Ch16: Hash Tables

305233, 305234

Algorithm Analysis and Design

Jiraporn Pooksook
Naresuan University

Dictionaries: Abstract Data Type

- Dictionaries (Abstract Data Type) is to maintain set of items, each with a key
  - INSERT(item)
  - DELETE(item)
  - SEARCH(key) -> return item with given key or report does not exist
- A has table is an effective data structure for implementing dictionaries.
- Worst case time for searching is O(n) but its expected time is O(1).

Dictionaries in Python

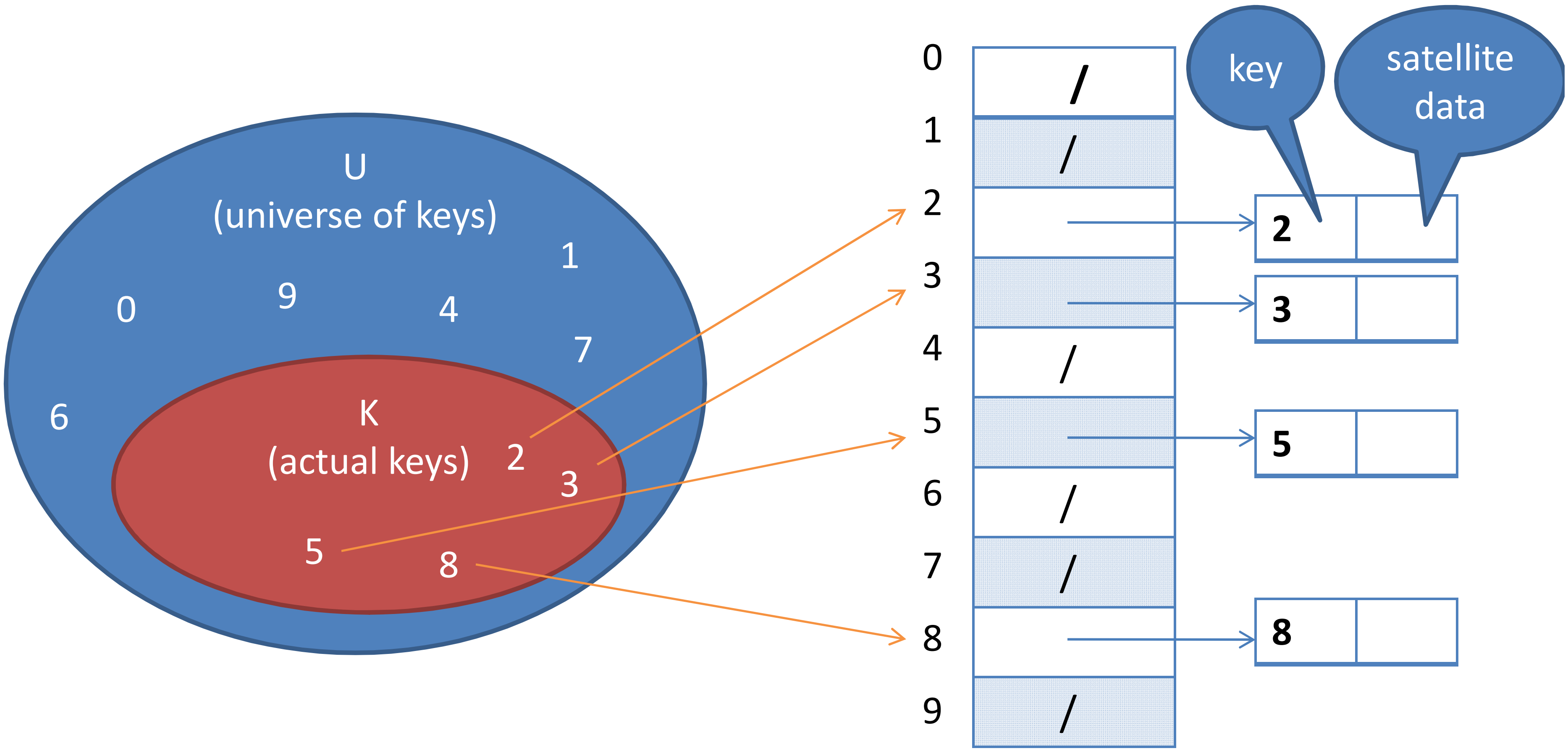
- D={1234: ‘Bob’,

  5678, ‘Alice’}
- Search by D[key]
- Insert by D[key] = value
  - D[5678]=‘Robert’
- Delete by D[key]
  - del D[1234]

Direct-Address Tables

- A set of keys is in a set of **universe** U = {0,1,…,m-1} where m is not too large.
- A direct-address table is an array denoted by T[0..m-1] in which each position, or **slot**, corresponds to a key in the universe U.

Direct-Address Table



Direct-Address Tables

```
DIRECT-ADDRESS-SEARCH(T, k)
        return T[k]
```

```
DIRECT-ADDRESS-INSERT(T, x)
        T[key[x]] = x
```

```
DIRECT-ADDRESS-DELETE (T, k)
        T[key[x]] = NIL
```

Each operation takes only O(1) time.

Disadvantages of Direct-addressing

- Keys may not be non-negative integers.
- Direct-address tables require a large size of memory.
  - If the universe U is large, we have to store a table T of size U.

Disadvantages of Direct-addressing

- Keys may not be non-negative integers.
- Solution: using **prehash** to map key to non-negative integers.
  - A string of bits represents an integer.
  - In python using function hash(x) means prehash.
- Direct-address tables require a large size of memory.
- Solution: using hashing

