

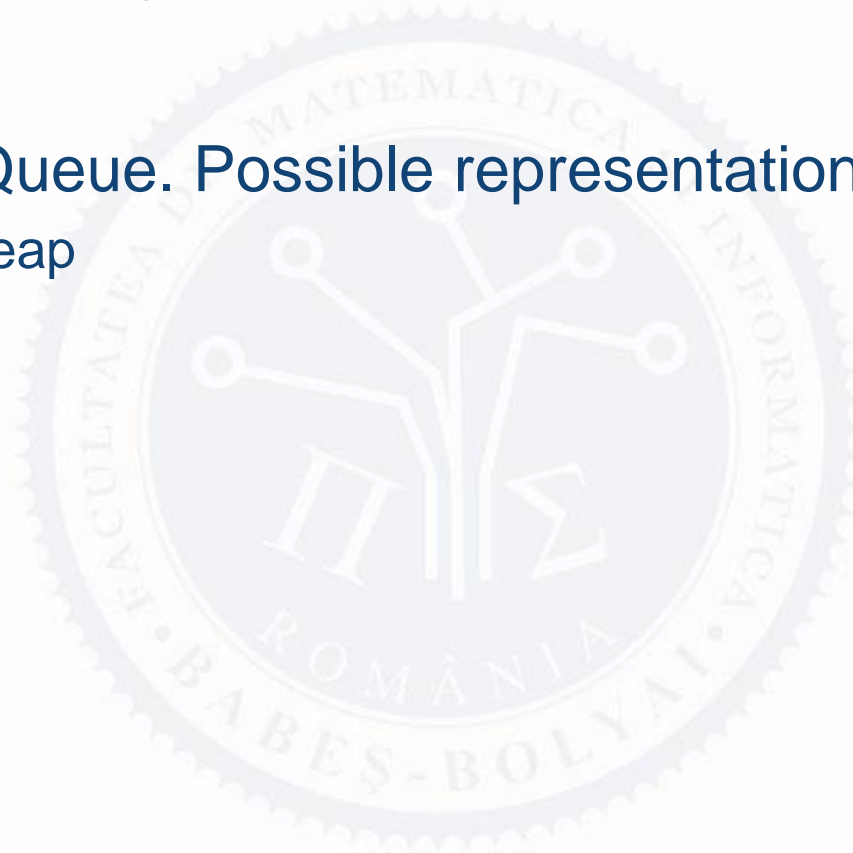
## Lecture – week 8

- Direct Access Table
- Hash Function , Hash Table  
& Collision Resolution Methods

