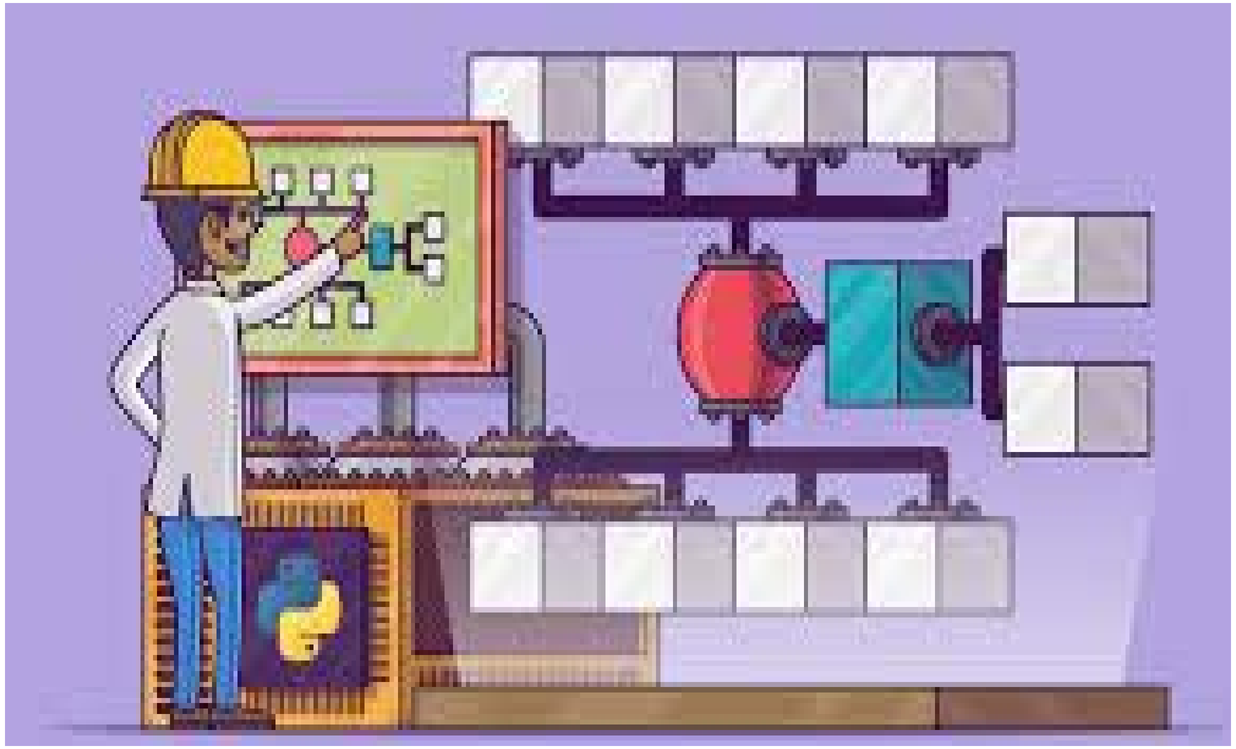
ساختمان داده ها



مدرس:
سمانه حسینی سمنانی

دانشگاه صنعتی اصفهان- دانشکده برق و کامپیوتر

درهم سازی – Hashing

- Direct-address tables
- Hash tables
- Hash functions
- Open addressing
- Perfect hashing