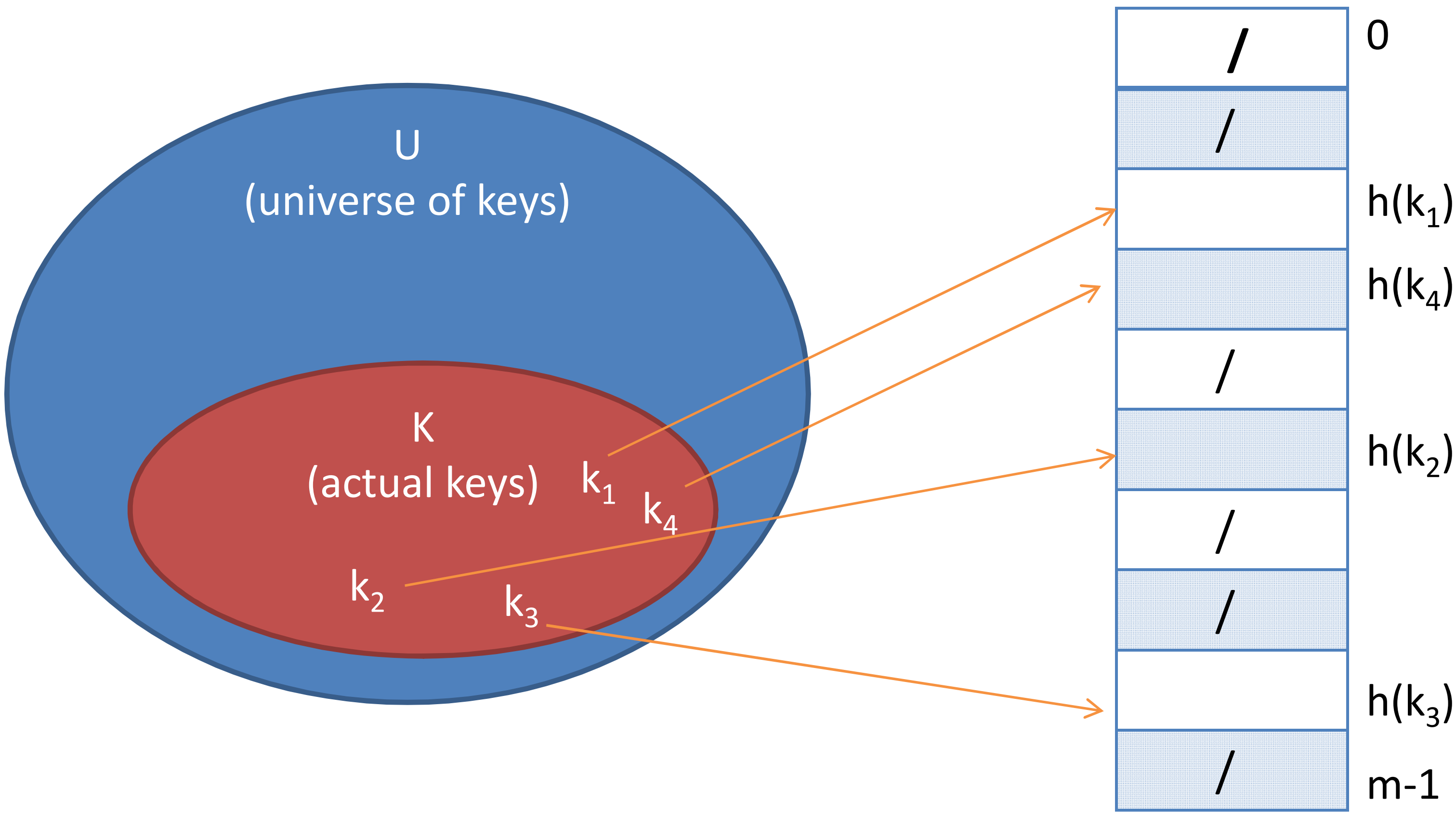
Hash Tables

- We use a hash function h to compute the slot from the key k.
- Hence h maps the universe U of keys into the slots of a hash table T[0..m-1]:

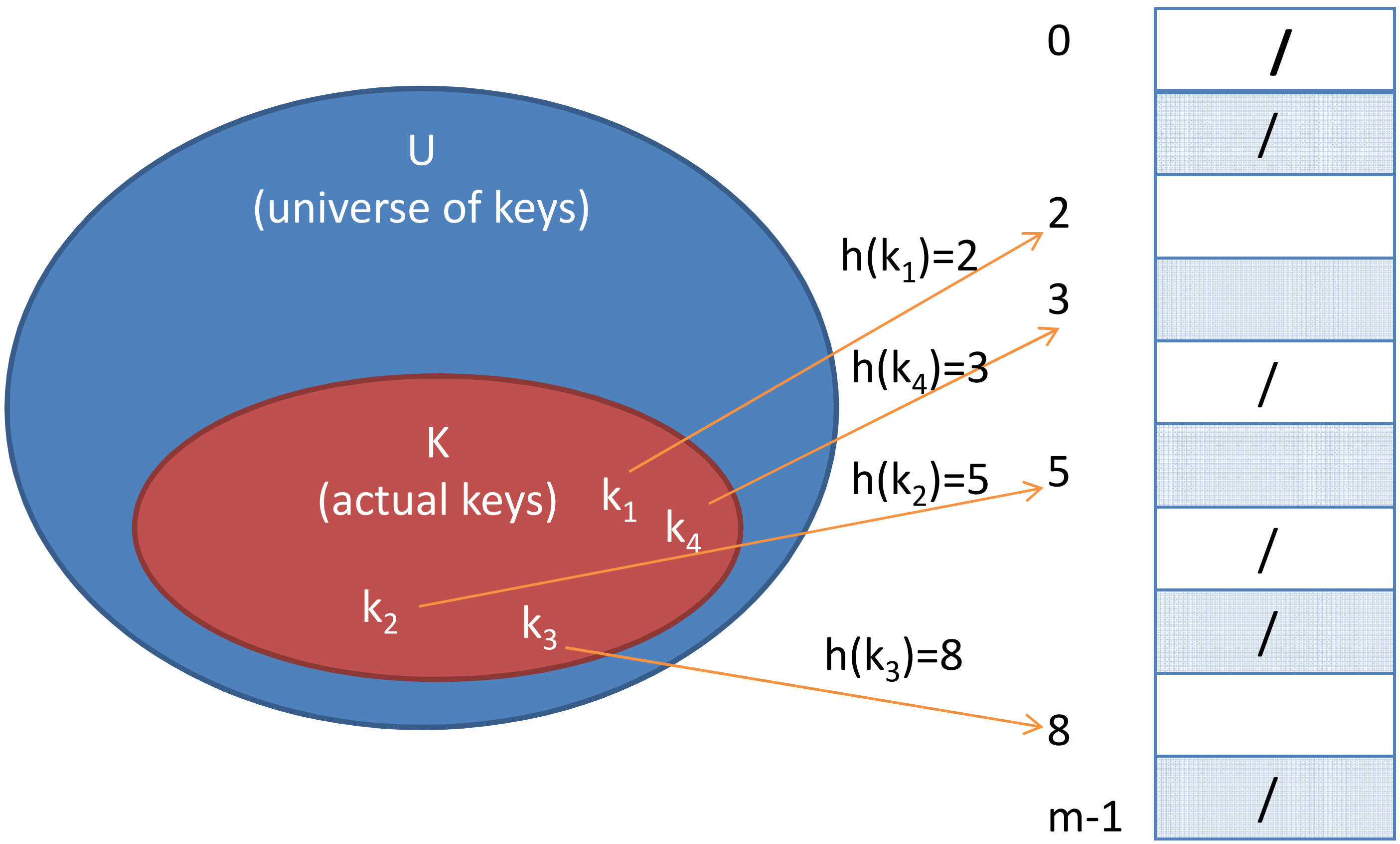
  $$h: U \rightarrow \{0,1,\ldots,m-1\}$$

- We say that an element with key k hashes to slot h(k); we also say that h(k) is the hash value of key k.

