Hash Tables

Heejin Park

Division of Computer Science and Engineering

Hanyang University

Contents

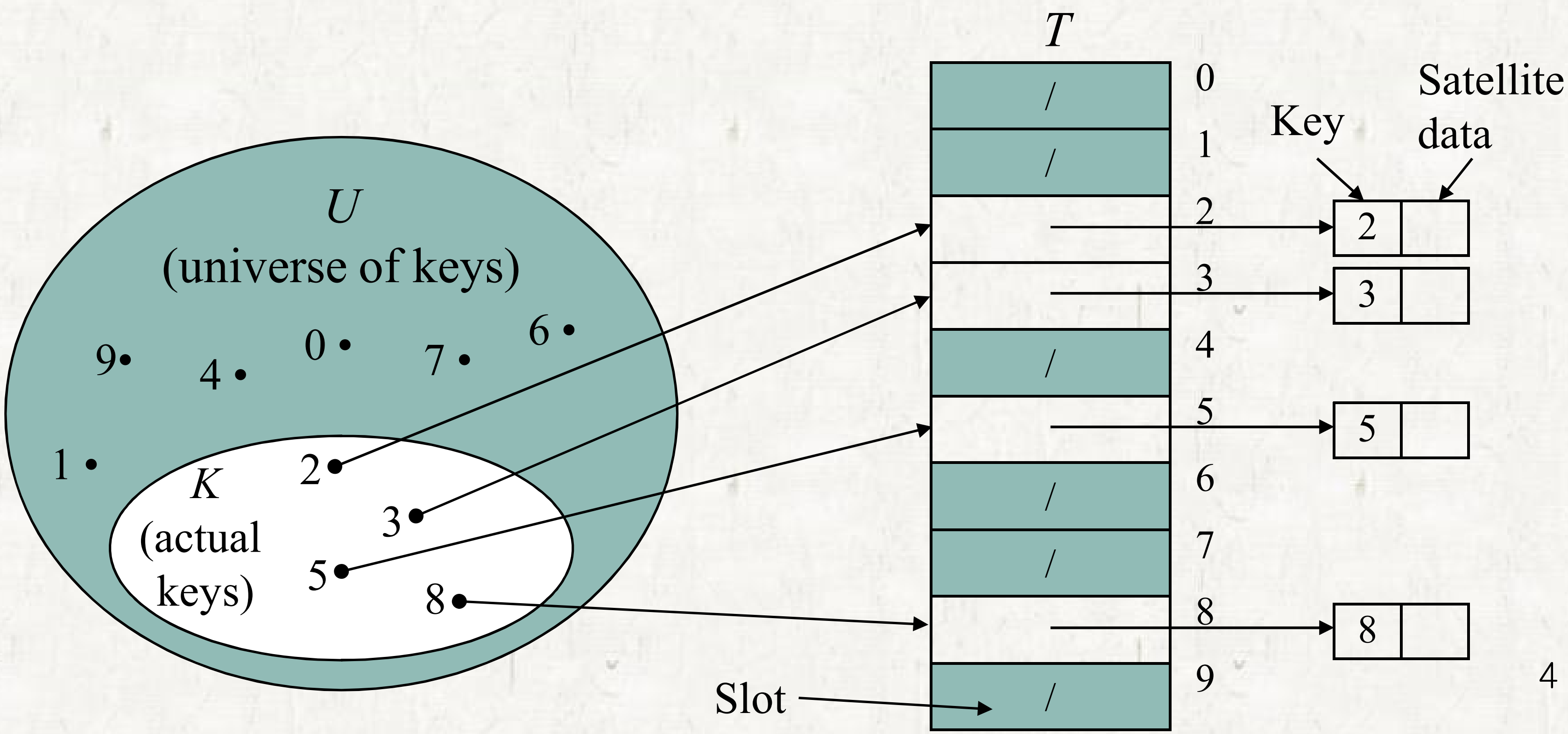
- **Data structure review**
- **Direct-address tables**
- **Hash tables**
- **Hash functions**
- **Open addressing**