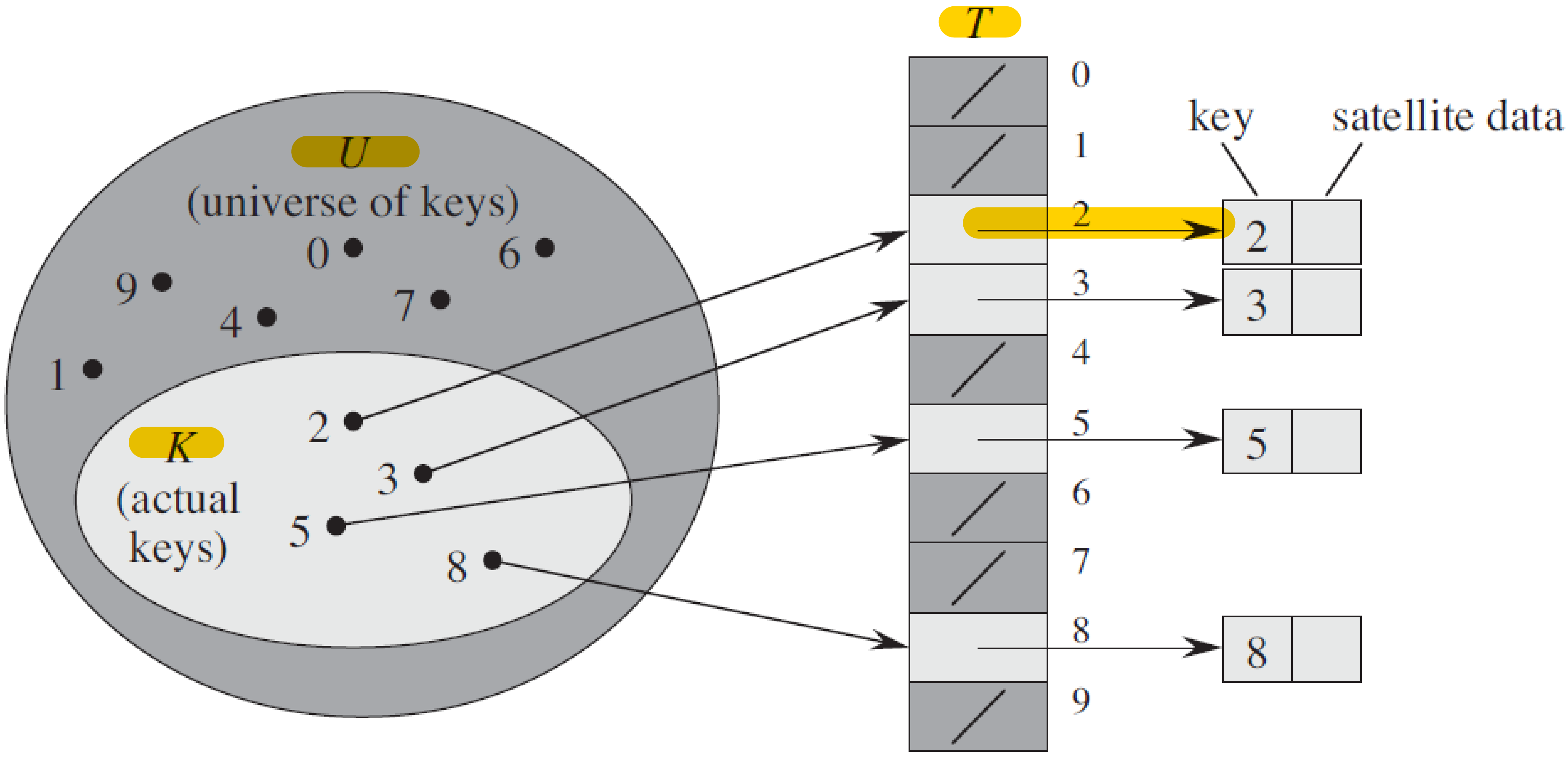
درهم سازی – Hashing

- Many applications require : INSERT, SEARCH, and DELETE.
- For example, a compiler that translates a programming language maintains a symbol table, in which the keys of elements are arbitrary character strings corresponding to identifiers in the language.
- A hash table is an effective data structure for implementing dictionaries.
- Under reasonable assumptions, the average time to search for an element in a hash table is O(1)

Direct-address tables

- Direct addressing is a simple technique that works well when the universe U of keys is reasonably small.
- Suppose that an application needs a dynamic set in which each element has a key drawn from the universe $U = \{0, 1, \ldots, m-1\}$, where m is not too large.
- We shall assume that no two elements have the same key.
- To represent the dynamic set, we use an array, or direct-address table, denoted $T[0 \,..\, m-1]$
- in which each position, or slot, corresponds to a key in the universe U

Direct-address tables



Direct-address tables

DIRECT-ADDRESS-SEARCH(T, k)

return $T[k]$

DIRECT-ADDRESS-INSERT(T, x)

$T[x.key] = x$

DIRECT-ADDRESS-DELETE(T, x)