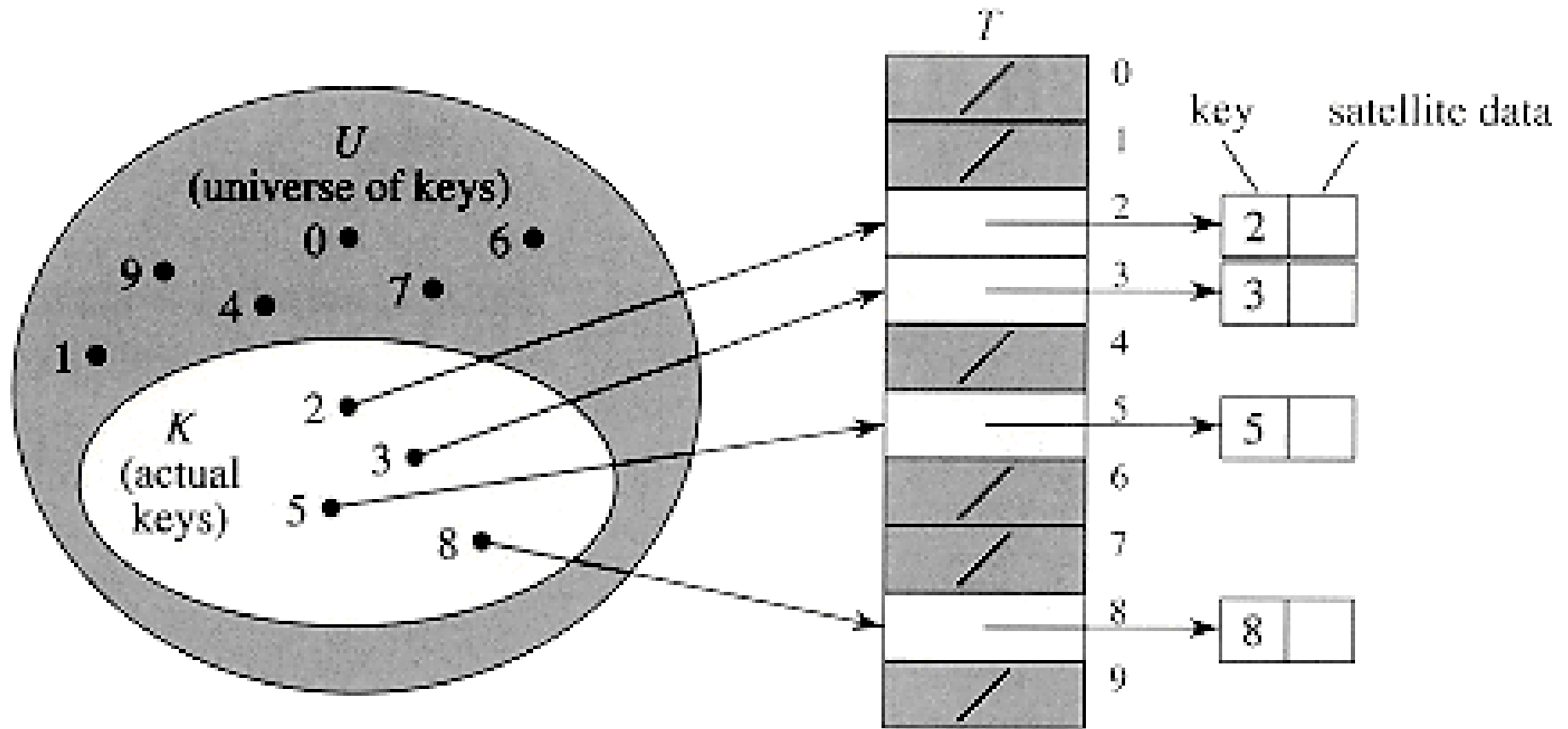
Lect. PhD. Lupsa Dana  
Babes - Bolyai University  
Computer Science and Mathematics Faculty  
2022 - 2023

## Previously, in Lecture 6

- Priority Queue. Possible representation
  - Binary Heap
- Heap-sort



# Direct address table



Source: Cormen

# Direct address table

## *Problem:*

- We have data where every element has a key (a natural number).
- The universe of keys (the possible values for the keys) is relatively small,  $U = \{ 0 ; 1 ; 2 ; \dots ; m-1 \}$
- No two elements have the same key
- We have to support the basic dictionary operations: INSERT, DELETE and SEARCH

## *Idea:*

- Use an array that has one position for every possible key

*Applicable: when we can afford to ...*

the universe  $U$  of keys is reasonably small

# Direct address table

## *Operations:*

```
function search(T, k) is:  
    search  $\leftarrow$  T[k]  
end-function
```

```
subalgorithm insert(T, x) is:  
    T[key(x)]  $\leftarrow$  x  
end-subalgorithm
```

```
subalgorithm delete(T, x) is:  
    T[key(x)]  $\leftarrow$  NIL  
end-subalgorithm
```

