



# Theory of algorithms (4th lecture)

Pál Pusztai  
[pusztai@sze.hu](mailto:pusztai@sze.hu)

# Outline

- Hash tables
  - Direct-address tables
  - Hash functions
  - Open addressing
- Exercises



## Hash tables

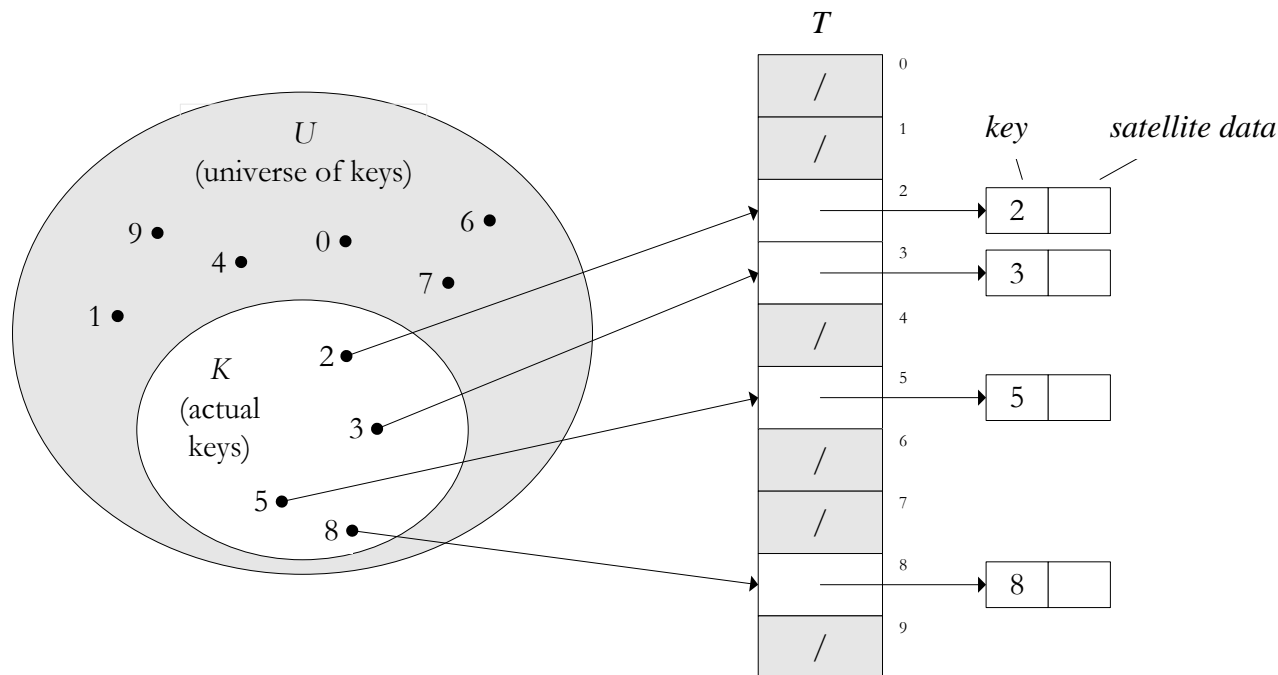
- How can we implement a dynamic set with arrays that supports only the dictionary operations INSERT, SEARCH, and DELETE?
- Solutions
  - With data maintenance
    - Unsorted data: INSERT:  $O(1)$ , SEARCH:  $O(n)$ , DELETE:  $O(1)$
    - Sorted data: INSERT:  $O(n)$ , SEARCH:  $O(\lg n)$ , DELETE:  $O(n)$
  - Direct-address tables
    - **Criteria:** Each element has unique key from the universe  $U = \{0, 1, 2, \dots, m-1\}$ , where  $m$  is not too large.
    - **Efficiency:** All operations:  $O(1)$
  - Hash tables
    - **Criteria:** The number of keys actually stored is small relative to the total number of possible keys.
    - **Solution:** An array of size proportional to the number of keys actually stored is used, and the index is **computed** with hash function from the keys.
    - **Efficiency:** All operations:  $O(1)$  on the **average**.



## Direct-address tables

**Criteria:** Each element has unique key from the universe  $U = \{0, 1, 2, \dots, m-1\}$ , where  $m$  is not too large.

**Solution:** An array, or **direct-address table**, denoted by  $T[0..m-1]$ , in which each position, or **slot**, corresponds to a key in the universe  $U$ .



