Lecture 8

- Direct Access Table
- Hash Function & Hash Table

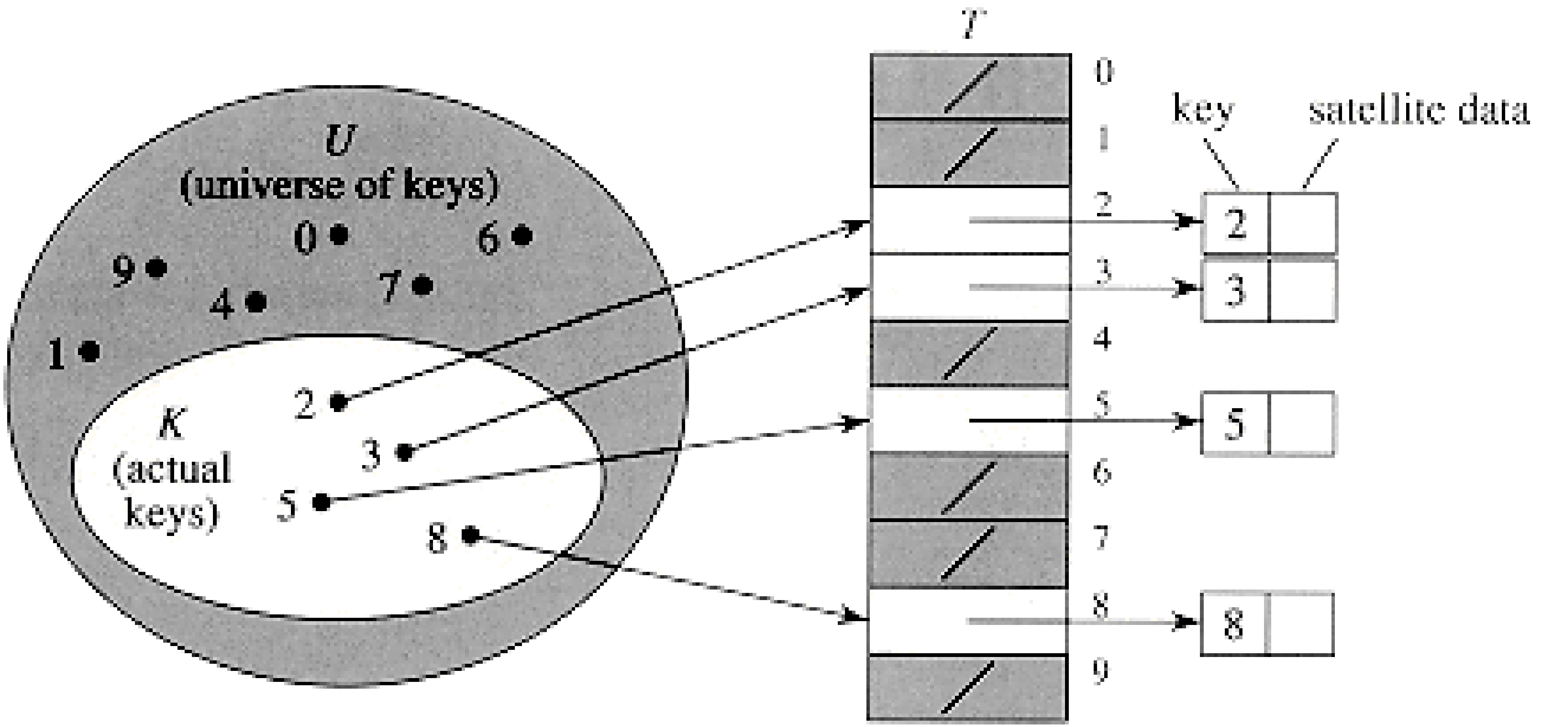
Lect. PhD. Lupsa Dana
Babes - Bolyai University
Computer Science and Mathematics Faculty
2021 - 2022

Some slides borrowed from: Lect. PhD. Onet-Marian Zsuzsanna

Previously, in Lecture 7

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

Direct address table



Source: Cormen

Direct address table

Operations:

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

```
function search(T, k) is:
            search ← T[k]
end-function
```

```
subalgorithm insert(T, x) is:
            T[key(x)] ← x
end-subalgorithm
```