$T[x.key] = \text{NIL}$

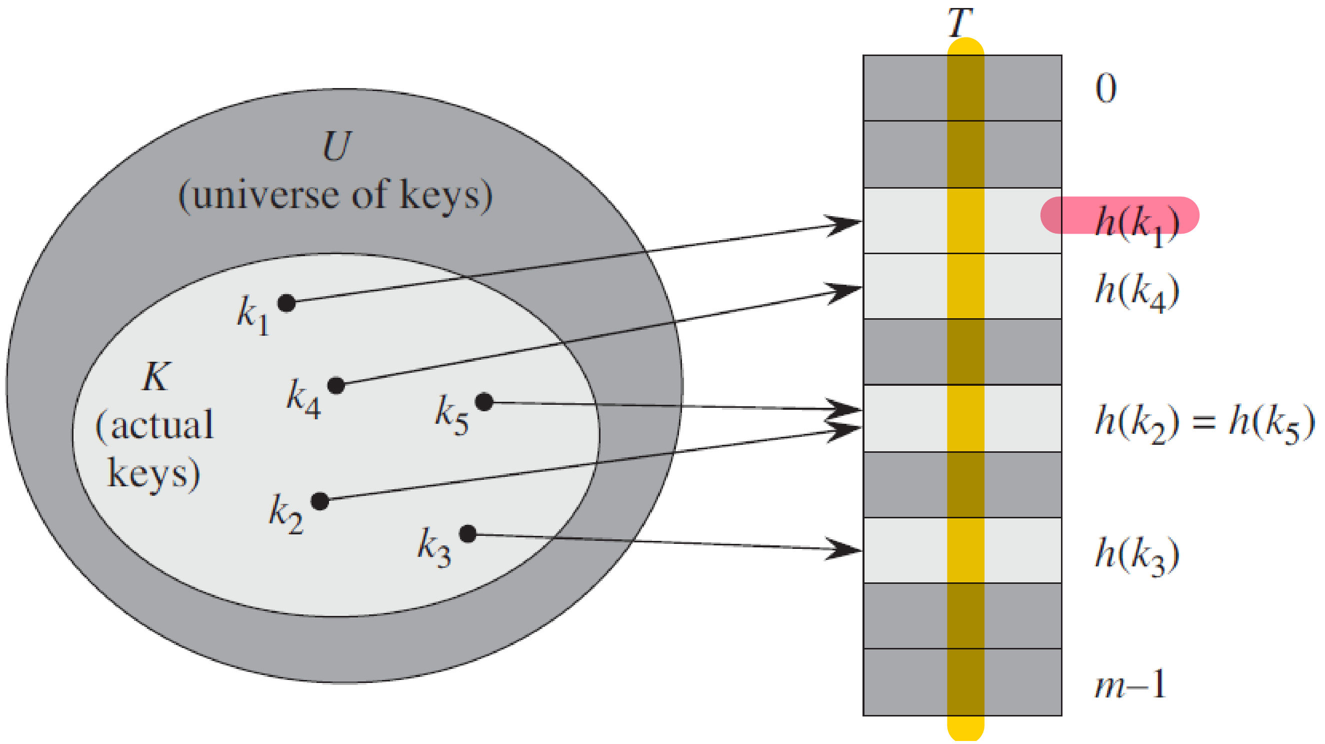
Each of these operations takes only $O(1)$ time.

Hash table vs. Direct-address tables

- if the universe U is large, storing a table T of size |U| may be impractical or even impossible, given the memory available on a typical computer.
- Furthermore, the set K of keys actually stored may be so small relative to U that most of the space allocated for T would be wasted.
- When the set K of keys stored in a dictionary is much smaller than the universe U of all possible keys, a hash table requires much less storage than a direct address table.
- we can reduce the storage requirement to $\Theta(|K|)$ while we maintain the benefit that searching for an element in the hash table still requires only O(1) time.
- The catch is that this bound is for the average-case time, whereas for direct addressing it holds for the worst-case time.