## *Notations:*

T is an array (the direct-address table), k is a key

x is an element

key(x) returns the key of an element

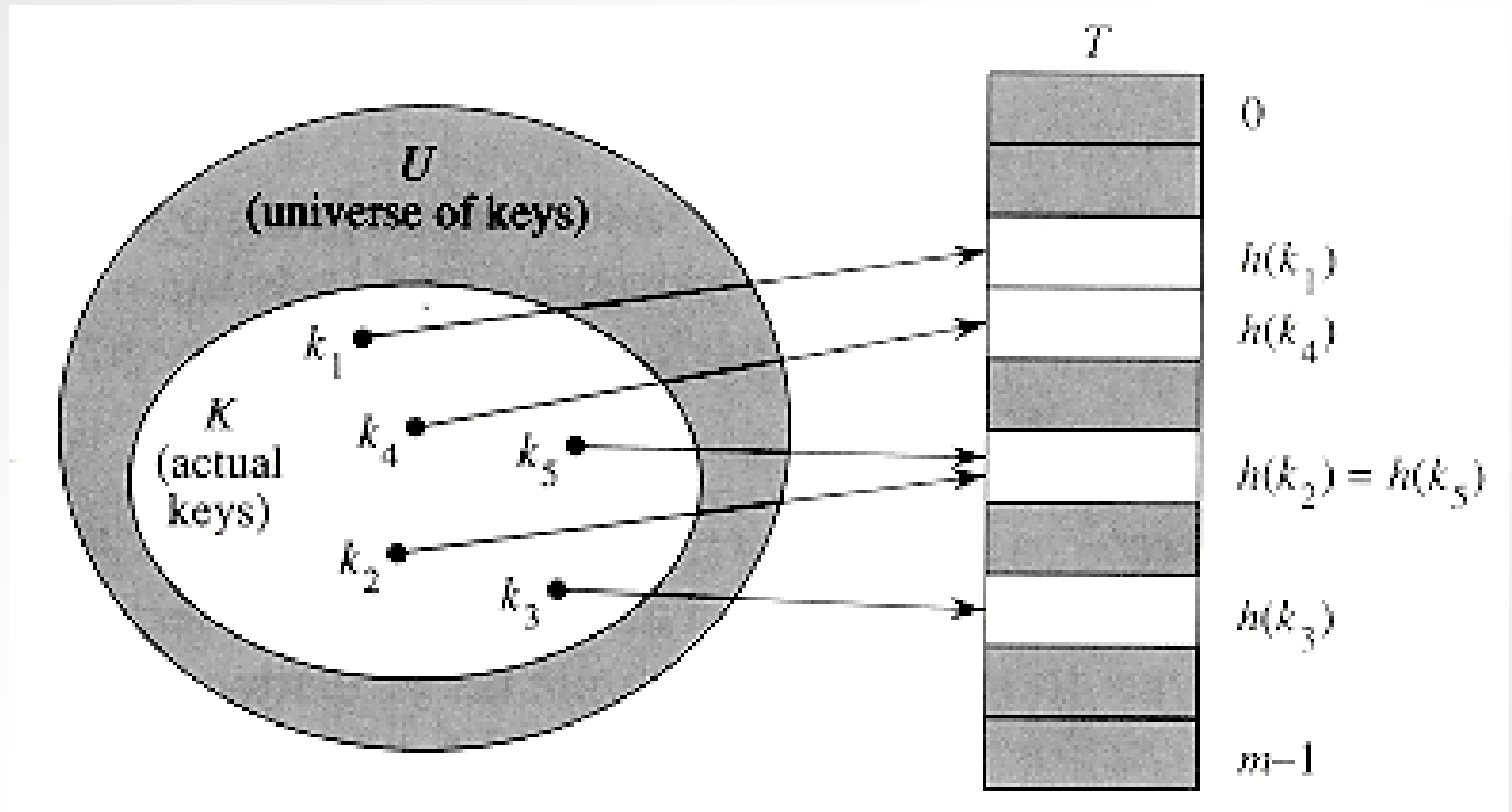
***Complexity: ... ?***

# Direct address table

*Think about:*

- Assume that we have a direct address  $T$  of length  $m$ . How can we find the maximum element of the direct-address table? What is the complexity of the operation?