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

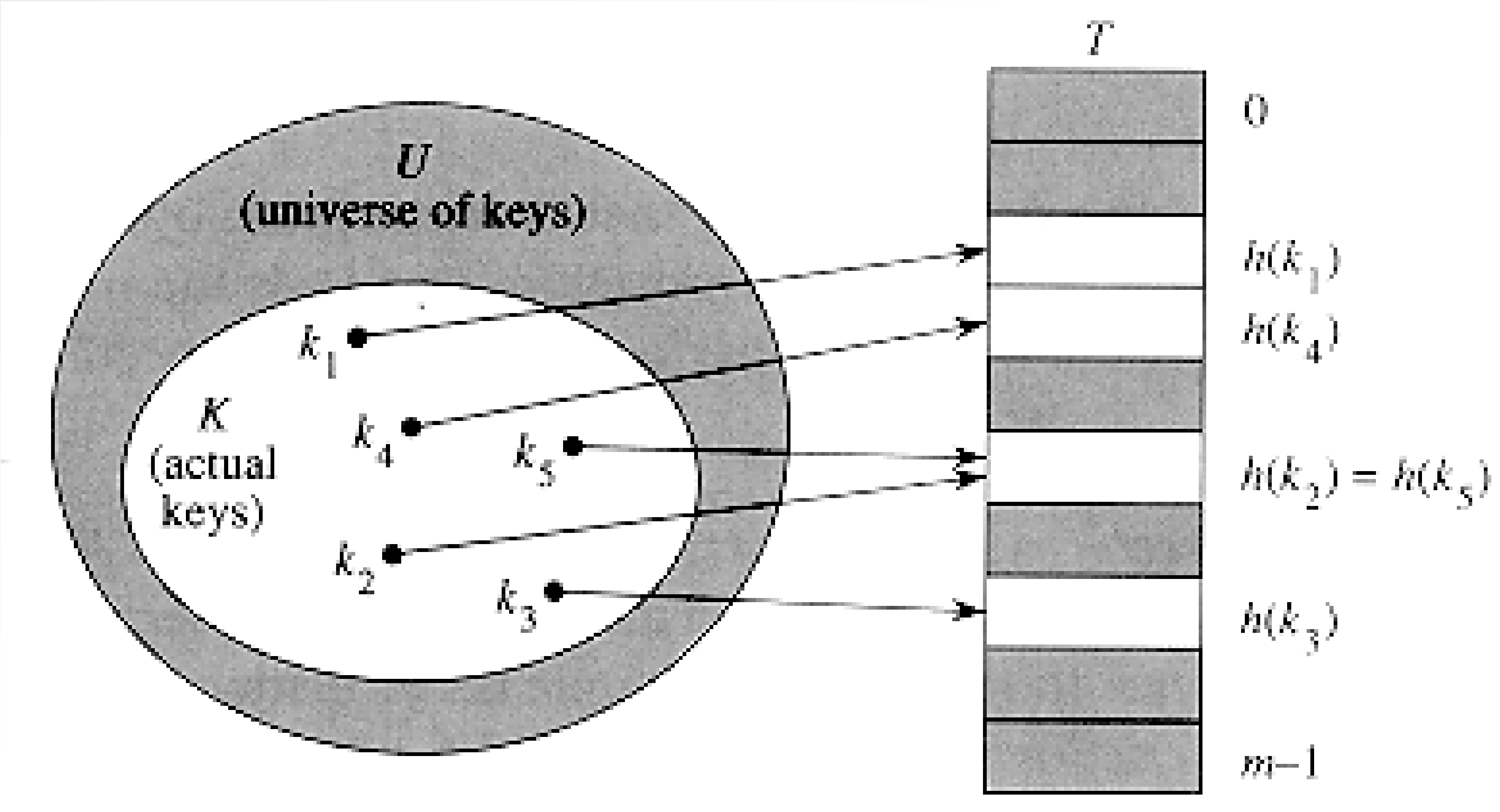
Complexity: … ?

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:

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

- ➢ 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) = 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(CNP) = 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).

  $h(k) = [k * \boldsymbol{m}]$

  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

  $h(k) = k \bmod \boldsymbol{m}$

  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)

h(4567) = middle 2 digits of 4567 * 4567

= middle 2 digits of 20857489 = 57

- Same thing for: $m = 2^r$

and the binary representation of the numbers

e.g.:

$m = 2^4$,

h(1011) = middle 4 digits of 01111001 = 1110

Hash function

The multiplication method

h(k) = floor(m * frac(k * A))

where

m - hash table size

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

Remark:

• the value of *m* is not critical

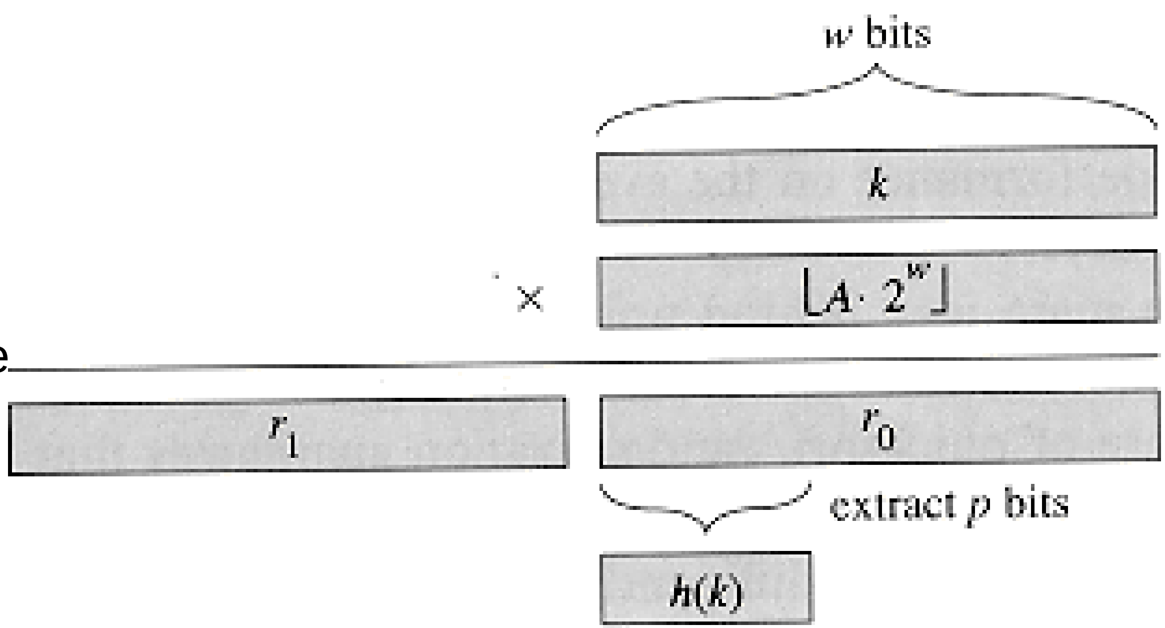
source: Cormen

$m = 2^p$ 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**

<u>Numeric example</u>:

$k = 123456$

$m = 10000$

$A = 0.6180339887...$

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

$k = 50$

$h(k) = 9016$

Hash function

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

the value of *m* is not critical

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, … ,.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 $p >$ *the maximum possible value for a key from* U.
For every $a \in \{1, \ldots, p-1\}$ and $b \in \{0, \ldots, p-1\}$ we can define a hash function $h_{a,b}(k) = ((a * k + b) \ mod \ p) \ mod \ m$.

Example 2

If the key k is an array $< k_1, k_2, \ldots, k_r >$ 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 $< x_1, x_2, \ldots, x_r >$ be a fixed sequence of random numbers, such that $x_i \in \{0, \ldots, m-1\}$ (another number in base m with the same length).
$h(k) = \sum_{i=1}^{r} k_i * x_i \ \ mod \ m$

How many hash functions in collection?

Universal hashing

•

Example 3

Suppose the keys are $u - bits$ long and $m = 2^b$.
Pick a random $b - by - u$ 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

Hash function when keys that are not natural numbers

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

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

not uniformly distributed keys

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**) = numeric_part(**fn**) mod m*