Dynamic set representation with direct-address table  $T$

## Direct-address tables

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     $T[x.key] = \text{NIL}$

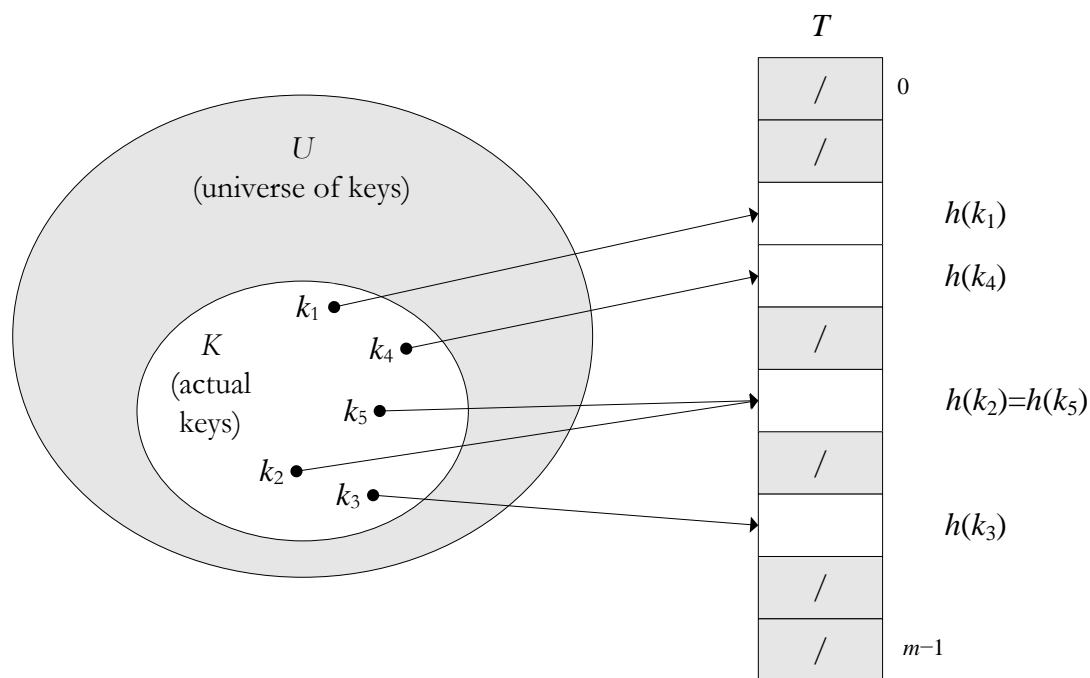
**Efficiency:** Each of these operations takes only  $O(1)$  time.

# Hash tables

**Hash function:**  $h : U \rightarrow \{0, 1, \dots, m-1\}$

**Solution:** An array  $T[0..m-1]$ , called **hash table**, where  $m$  is typically much less than  $|U|$ . An element with key  $k$  **hashes** to slot  $h(k)$  (in other words  $h(k)$  is the **hash value** of key  $k$ ).