# Hash table



Source: Cormen

# Collisions

- When two keys,  $x$  and  $y$ , have the same value for the hash function  $h(x) = h(y)$  we have a collision.
- Which hash function ?

Birthday  
paradox

## Collision resolution methods:

- Separate chaining
- Coalesced chaining
- Open addressing



# Hash function

- perfect hash function
    - injective: maps distinct elements with no collisions
    - it is too expensive to compute it for every input
- build a hash function to minimize collisions

## good hash function

In practice:

- use heuristic information to create a hash function that is likely to perform well

Choose between:

- simple and fast, but have a high number of collisions;
- more complex functions, with better quality, but take more time to calculate

# Hash function

A good hash function:

- is deterministic
- can be computed in  $\Theta(1)$  time
- satisfies (approximately) the assumption of **simple uniform hashing**: each key is equally likely to hash to any of the  $m$  slots, independently of where any other key has hashed to:

$$P(h(k) = j) = \frac{1}{m} \quad \forall j = 0, \dots, m-1 \quad \forall k \in U$$

- 
- The simple uniform hashing theorem is hard to satisfy, especially when we do not know the distribution of data. Data does not always have a uniform distribution.
  - In practice we use heuristic techniques to create hash functions that perform well.
  - Most hash functions assume that the keys are natural numbers. If this is not true, they have to be interpreted as natural number.

# Hash functions

## Bad hash functions

- $h(k) = \text{constant number}$
- assuming that the keys are CNP numbers:  
    assume  $m = 100$  and you use the birth day from the CNP  
    (as a number):  
     $h(\text{CNP}) = \text{birthdayyear} \% 100$

Data does not always have a uniform distribution.

e.g.: dates, group numbers at our faculty, letter of an English word

# Good hash function

Need: qualitative information about P

Assume: **uniform distributed keys**

Example:

- keys are random real numbers independently and uniformly distributed in the range  $[0,1)$ .

$$h(k) = [k * m]$$

satisfies the simple uniform hashing property

- keys are random integers independently and uniformly distributed in the range 0 to  $N-1$

where  $N$  much larger than  $m$

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

satisfies the simple uniform hashing property

# Hash function

## The division method

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

- Experiments show that good values for  $m$  are primes not too close to exact powers of 2

# Hash function

## Mid-square method

- For getting the hash of a number, multiply it by itself and take the middle  $r$  digits.

e.g.:

Assume that the table size is  $10^r$ , for example  $m = 100$  ( $r = 2$ )

$$\begin{aligned} h(4567) &= \text{middle 2 digits of } 4567 * 4567 \\ &= \text{middle 2 digits of } 20857489 = 57 \end{aligned}$$

- Same thing for:  $m = 2^r$   
and the binary representation of the numbers

e.g.:

$$\begin{aligned} m &= 2^4, \\ h(1011) &= \text{middle 4 digits of } 01111001 = 1110 \end{aligned}$$

# Hash function

## The multiplication method

$$h(k) = \text{floor}(m * \text{frac}(k * A))$$

where

$m$  - hash table size

$A$  - constant in the range  $0 < A < 1$

Remark:

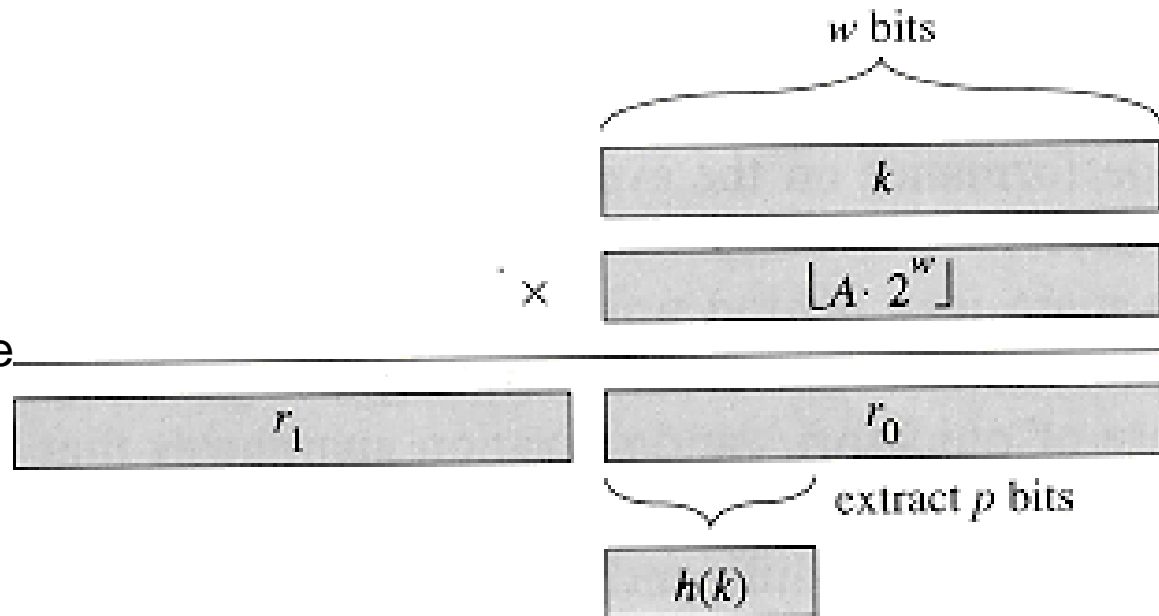
- the value of  $m$  is not critical

source: Cormen

$m = 2^p$  for some integer  $p$

➤ implementation:

restrict  $A$  to  $w$  bits,  
 $w$ =machine word size



# Hash function

## The multiplication method

*good value for A  
(experimental)*

$$A \approx \frac{\sqrt{5} - 1}{2} \approx 0.6180339887$$

**Donald Knuth**, *The Art of Computer Programming* , **1968**

Numeric example:

$$k = 123456$$

$$m = 10000$$

$$A = 0.6180339887...$$

$$h(k) = \text{floor}(41.151...) = 41$$

$$k = 50$$

$$h(k) = 9016$$



# Hash function

	Multiplication Method	Division Method
m	1000	1000
A	0.618033988749895	
key	$h(key) = \text{floor}(m * \text{frac}(key * A))$	$h(key) = key \bmod 1000$
123456	4	456
123459	858	459
123496	725	496
123956	21	956
129456	208	456
193456	383	456
923456	195	456

# Universal hashing

- Instead of having one hash function, we have a collection  $\mathcal{H}$  of hash functions that map a given universe  $U$  of keys into the range  $\{0, 1, \dots, m-1\}$
- Such a collection is said to be **universal** if for each pair of distinct keys  $x, y \in U$  the number of hash functions from  $\mathcal{H}$  for which  $h(x) = h(y)$  is precisely:  $|\mathcal{H}| / m$
- In other words, with a hash function randomly chosen from  $\mathcal{H}$  the chance of collision between  $x$  and  $y$ , where  $x \neq y$ , is exactly  $1/m$

# Universal hashing

## Example 1

Fix a prime number  $p > \text{the maximum possible value for a key from } U$ .

For every  $a \in \{1, \dots, p-1\}$  and  $b \in \{0, \dots, p-1\}$  we can define a hash function  $h_{a,b}(k) = ((a * k + b) \bmod p) \bmod m$ .

## Example 2

If the key  $k$  is an array  $\langle k_1, k_2, \dots, k_r \rangle$  such that  $k_i < m$  (or it can be transformed into such an array, by writing the  $k$  as a number in base  $m$ ).

Let  $\langle x_1, x_2, \dots, x_r \rangle$  be a fixed sequence of random numbers, such that  $x_i \in \{0, \dots, m-1\}$  (another number in base  $m$  with the same length).

$$h(k) = \sum_{i=1}^r k_i * x_i \bmod m$$

*How many hash functions in collection?*

# Universal hashing

## Example 3

Suppose the keys are  $u$  – bits long and  $m = 2^b$ .