Hash tables

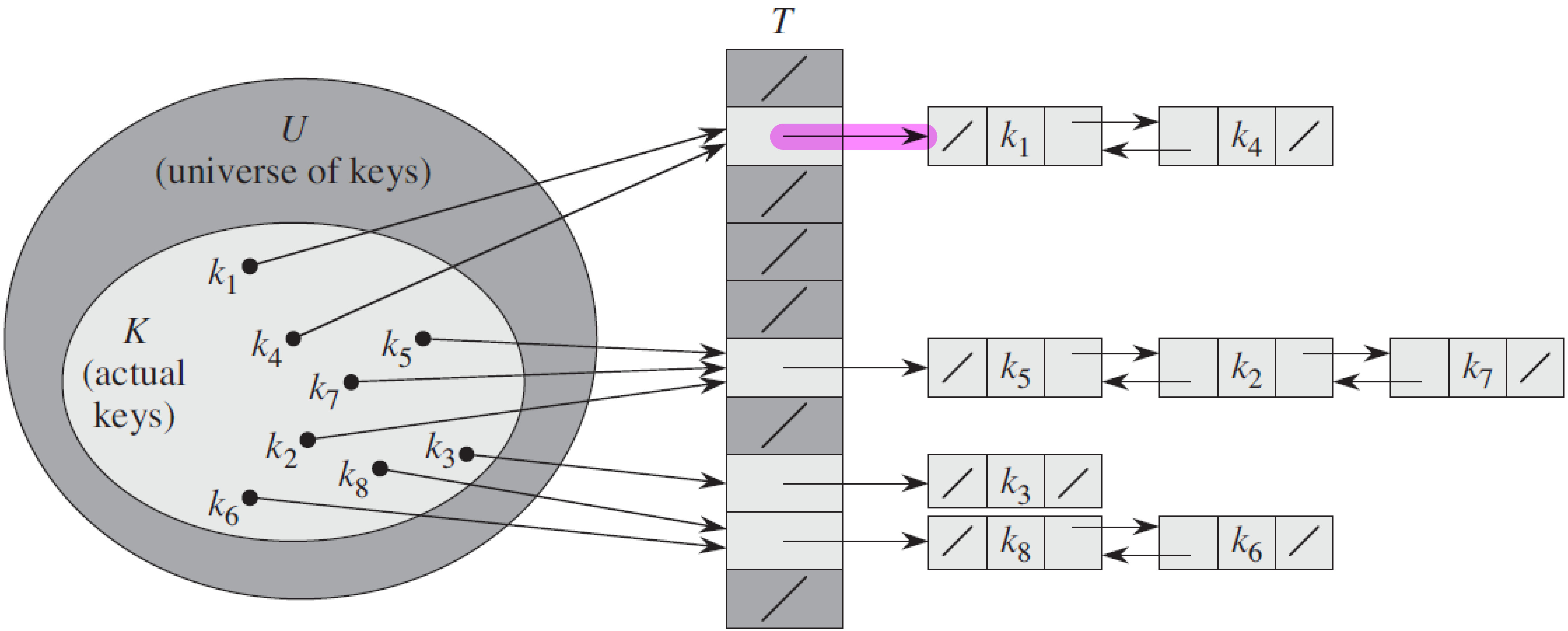
$$h : U \rightarrow \{0, 1, \ldots, m-1\}$$



Hash tables

- Of course, the ideal solution would be to avoid collisions altogether.
- We might try to achieve this goal by choosing a suitable hash function h.
- “random”-looking hash function can minimize the number of collisions,
- we still need a method for resolving the collisions that do occur.
- collision resolution technique:
  - chaining.
  - open addressing.

Collision resolution by chaining



Collision resolution by chaining

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)

