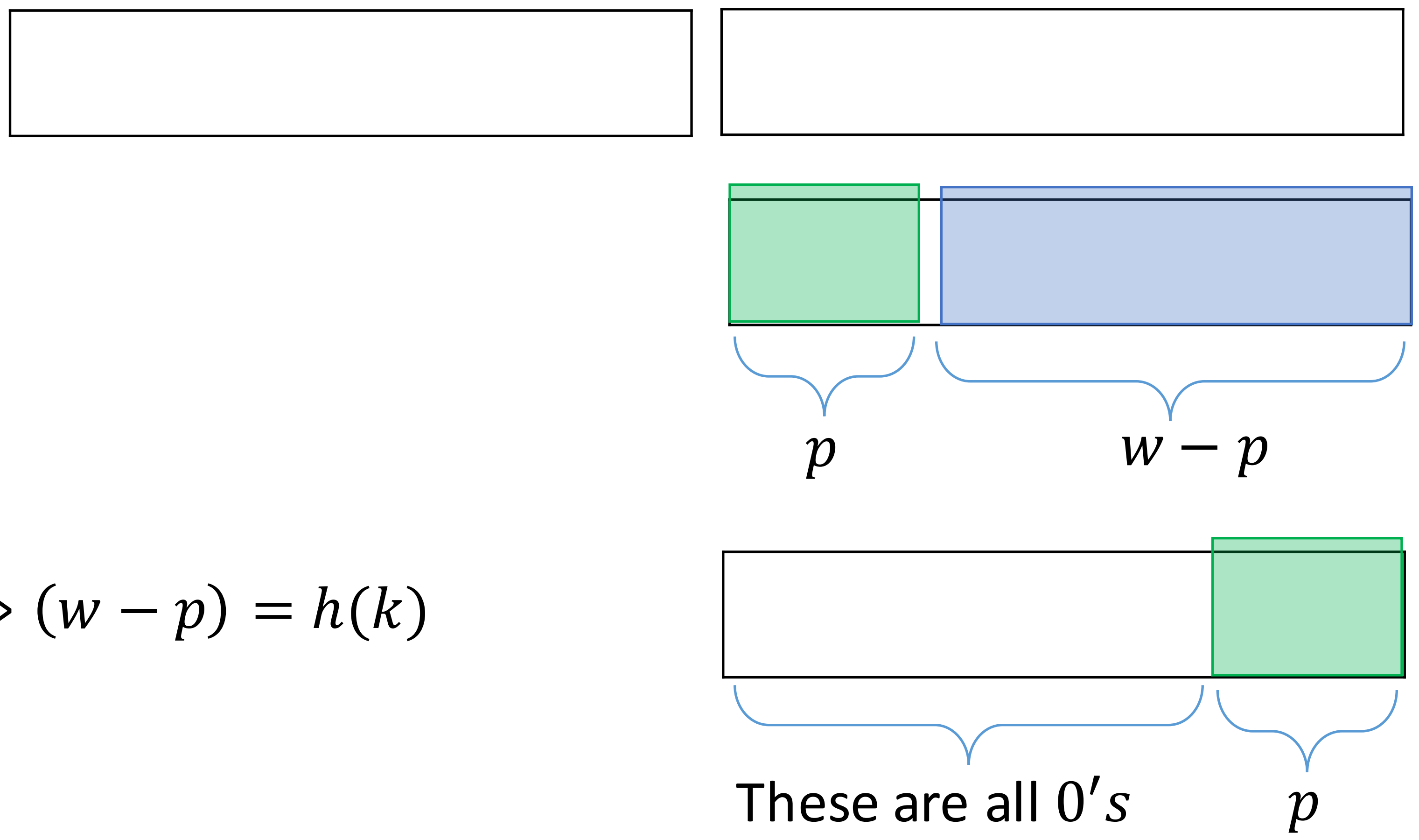
The multiplication method

$$h(k) = [(k \cdot s)\ \%\ 2^w] \gg (w - p)$$

$k.s$

$(k.s)\%2^w$

$[(k.s)\%2^w] \gg (w-p) = h(k)$



Universal hash functions

- A hash function is a universal hash function if:
  $\forall k_1, k_2 \in U, k_1 \neq k_2$: $\mathbf{Pr}[\boldsymbol{h}(\boldsymbol{k_1}) = \boldsymbol{h}(\boldsymbol{k_2})] \leq \mathbf{1}/\boldsymbol{m}$

- This property **guarantees** that even for the worst-case choice of keys, the expected number of collisions in any specific slot is low, i.e. $O(\alpha)$.

Proof of low number of collisions

- The proof is very similar to the one we had for expected length of linked lists assuming **simple uniform hashing**.

- However, since we **don't have that assumption** here we have to use the fact that $\mathbf{Pr}[\boldsymbol{h}(\boldsymbol{k}_1) = \boldsymbol{h}(\boldsymbol{k}_2)] \leq \mathbf{1}/\boldsymbol{m}$ and define our indicator random variables differently.

Proof of low number of collisions

We define $X_{i,j} = \begin{cases} \mathbf{1} & if\ \boldsymbol{h(k_i) = h(k_j)} \\ 0 & otherwise \end{cases}$

We know $\boldsymbol{E[X_{ij}]} = \Pr(X_{ij} = 1) = \Pr\left(h(k_i) = h(k_j)\right) \boldsymbol{\leq \frac{1}{m}}$

Suppose for a specific slot i, Y_i is the number of keys hashed to that slot. (we have n keys overall)

$$Y_i = \sum_{j \neq i} X_{i,j} \rightarrow E[Y_i] = E\left[\sum_{j \neq i} X_{i,j}\right] = (linearity) \sum_{j \neq i} E[X_{i,j}] \leq \frac{n}{m}$$

The last inequality is because (1) we have n keys, and (2) $E[X_{i,j}] \leq \frac{1}{m}$

Universal hash functions

- An easy way to construct a universal hash function is:

1. Pick a prime p such that all keys fall in $[0, p-1]$
2. Pick a, b **independently at random** from $[0, p-1]$
3. Then, $h(k) = \big((ak + b)\ \%\ p\big)\%m$

Universal hash functions

- An easy way to construct a universal hash function is:

1. Pick a prime p such that all keys fall in $[0, p-1]$
2. Pick a, b **independently at random** from $[0, p-1]$
3. Then, $h(k) = \big((ak + b) \,\%\, p\big)\%m$

Theorem: if k and l are different keys $\Pr\big(h(k) = h(l)\big) \leq \frac{1}{m}$;
so, h is a universal hash function.
The proof is optional and is in page 267-268 of CLRS.