Data structure review

	Arrays		**Linked-lists**		**Binary search trees (avg)**	**Balanced search trees**	**Hash tables (avg)**
	Not sorted	Sorted	Not sorted	Sorted			
Search(x)	$O(n)$	$O(\lg n)$	$O(n)$	$O(n)$	$O(\lg n)$	$O(\lg n)$	$\Theta(1)$
Insert(x)	$\Theta(1)$	$O(n)$	$\Theta(1)$	$O(n)$	$O(\lg n)$	$O(\lg n)$	$\Theta(1)$
Insert(x) (dup search)	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(\lg n)$	$O(\lg n)$	$\Theta(1)$
Delete($\boldsymbol{i}$)	$\Theta(1)$	$O(n)$	$\Theta(1)$ (DLL)	$\Theta(1)$ (DLL)	$O(\lg n)$	$O(\lg n)$	$\Theta(1)$
Delete(x)	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(\lg n)$	$O(\lg n)$	$\Theta(1)$

Direct-address tables

- Generate a table T with $|U|$ slots.
  - Store **key *k*** into **slot *k***.
  - Assumption: ***No two elements have the same key***.



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] = NIL
```

- Running time : $O(1)$

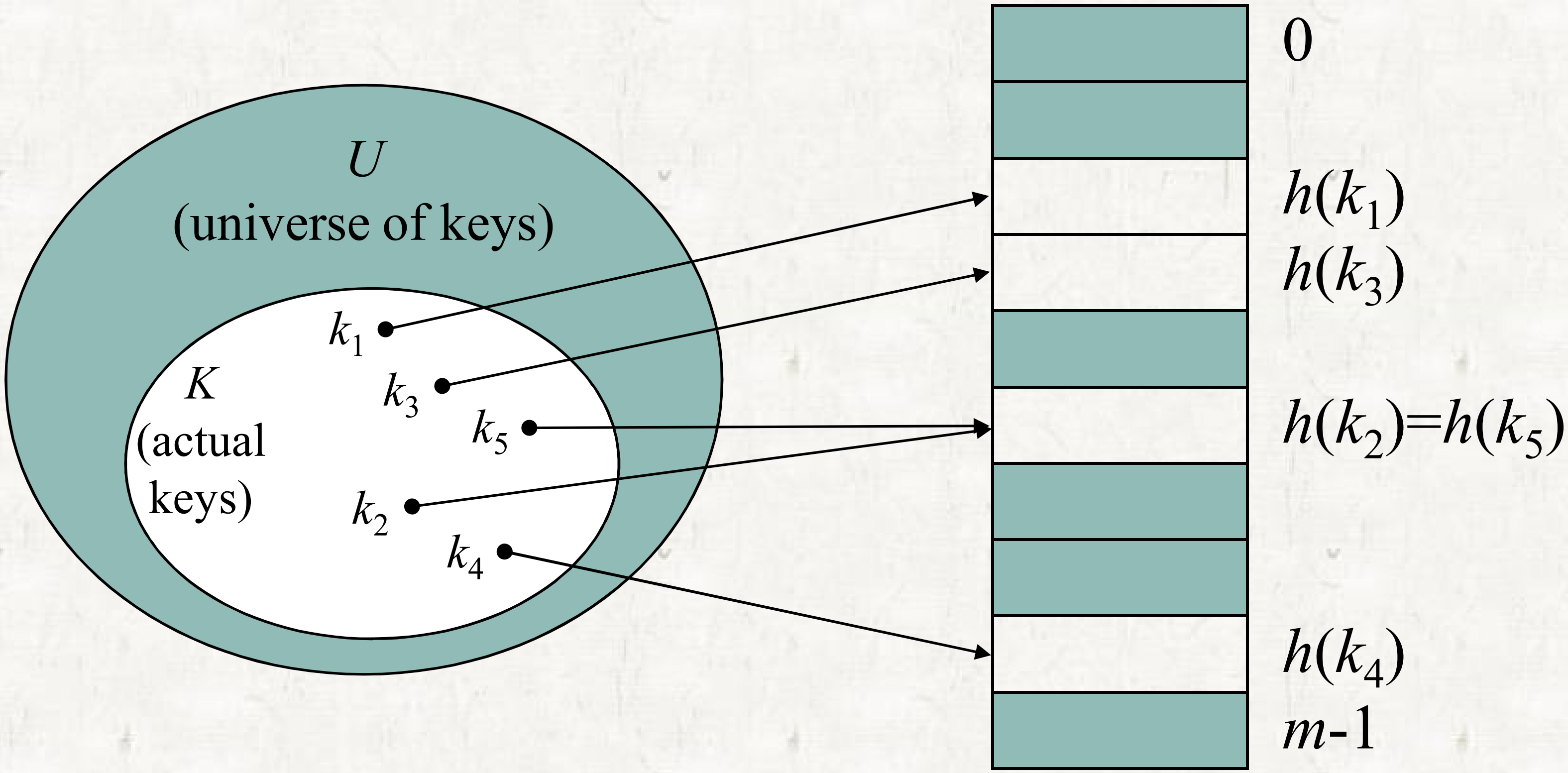
Direct-address tables

- Space consumption of direct addressing
  - $\Theta(|U|)$
  - If $|K|/|U|$ is small, most of the slots are wasted.
  - Is it possible to reduce the space requirement to $\Theta(|K|)$ while the running time is still $O(1)$?
    - No: worst case
    - Yes: Average case

Hash tables

- Hashing
  - **Key *k* is stored in slot *h*(*k*).**
  - *h* is called a ***hash function.***
    - A hash function computes the slot from the key *k*.
    - $h : U \rightarrow \{0, 1, ..., m - 1\}$.
  - We say that key *k* ***hashes*** to slot *h*(*k*).
  - We also say that *h*(*k*) is the ***hash value*** of *k*.

Hash tables



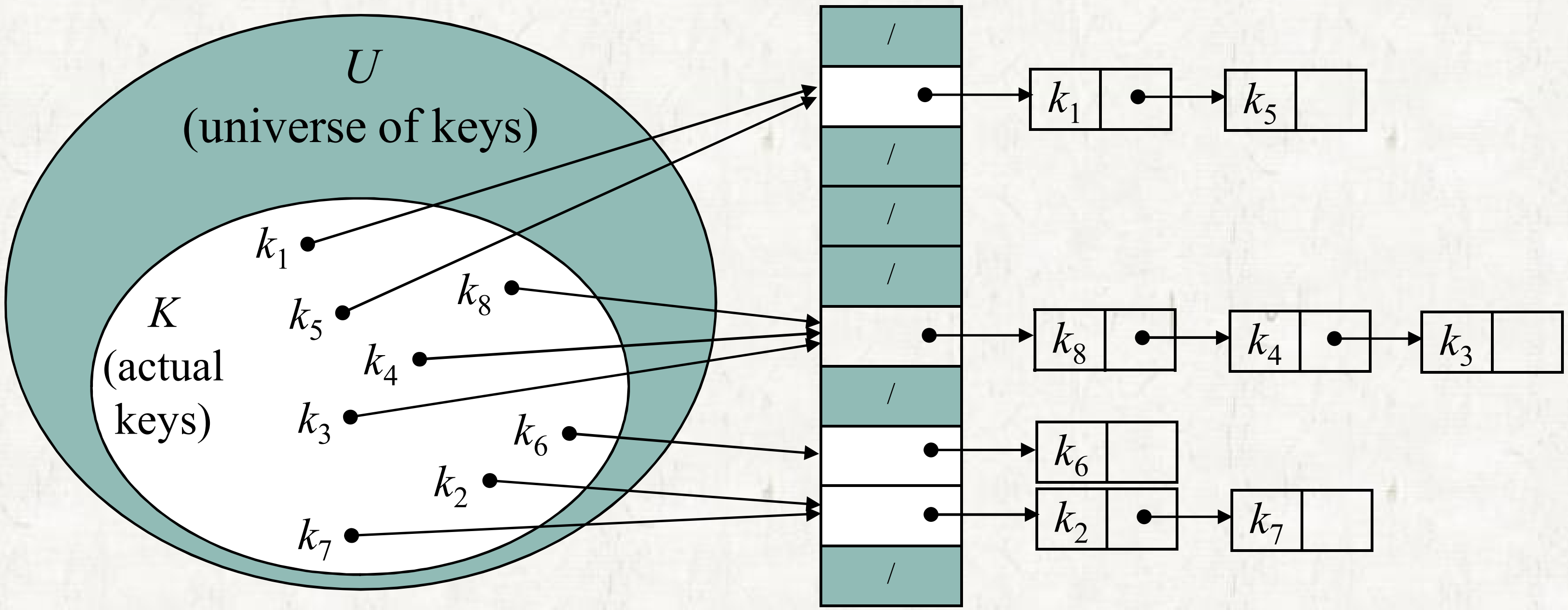
U
(universe of keys)
K
(actual
keys)
k_1
k_3
k_5
k_2
k_4
0
$h(k_1)$
$h(k_3)$
$h(k_2)=h(k_5)$
$h(k_4)$
$m-1$

Hash tables

- ***Collision*** : two keys may hash to the same slot.
- Avoiding collisions
  - A good hash function minimizes collisions.
  - But avoiding collisions is impossible if $|U| > m$.

Hash tables

- **Collision resolution by chaining**
  - Maintain a ***linked list*** for each slot to store keys.



Hash tables

CHAINED-HASH-INSERT(T, $\boldsymbol{x}$)
    insert x into list $T[h(x.key)]$