Pick a random  $b \times b \times u$  matrix (called  $h$ ) with 0 and 1 values only.

Pick  $h(k) = h * k$  where in the multiplication we do addition mod 2.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$

# Hash function

What can we do if keys are not natural numbers?

- Define special hash functions that work with your keys

e.g.: for  $k$  - real nr. from  $[0,1)$  use:  $h(k) = [k * m]$

- Use a **hashCode** function that transforms the key to a natural number

# Hash function: transforming the key into a natural number

e.g.:

If the key is a string  $s$ :

- we can consider the ASCII codes for every letter
- we can use 1 for a, 2 for b, etc.

**hashCode:**  $s[0] + s[1] + \dots + s[n - 1]$

But: Anagrams have the same sum (see: SAUCE and CAUSE)

- Assuming maximum length of 10 for a word (and the second letter representation), hashCode values range from 1 (the word a) to 260 (zzzzzzzzzz). Considering a dictionary of about 50,000 words, how many words would we have for a hashCode value, on average?

**hashCode:**  $s[0] * 26^{n-1} + s[1] * 26^{n-2} + \dots + s[n - 1]$

where  $n$  - the length of the string

- Instead of 26 (which was chosen since we have 26 letters) we can use a prime number as well (Java uses 31, for example).

# Hash function when keys are not uniformly distributed

Need: qualitative information about P

## Special-purpose hash function

- exceptionally good for a specific kind of data  
no performance on data with different distribution

Example:

input data: file names such as FILE0000.CHK, FILE0001.CHK, FILE0002.CHK, etc., with mostly sequential numbers.

- extracts the numeric part  $k$  of the file name  $fn$   
 $h(fn) = \text{numeric\_part}(fn) \bmod m$

Example:

input data: text in any natural language

has highly non-uniform distributions of characters, and character pairs, very characteristic of the language

- it is prudent to use a hash function that depends on all characters of the string—and depends on each character in a different way