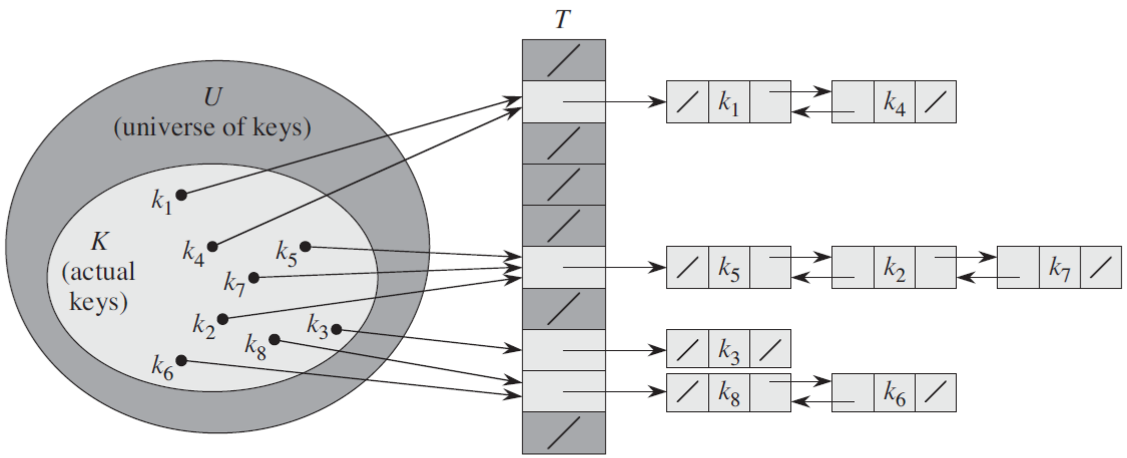
1 delete x from the list $T[h(x.key)]$

Analysis of hashing with chaining

- Given a hash table T with m slots that stores n elements
- we define the load factor α for T as n/m
- the average number of elements stored in a chain.
- The worst-case behavior of hashing with chaining is terrible: all n keys hash to the same slot, creating a list of length n.
- The worst-case time for searching is thus $\theta(n)$ plus the time to compute the hash function.
- The average-case performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average.

Simple uniform hashing

any given element is equally likely to hash into any of the m slots, independently of where any other element has hashed to.



Analysis of hashing with chaining

Theorem 11.1

In a hash table in which collisions are resolved by chaining, an unsuccessful search takes average-case time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.

Analysis of hashing with chaining

Theorem 11.2

In a hash table in which collisions are resolved by chaining, a successful search takes average-case time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.

Hash functions

- design of good hash functions:
  - hashing by division
  - hashing by multiplication
  - universal hashing

What makes a good hash function?

- A good hash function satisfies (approximately) the assumption of simple uniform hashing
- Occasionally we do know the distribution.
- For example: if we know that the keys are random real numbers k independently and uniformly distributed in the range $0 \leq k < 1$, then the hash function:

$$h(k) = \lfloor km \rfloor$$

- satisfies the condition of simple uniform hashing.

What makes a good hash function?

- Qualitative information about the distribution of keys may be useful in this design process.
- For example, consider a compiler’s symbol table, in which the keys are character strings representing identifiers in a program.
- Closely related symbols, such as pt and pts, often occur in the same program.
- A good hash function would minimize the chance that such variants hash to the same slot.

The division method

In the ***division method*** for creating hash functions, we map a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is

$$h(k) = k \bmod m$$

For example, if the hash table has size $m = 12$ and the key is $k = 100$, then $h(k) = 4$. Since it requires only a single division operation, hashing by division is quite fast.

The division method

- When using the division method, we usually avoid certain values of m.
- m should not be a power of 2,
- since if $m = 2^p$, then $h(k)$ is just the p lowest-order bits of k.
- we are better off designing the hash function to depend on all the bits of the key.
- A prime not too close to an exact power of 2 is often a good choice for m.

The division method

- e.g. suppose we wish to allocate a hash table, with collisions resolved by chaining, to hold roughly n = 2000 character strings, where a character has 8 bits.
- We don't mind examining an average of 3 elements in an unsuccessful search, and so we allocate a hash table of size m = 701.
- We could choose m = 701 because it is a prime near 2000/3 but not near any power of 2.
- Treating each key k as an integer, our hash function would be