CHAINED-HASH-DELETE(T, $\boldsymbol{x}$)
    delete x from the list $T[h(x.key)]$

CHAINED-HASH-SEARCH(T, $\boldsymbol{k}$)
    search for an element with key k in list $T[h(k)]$

Hash tables

- ***Insertion*: $O(1)$**
  - Assumption: The element x being inserted is not already present in the table.
- ***Deletion*: $O(1)$**
  - If the lists are doubly linked.
  - Note that CHAIN-HASH-DELETE takes as input an element x so that we don't have to search for x.
- ***Search*: $O(l)$**
  - l is the length of the list.

Hash tables

- **Worst search time**
  - $\Theta(n)$
  - All n keys hash to the same slot, creating a list of length n.
  - No better than a linked list storing all the elements.

Hash tables

- **Average search time**
  - $\Theta(1+\alpha)$
  - α: the average length of a chain, called ***load factor***.
  - $\boldsymbol{\alpha = n/m}$ where n is the number of elements in the table and m is the number of slots.
  - unsuccessful search vs. successful search

Self-study

- **Exercise 11.2-1**
  - Counting the number of collisions in a hash table.
- **Exercise 11.2-3**
  - Are sorted lists useful for the hash table?
- **Exercise 11.2-5**
  - Existence of the worst-case

Contents

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

Hash functions

- **What makes a good hash function?**
  - A good hash function satisfies ***simple uniform hashing*** assumption.
    - Each key is equally likely to hash to any of the m slots.
    - Each key hashes independently of where any other key has hashed to.

Hash functions

- **Interpreting keys as natural numbers**
  - Hash functions assume that the universe of keys is the set **N** = {0, 1, 2, ...} of natural numbers.
  - If the keys are not natural numbers (such as character strings), it is necessary to interpret them as natural numbers.

Hash functions

- **Character strings → numbers**
  - For example: ***pt***
  - p = 112 and t = 116 in the ASCII character set
  - Expressed as a radix-128 integer
  - pt becomes $(112 \cdot 128)+116 = 14452$

The division method

- **The division method**
  - Divide k by m and take the remainder.