Hash Table

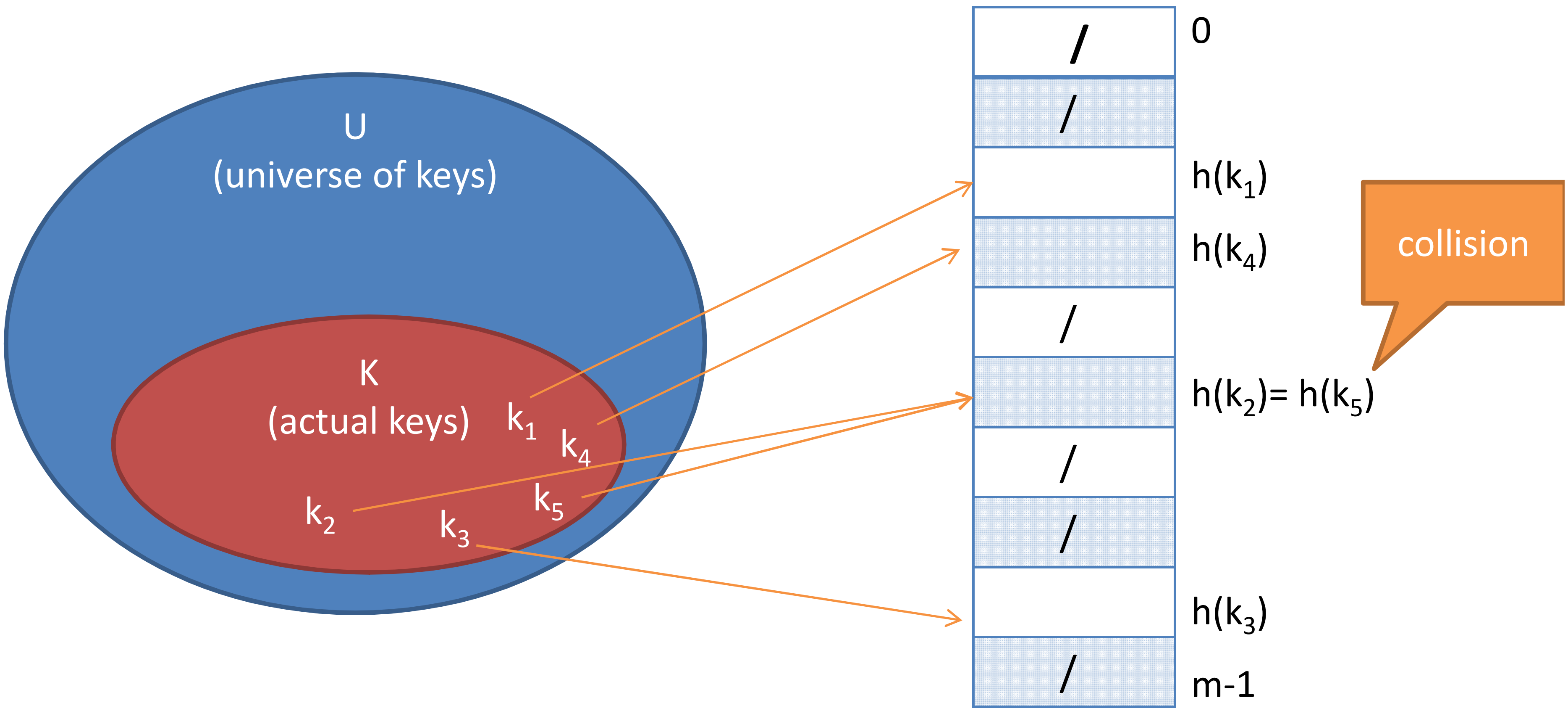


Hash Table



U
(universe of keys)
K
(actual keys)
k1
k4
k2
k3
h(k1)=2
h(k4)=3
h(k2)=5
h(k3)=8
0
2
3
5
8
m-1
/
/
/
/
/
/