**Problem:** More than one key map to the same slot, they **collide**.



Using a hash function  $h$  to map keys to hash-table slots

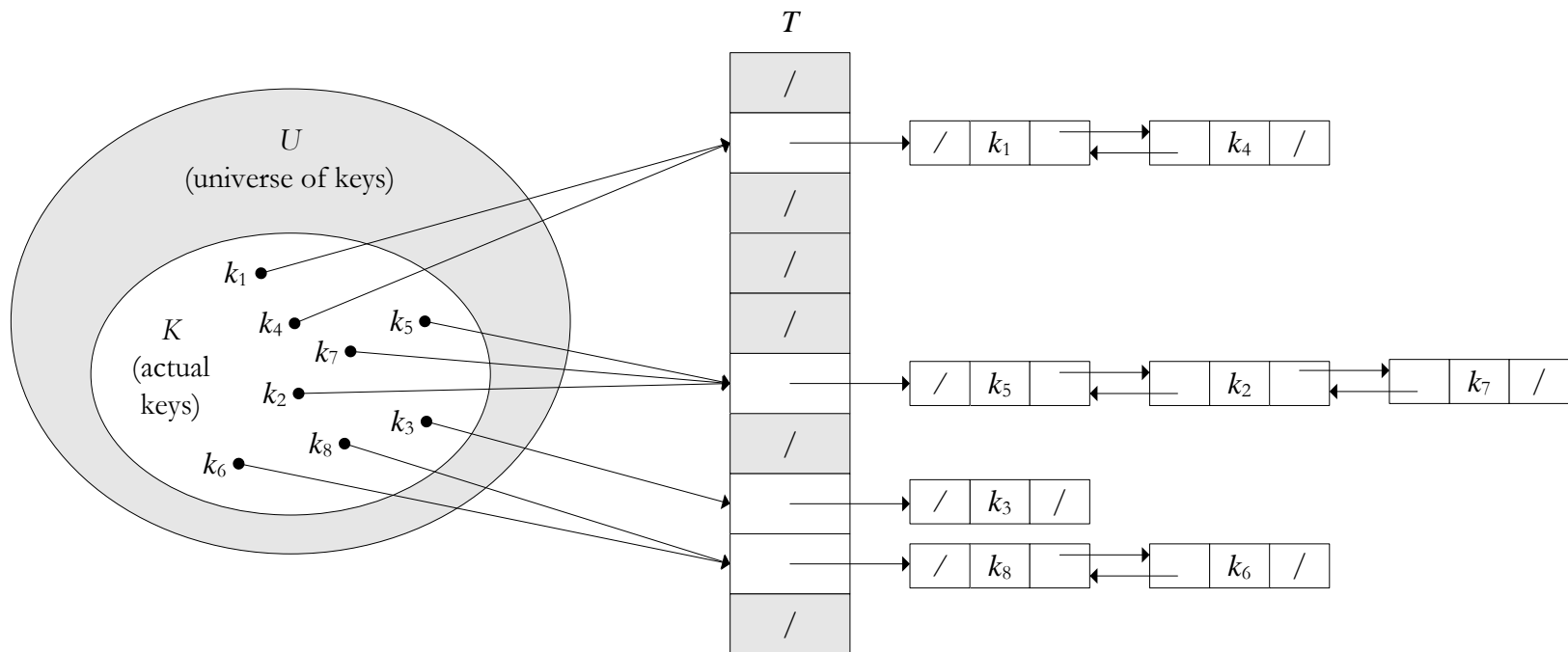
# Hash functions

- Keys and hashing
  - Universe of keys
    - The universe of keys usually is the set  $N = \{0, 1, 2, \dots\}$  of natural numbers.
  - Simply uniform hashing
    - Each key is equally likely to hash to any of the  $m$  slots, independently of where any other key has hashed to.
    - A good hash function satisfies (approximately) the assumption of simply uniform hashing.
- Create hash functions
  - The division method:  $h(k) = k \bmod m$ 
    - A prime not too close to an exact power of 2 is often a good choice for  $m$ .
    - For example: if  $n=2000$ , we could choose  $m=701$  because it is a prime near  $2000/3$  but not near any power of 2, we examine an average 3 elements in an unsuccessful search.
  - The multiplication method:  $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$ 
    - $A$  is a constant in the range  $0 < A < 1$ .
    - The value of  $m$  is not critical, usually it is a power of 2.
  - Universal hashing
    - We choose the hash function randomly from a finite collection of hash functions.



# Hash tables

**Collision resolution by chaining:** Each hash-table slot  $T[j]$  contains a linked list of all the keys whose hash value is  $j$ . The linked list can be either singly or doubly linked. The figure below shows doubly linked lists because deletion is faster that way.



**Collision resolution by chaining**



## Hash tables

CHAINED-HASH-SEARCH( $T, k$ )

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

CHAINED-HASH-INSERT( $T, x$ )

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

CHAINED-HASH-DELETE( $T, x$ )

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

**Efficiency:** Insertion is  $O(1)$  time (because it assumes that the element  $x$  being inserted is not already present in the table). Deletion is  $O(1)$  time, if the lists are doubly linked. In the worst case of search and deletion from a single linked list is proportional to the length of the list.

Using **simple uniform hashing** and the number of slots  $m$  is proportional to the number of keys actually stored, the **average** running time of searching is  $O(1)$ .

## Exercises

- Illustrate the operation of CHAINED-HASH-INSERT with below given keys. Give the hash table  $T$  after insertion of all keys in order.  $T$  is empty at the beginning,  $m = 9$ ,  $h(k) = k \bmod m$ .
  - $\langle 5, 28, 19, 15, 20, 33, 12, 17, 10 \rangle$



## Open addressing

### ■ Open addressing

- It is another way to handle collisions.
- All elements occupy the hash table itself, thus the hash table can “fill up”.
- We systematically examined, or **probe**, the slots of the hash table.
- The sequence of **positions probed** depends upon the **key** being inserted and the **probe number**.
  - The **hash function**:  $h : U \times \{0, 1, \dots, m-1\} \rightarrow \{0, 1, \dots, m-1\}$
  - **Criteria**: For every  $k$  key, the **probe sequence**  $\langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$  be a permutation of  $\langle 0, 1, \dots, m-1 \rangle$ , so that every hash-table position is eventually considered as a slot.
  - **Uniform hashing**: The probe sequence  $\langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$  of each key  $k$  is equally likely to be any of the  $m!$  permutation of  $\langle 0, 1, \dots, m-1 \rangle$ .