    - $h(k) = k \bmod m$.
  - Example
    - $m = 12$, $k = 100$
      - $h(k) = 100 \bmod 12 = 4$

The division method

- **Certain values of *m* are avoided.**
  - $m = 2^p$
    - $h(k)$ is just the p lowest-order bits of k.
    - $m = 2^4$, $h(k) = k \bmod 2^4$
    - k = 10110100, m = 00010000, h(k) = 0100
    - The hash function depends on only low-order p-bits instead of all the bits.
    - So, if all low-order p-bit patterns are equally likely, it is not a bad choice.
    - But, otherwise, m should not be a power of 2.

The division method

- **Certain values of *m* are avoided.**
  - $m = 2^p - 1$
    - When k is a character string interpreted in radix 2^p, permuting the characters of k does not change its hash value.
    - Example

The division method

- **A good choice**
  - **A prime not too close to an exact power of 2**.
    - If the number of keys are about 2000 and we don't mind examining an average of 3 elements in an unsuccessful search, we can allocate a hash table of size $m = 701$.
      - 701 is a prime near 2000/3 but not near any power of 2.
    - $h(k) = k \bmod 701$.

Contents

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

Open addressing

- **Open addressing**
  - Another method to resolve the collision.
  - All elements are stored in the hash table itself.

- The advantage of open addressing
  - It avoids pointers altogether.
  - The extra memory provides the hash table with a larger number of slots for the same amount of memory.
    - Yielding fewer collisions and faster retrieval.

Linear probing

- $m = 13$
- $k = \{5, 14, 29, 25, 17, 21, 18, 32, 20, 9, 15, 27\}$

$h(k) = k \mod 13$

	T	
0		
1	14	27
2	15	
3	29	
4	17	
5	5	18
6		32
7		20
8	21	
9		9
10		
11		
12	25	

Open addressing

- **Insertion**
  - Examine the hash table (***probe***) until it finds an empty slot.
  - The sequence of positions probed *depends upon the key being inserted*.
  - The ***probe sequence*** for every key k

$$< h(k, 0), h(k, 1), \ldots , h(k, m-1) >$$

  - be a permutation of $<0, 1, \ldots , m-1>$.

Open addressing

- **HASH-INSERT**
  - It takes as input a hash table *T* and a key *k*.