Hash Table

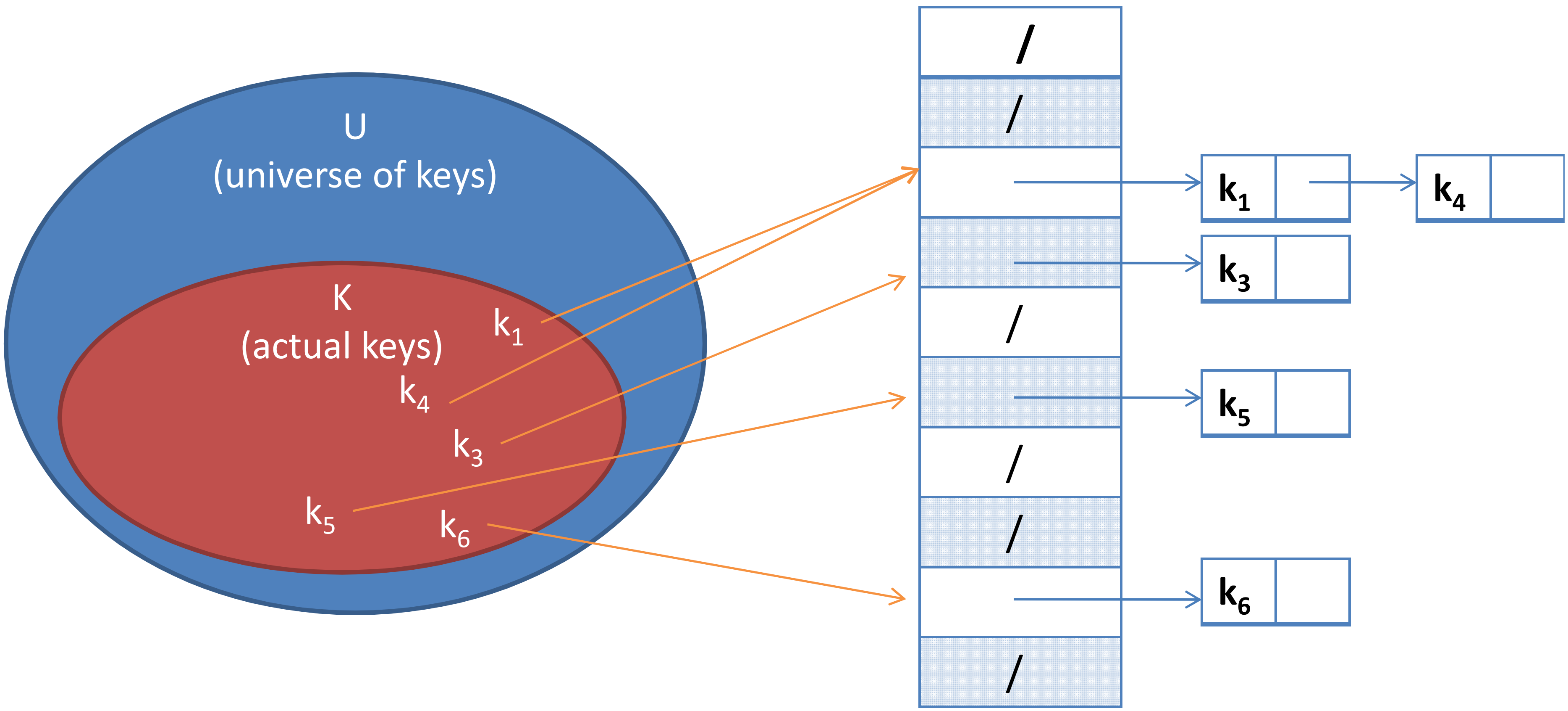


U
(universe of keys)
K
(actual keys)
k_1
k_4
k_2
k_5
k_3
/
/
/
/
/
/
0
$h(k_1)$
$h(k_4)$
$h(k_2) = h(k_5)$
$h(k_3)$
m-1
collision

Collision resolution by Chaining

- We put all the elements that hash to the same slot in a linked list.

Hash with Chaining



Hash with Chaining

Worst-case = length of the list

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

Worst-case = O(1)

CHAINED-HASH-INSERT(T, x)
insert x at the head of list T[h(key[x])]

Worst-case = O(1) if lists are doubly linked.

CHAINED-HASH-DELETE (T, k)
delete x from the list T[h(key[x])]

Analyze Hash with Chaining

- **Simple uniform hashing** is an assumption that any given element is equally likely to hash into any of the m slots independently of where any other element has hashed to.
- For $j = 0, 1, \ldots, m-1$. Let us denote the length of the list $T[\,j\,]$ by n_j, so that

$$n = n_0 + n_1 + \ldots + n_{m-1}$$

- The average value of n_j is $E[n_j] = \alpha = n/m$

Analyze Hash with Chaining

- **We assume** that the hash value h(k) can be computed in O(1) time, so that the time required to search for an element with key k depends linearly on the length $n_{h(k)}$ of the list T[h(k)].
- We consider two cases:
  - The search is unsuccessful.
  - The search successfully finds an element with key k.