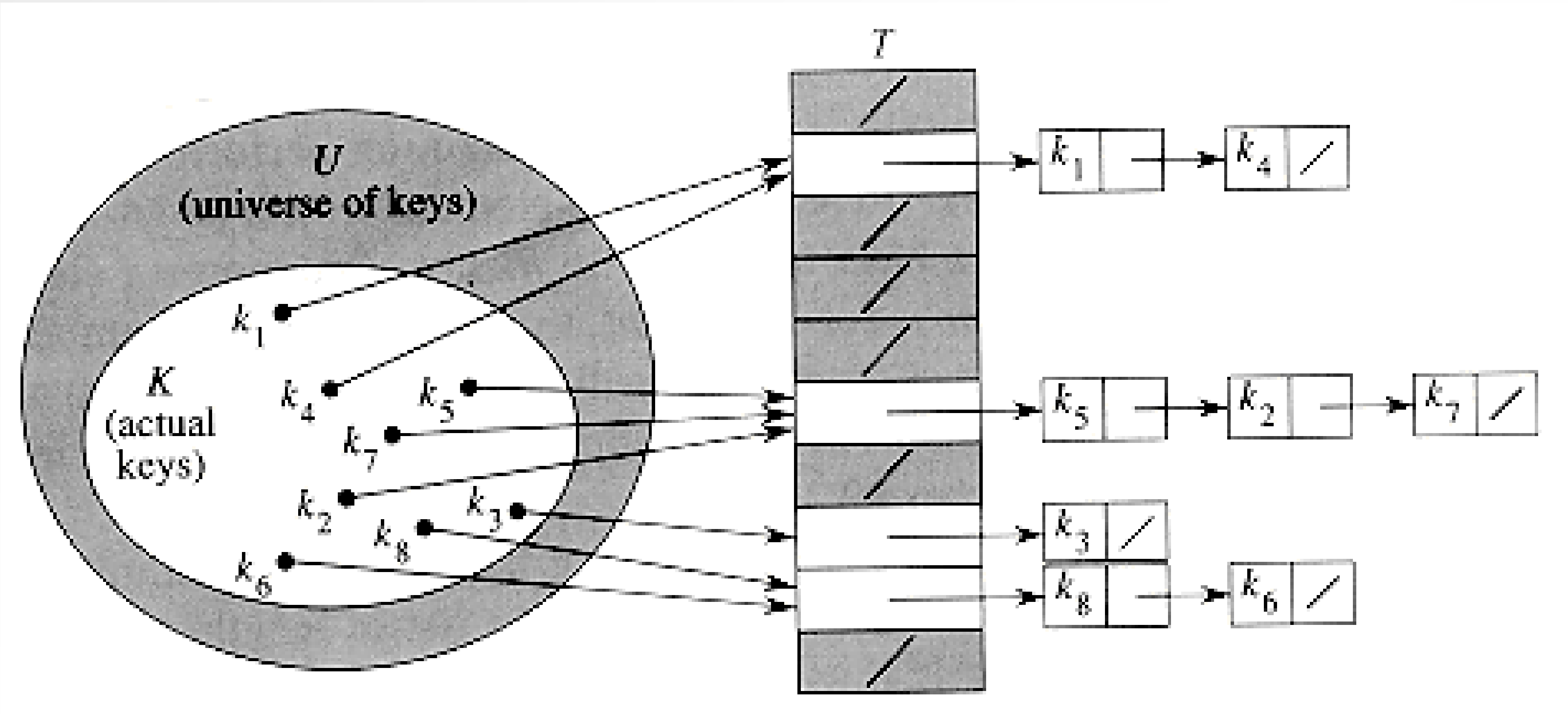
Example:

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

Separate chaining



Representation ?

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

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

We assume that:

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

Load factor *α* 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

All dictionary operations can be supported in $\Theta(1)$ time on average.

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 α is the average time needed to search one of the m lists

WC for search?
Can we do better ?

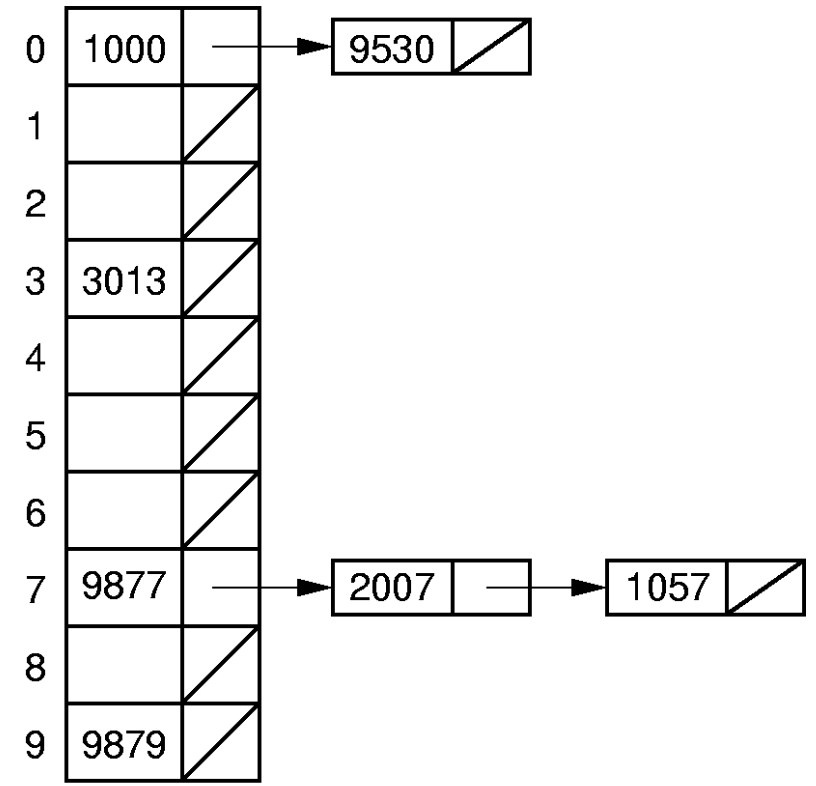
Separate chaining. Variations

- Use a balanced tree instead of a SLL
- 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.

Representation ?



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: <u>38</u>, 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 random 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 $|\boldsymbol{U}| > \boldsymbol{n} \cdot \boldsymbol{m}$, 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 $\boldsymbol{\Theta}(\boldsymbol{n})$.

- 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?