```
HASH-INSERT(T, k)
i = 0
repeat
    j = h (k, i)
    if T [ j] == NIL
        T [ j] = k
        return  j
    else  i = i +1
until  i == m
error  "hash table overflow"
```

Open addressing

- **HASH-SEARCH**
  - It takes as input a hash table *T* and a key *k*.

```
HASH-SEARCH(T, k)
i = 0
repeat
        j = h (k, i)
        if T [ j] == k
                return j
        i = i +1
until T [ j] == NIL or i == m
return NIL
```

Open addressing

- **Deletion**
  - Can you remove the key physically?
  - Mark the slot by the special value “DELETED”.
  - When we use the special value DELETED, however, search times are no longer dependent on the load factor α.
  - For this reason, chaining is more commonly selected when keys must be deleted.

Open addressing

- Three common techniques for open addressing.
  - **Linear probing**
  - **Quadratic probing**
  - **Double hashing**

Linear probing

- Given an ordinary hash function $h' : U \rightarrow \{0, 1, \ldots , m\text{-}1\}$, which we refer to as an ***auxiliary hash function***, the method of ***linear probing*** uses the hash function

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

  - for $i = 0, 1, \ldots, m\text{-}1$

Linear probing

- $m = 13$
- $k = \{5, 14, 29, 25, 17, 21, 18, 32, 20, 9, 15, 27\}$

$h(k, i) = (k + i) \bmod 13$

	T	
0		
1	14	27
2	15	
3	29	
4	17	
5	5	18
6		32
7		20
8	21	
9		9
10		
11		
12	25	

Linear probing

- Linear probing is easy to implement, but it suffers from a problem known as ***primary clustering***.
  - Long runs of occupied slots build up, increasing the average search time.
  - Clusters arise since an empty slot preceded by i full slots gets filled next with probability $(i + 1) / m$.
  - Long runs of occupied slots tend to get longer, and the average search time increases.

Quadratic probing

- ***Quadratic probing*** uses a hash function of the form

$$h(k, i) = (h'(k) + c_1\, i + c_2\, i^2) \bmod m$$

  - where h' is an auxiliary hash function, c_1 and $c_2 \neq 0$ are auxiliary constants, and $i = 0, 1, \ldots, m$-1.

Quadratic probing

- $m = 13$
- $k = \{5, 14, 29, 25, 17, 21, 18, 32, 20, 9, 15, 27\}$

$h(k, i) = (k + i + 3i^2) \bmod 13$

	T	
0		
1	14	27
2	15	
3	29	
4	17	
5	5	18
6	32	
7	20	
8	21	
9		9
10		
11		
12	25	

Quadratic probing

- If two keys have the same initial probe position, then their probe sequences are the same, since $h(k_1, 0) = h(k_2, 0)$ implies $h(k_1, i) = h(k_2, i)$.
- This property leads to a milder form of clustering, called ***secondary clustering***.

Double hashing

- ***Double hashing*** uses a hash function of the form

$$h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$$

- The initial probe is to position $T[h_1(k)]$.
  - Successive probe positions are offset from previous positions by the amount $h_2(k)$, modulo m.

Double hashing

- $m = 13$
- $k = \{5, 14, 29, 25, 17, 21, 18, 32, 20, 9, 15, 27\}$

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

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

$$h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod 13$$

	T	
0		
1	14	27
2	15	
3	29	
4	17	
5	5	18
6	32	
7	20	
8	21	
9	9	
10		
11		
12	25	

Double hashing

- The value $h_2(k)$ must be relatively prime to the hash-table size m for the entire hash table to be searched.
  - A way to ensure this condition is to let m be a power of 2 and to design h_2 so that it always produces an odd number.
  - Another way is to let m be prime and to design h_2 so that it always returns a positive integer less than m.

Self-study

- **Exercise 11.4-1**
  - Hashing example
- **Exercise 11.4-2**
  - HASH-DELETE & HASH-INSERT