Analyze Hash with Chaining

- The expected time to search **unsuccessfully** for a key k is the expected time to search to the end of the list T[h(k)].
- The list T[h(k)] has expected length=$E[n_{h(k)}] = \alpha$
- Hence the expected number of elements examined in unsuccessful search is α, and the total time required (including the time for computing h(k)) = $O(1+ \alpha)$

Analyze Hash with Chaining

- The expected time to search **successfully** for an element x is 1 more than the number of elements that appear before x in x's list.
- Let x_i denote the ith element inserted into the table for i= 1,2,…,n
- Let $k_i = key[x_i]$
- For keys k_i and k_j we define the random variable $X_{ij} = I\{h(k_i)=h(k_j)\}$
- Under the simple uniform hashing assumption, we have $Pr\{h(k_i)=h(k_j)\} = 1/m$, and so $E[X_{ij}]= 1/m$

Analyze Hash with Chaining

- Hence the expected number of elements examined in a successful search is:

$$E\left[\frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}X_{ij})\right]$$

$$=\frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}E\left[X_{ij}\right])$$

$$=\frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}\frac{1}{m})$$

$$=1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)$$

$$=1+\frac{1}{nm}(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i)$$

$$=1+\frac{1}{nm}(n^2-\frac{n(n+1)}{2})$$

$$=1+\frac{n-1}{2m}$$

$$=1+\frac{\alpha}{2}-\frac{\alpha}{2n}$$

Total time required for a successful search is O(1+α)

Analyze Hash with Chaining

- If the number of hash-table slots is at least proportional to the number of elements in the table, we have n = O(m) and, consequently $\alpha = n/m = O(m)/m = O(1)$.
- Searching takes constant time on average.
- All dictionary operations can be supported in O(1) time on average.

Hash Functions

- A good hash function 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.

- It is typically not possible to check this condition.

The Division Method

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

- For example, if hash table has size $m = 12$ and key $k = 100$ then $h(k) = 4$
- We usually avoid certain values of m. For example m should not be a power of 2.
- A prime is often a good choice for m.