### ■ Remark

- If we have to delete elements then we use chaining instead of open addressing.



## Open addressing

HASH-INSERT( $T, k$ )

```
1   $i = 0$ 
2  repeat
3       $j = h(k, i)$ 
4      if  $T[j] == \text{NIL}$ 
5           $T[j] = k$ 
6          return  $j$ 
7      else
8           $i = i + 1$ 
9  until  $i == m$ 
10 error „hash table overflow”
```

**Remark:** It is assumed that the elements in the hash table  $T$  are keys with no satellite information. The key  $k$  is identical to the element containing key  $k$ .

## Open addressing

HASH-SEARCH( $T, k$ )

```
1   $i = 0$ 
2  repeat
3       $j = h(k, i)$ 
4      if  $T[j] == k$ 
5          return  $j$ 
6       $i = i + 1$ 
7  until  $T[j] == \text{NIL}$  or  $i == m$ 
8  return NIL
```

**Efficiency:** Given an open-address hash table with **load factor**  $\alpha = n/m < 1$ , assuming uniform hashing, the **expected** number of probes in an unsuccessful search is at most  $1/(1-\alpha)$ , and in a successful search is at most  $(1/\alpha) \ln(1/(1-\alpha))$ . For example:  $\alpha=1/2$ , 2 and 1.38;  $\alpha=9/10$ , 10 and 2.55.



# Open addressing

## ■ Open addressing

### ■ Assumption

- Let  $h': U \rightarrow \{0, 1, \dots, m-1\}$  be an ordinary hash function and  $i = 0, 1, \dots, m-1$ .

### ■ Linear probing

- $h(k, i) = (h'(k) + i) \bmod m$
- There are only  $m$  distinct probe sequences. Long runs of occupied slots build up, increasing the average search time.

### ■ Quadratic probing

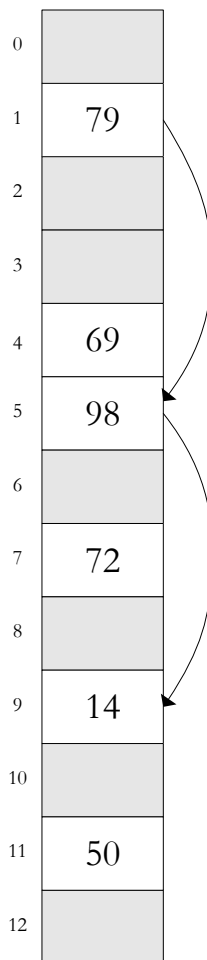
- $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$
- There are two positive auxiliary constants  $c_1$  and  $c_2$ . To make full use of the hash table, the values of  $c_1$ ,  $c_2$ , and  $m$  are constrained. As in linear probing, the initial probe determines the entire sequence, only  $m$  distinct probe sequences are used.

### ■ Double hashing

- $h(k, i) = (h_1(k) + i h_2(k)) \bmod m$
- There are two auxiliary hash functions  $h_1$  and  $h_2$ . The value  $h_2(k)$  must be relatively prime to the hash-table size  $m$  for the entire hash table to be searched. In double hashing  $m^2$  different probe sequences are used (but the "ideal" scheme of uniform hashing uses  $m!$ ).



# Open addressing



$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$$

$$m = 13$$

$$h_1(k) = k \bmod 13$$

$$h_2(k) = 1 + (k \bmod 11)$$

**Insertion by double hashing**



## Exercises

- Why does deleting from an open-address hash table cause problem?
- Illustrate the result of inserting below given keys (in order) into an empty hash table using with the open addressing methods.
  - $\langle 10, 22, 31, 4, 15, 28, 17, 88, 59 \rangle$
- Linear probing ( $m=11$ )
  - $h(k, i) = (h'(k) + i) \bmod m$
  - $h'(k) = k \bmod m$
- Quadratic probing ( $m=9, c_1=1, c_2=3$ )
  - $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$
  - $h'(k) = k \bmod m$
- Double hashing ( $m=11$ )
  - $h(k, i) = (h_1(k) + i h_2(k)) \bmod m$
  - $h_1(k) = k \bmod m$
  - $h_2(k) = 1 + (k \bmod (m-1))$