$$h(k) = k \bmod 701$$

The multiplication method

$$0 < A < 1$$

$$h(k) = \lfloor m\,(kA \bmod 1) \rfloor$$

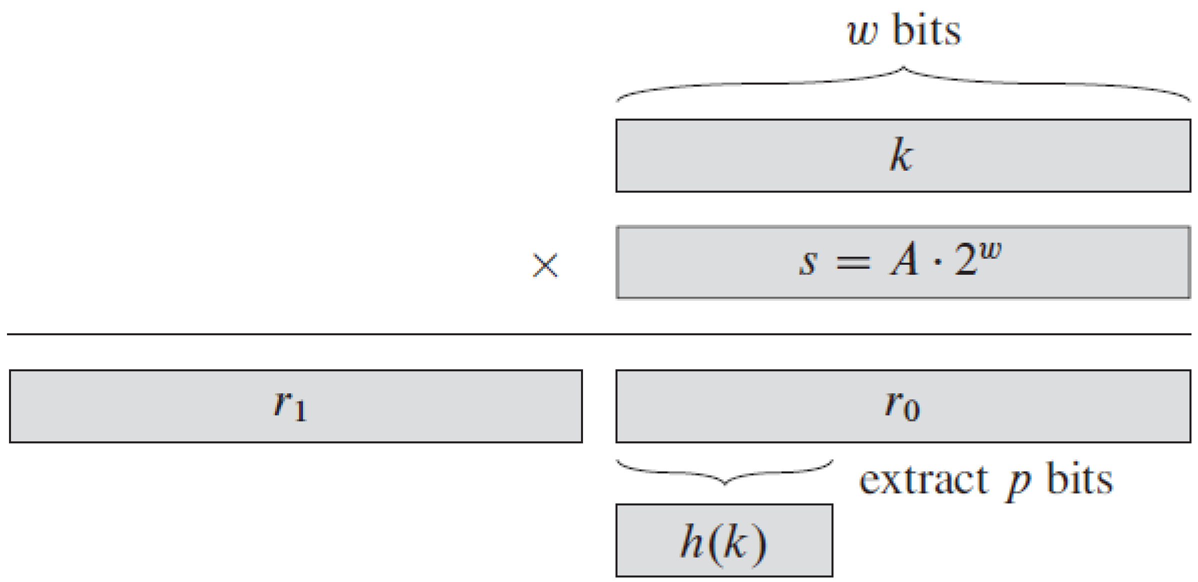
"$kA \bmod 1$" means the fractional part of kA, that is, $kA - \lfloor kA \rfloor$

$$A \approx (\sqrt{5} - 1)/2 = 0.6180339887\ldots$$

We typically choose it to be a power of 2 ($m = 2^p$ for some integer p).

The multiplication method

$$h(k) = \lfloor m\,(kA \bmod 1) \rfloor$$



Universal hashing

- If a malicious adversary chooses the keys to be hashed by some fixed hash function,
- then the adversary can choose n keys that all hash to the same slot,
- Average retrieval time of $\theta(n)$.
- Any fixed hash function is vulnerable to such terrible worst-case behavior
- the only effective way to improve the situation is to choose the hash function randomly in a way that is independent of the keys that are actually going to be stored.
- This approach, called universal hashing, can yield provably good performance on average, no matter which keys the adversary chooses.

Universal hashing

- at the beginning of execution we select the hash function at random from a carefully designed class of functions
- Because we randomly select the hash function, the algorithm can behave differently on each execution, even for the same input, guaranteeing good average-case performance for any input.
- compiler’s symbol table, we find that the programmer’s choice of identifiers cannot now cause consistently poor hashing performance.
- Poor performance occurs only when the compiler chooses a random hash function that causes the set of identifiers to hash poorly,
- but the probability of this situation occurring is small and is the same for any set of identifiers of the same size.

Universal hashing

- Let $\mathcal{H}$ be a finite collection of hash functions that map a given universe U of keys into the range

$$\{0, 1, \ldots, m-1\}$$

$$\mathcal{H} = \{h_1, h_2, h_3 \ \ldots\ldots, h_l\}$$

Designing a universal class of hash functions

$$\mathcal{H} = \{h_1, h_2, h_3 \ \ldots\ldots, h_l\}$$

- برای هر k و t تعداد توابعی که k و t را به یک خانه map می کنند حداکثر $\frac{|\mathcal{H}|}{m}$
- میشود نشان داد که می توانیم چنین مجموعه تابعی تعریف کنیم.

Designing a universal class of hash functions

- P = a prime number larger that all the numbers in the domain
- $h_{ab}(k) = ((ak + b) \bmod p) \bmod m$.
- $a \in \{1, \dots, p-1\}$
- $b \in \{0, \dots, p-1\}$
- $\mathcal{H}_{pm} = \{h_{ab} : a \in \mathbb{Z}_p^* \text{ and } b \in \mathbb{Z}_p\}$

مثال

متوسط زمان جستجو $O(\alpha)$

Theorem 11.5
The class $\mathcal{H}_{pm}$ of hash functions defined by equations (11.3) and (11.4) is universal.

برای هر k و t تعداد توابعی که k و t را به یک خانه map می کنند حداکثر $\frac{|\mathcal{H}|}{m}$