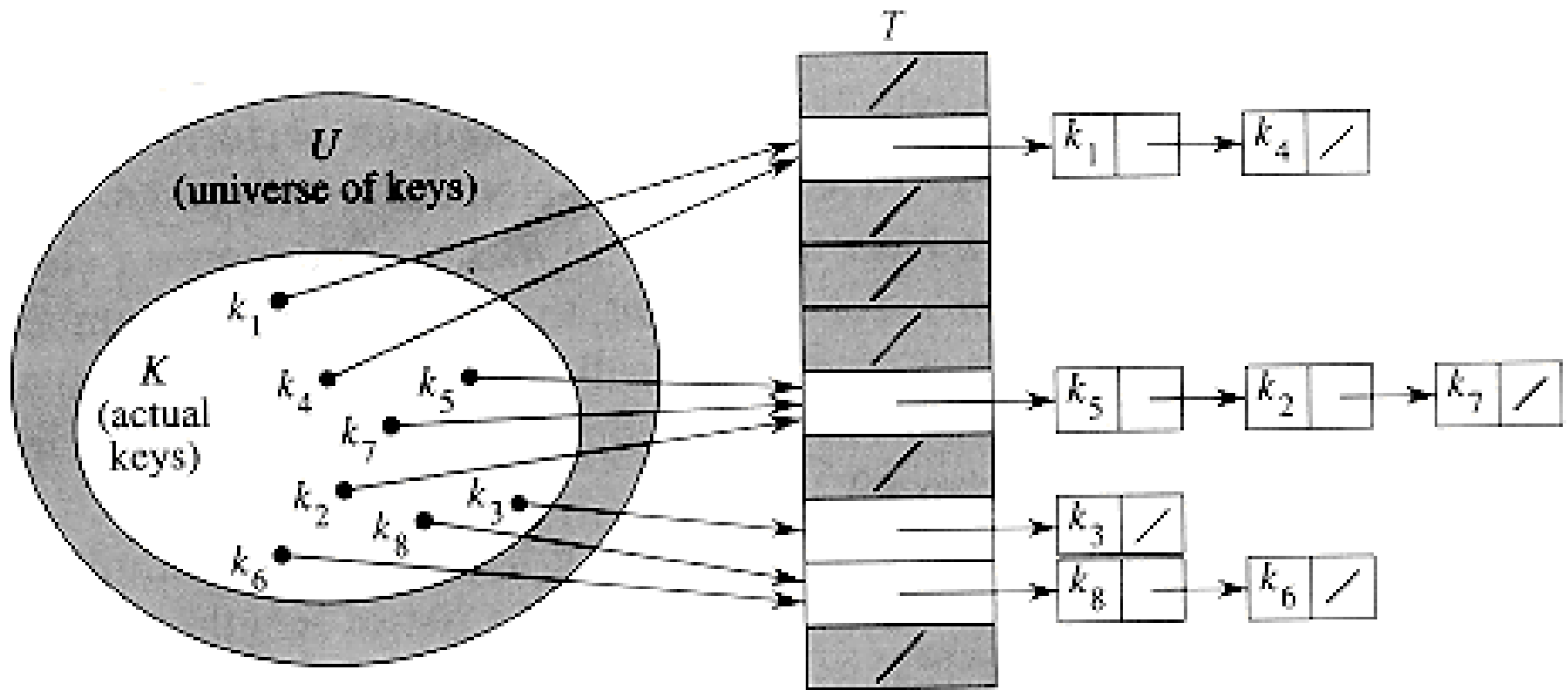
# Hash function

“ The design of the "hash function" is somewhat of a black art. It can be highly influenced by the data that's intended to be stored in the hashing-based data structure, so the discussion on a sensible hashing function can often stray into a discussion about specific inputs. “

(someone on stackoverflow...)



# Collision resolution: separate chaining



By default, in these classes, separate chaining will use an SLL.

# Collision resolution: separate chaining

## *Representation:*

Node:

key: TKey  
next:  $\uparrow$  Node

HashTable:

T: ( $\uparrow$  Node) []  
m: Integer  
h: TFunction

## *Operations:*

**insert( $T, x$ )**

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

**search( $T, k$ )**

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

**delete( $T, x$ )**

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

Remark: By default, we will consider having only keys.

# Analysis of hashing with chaining

The average performance depends on how well the hash function  $h$  can distribute the keys to be stored among the  $m$  slots.

## Assumptions:

- the hash value can be computed in constant time
- hash function satisfy **Simple Uniform Hashing Assumption**

**Load factor  $\alpha$**  of table  $T$  with  $m$  slots containing  $n$  elements is  $n/m$

- represents the average number of elements stored in a chain

## Analysis of hashing with chaining

- Insert      WC:  $\Theta(1)$
- Search
- Delete

# Analysis of hashing with chaining: Search

## Theorem:

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

## Theorem:

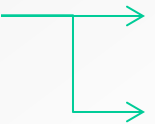
In a hash table in which collisions are resolved by chaining, a successful search takes  $(1 + \alpha)$ , on the average, under the assumption of simple uniform hashing.

- Proof idea: (1) is needed to compute the value of the hash function and  $\alpha$  is the average time needed to search one of the  $m$  lists

**Think about:** WC for search

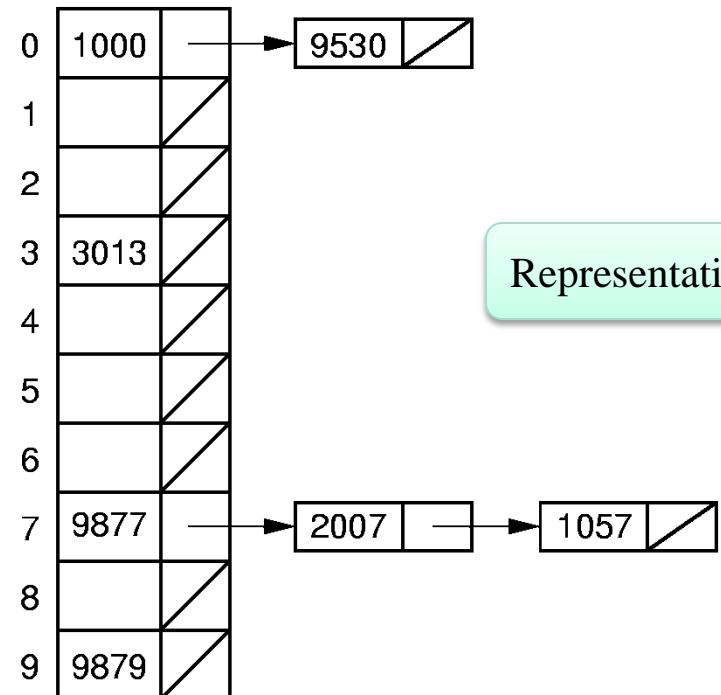
Can we do better than by using SLL?

# Separate chaining. Variations

- Use another data structure instead of a SLL  
e.g.: use balanced tree
- Elements:  stored external to the table  
stored in the table itself

## Example:

a hash table where  
each location stores one record  
and a link pointer  
to the rest of the list.



# Think about:

Assume we have a hash table with  $m = 6$  that uses separate chaining for collision resolution, with the following policy: if the load factor of the table after an insertion is greater than or equal to 0.7, we double the size of the table

- Using the division method, insert the following elements, in the given order, in the hash table: 38, 11, 8, 72, 57, 29, 2.
- 

Iterator for a hash table with separate chaining

- How can we implement the *init* operation? BC, WC ?
- How can we implement the *getCurrent* operation?
- How can we implement the *next* operation?
- How can we implement the *valid* operation?

# Think about:

- Suppose we use a hash function  $h$  to hash  $n$  distinct keys in a table  $T$  of size  $m$ .  
What is the average number of collisions?  
(the probable cardinal of  $\{(x,y) \in TKey \times TKey : d(x)=d(y)\}$ )
- Show that if  $|U| > n \cdot m$ , then there is a subset of  $U$  of size  $n$  containing keys that all hashes to the same slot, so that the search time, in the worst case, is  $\Theta(n)$ .
- Suppose we use a hash table with separate chaining, but each list is sorted by key.  
What is the time-complexity for search (successfully, unsuccessfully), add, and delete?
- Can we define a sorted container on a hash table with separate chaining?

# Direct address & hash table

## *Think about:*

Finding the maximum key

- Assume that we have  $n$  keys in a direct address  $T$  of length  $m$ . How can we find the maximum key of the direct-address table? What is the complexity of the operation?
- How does the operation for finding the maximum change if we have a hash table, instead of a direct-address table (consider collision resolution by separate chaining)?



# Hash table in programming languages

- Hash tables are an effective way to implement containers if we want to achieve good performance for **search**, **insert** and **delete**

## JAVA Util

- HashSet
- HashMap

## C++ STL

- unordered\_set
- unordered\_map

They use hash table with separate chaining.

# Hash table in programming languages

## Java.util

### HashSet

- implements the Set interface; uses a hash table

### HashMap

- implements the Map interface; uses a hash table
- `HashMap()`  
Constructs an empty `HashMap` with the default initial capacity (16) and the default load factor (0.75).
- `HashMap(int initialCapacity)`  
Constructs an empty `HashMap` with the specified initial capacity and the default load factor (0.75).
- `HashMap(int initialCapacity, float loadFactor)`  
Constructs an empty `HashMap` with the specified initial capacity and load factor.

# Hash in programming languages. Examples

## Java Object

- `public int hashCode()`

*As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java™ programming language.)*

- `public boolean equals(Object obj)`

*If two objects are equal then they must return same hash code  
( when they are compared by equals() )*