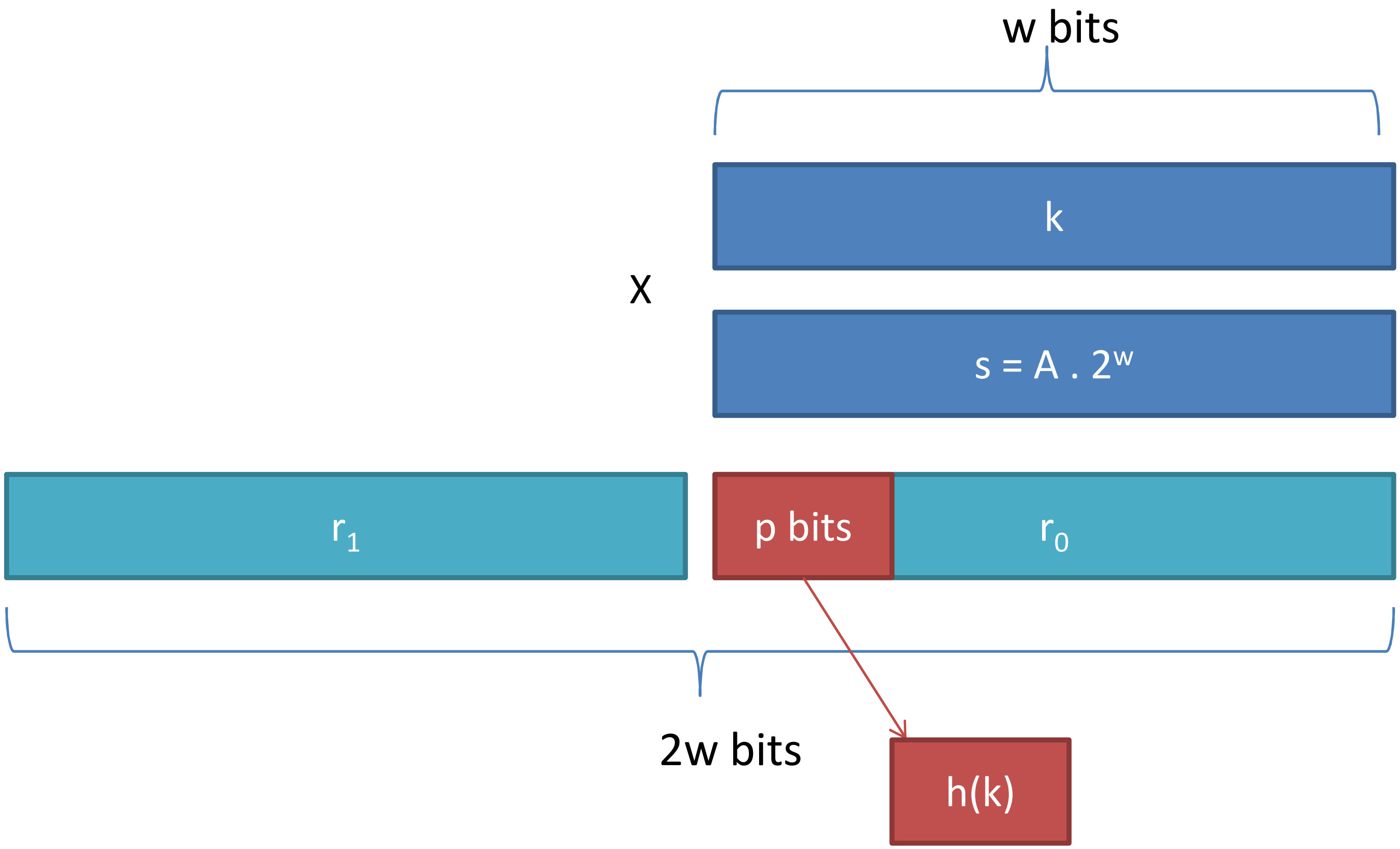
The Multiplication Method

$$h(k) = [m(k\ A \bmod 1)]$$

$$h(k) = [(k.A) \bmod 2^{w}\ ] >> (w-p)$$

- A is in the range $0 < A < 1$,
  suggest that $A = (5^{1/2} - 1)/2 = 0.6180339887...$
- $m = 2^{p}$
- k has w bits.

The Multiplication Method



Example: The Multiplication Method

- $k = 123456$, $p = 14$, $m = 2^{14} = 16384$, $w = 32$
- Hence choose A to be the fraction of the form $s/ 2^{32}$ that is closest to $(5^{1/2} - 1)/2$.
- $A = 2654435769$
- $k.s = 327706022297664$

$$= (\underbrace{76300}_{r_1}.\ 2^{32}) + \underbrace{17612864}_{r_2}$$

14 most significant bits of r0 yield the value h(k) = 67

Universal Hashing

$$h(k) = [(ak+b) \bmod p] \bmod m$$

- a , b are randomed and be in {0,1,..,p-1}
- p is a prime which is greater than the size of universe.
- The worst case key k_i != k_j :

$$\Pr\{h(k_i)= h(k_j)\} = 1/m$$

Ideal situation of collision

Collision resolution by Open Addressing

- Each table entry contains either an element of the dynamic set or NIL.
  - No chaining and only 1 item per slot
- When searching for an element, we examine table slots until the desired element is found or it is clear that the element is not in the table.
- In open addressing the hash table can fill up so that no further insertions can be made.
- The load factor α can never exceed 1.

Open Addressing

- To perform insertion using open addressing, we successively examine, or **probe**, the hash table until we find an empty slot in which to put the key.
- Instead of being fixed in the order 0,1,…,m-1 the sequence of positions probed depends upon the key being inserted.

Open Addressing

- The hash function becomes:

$$h: U \times \{0,1,\dots,m-1\} \to \{0,1,\dots,m-1\}$$

- For every key k, the probe sequence $\langle h(k,0), h(k,1), \dots, h(k, m-1)\rangle$

be a permutation of $\langle 0,1,\dots,m-1\rangle$

Open Addressing

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

Open Addressing

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

Example: Open Addressing

- insert(586) , h(586,1) = 1
- …..
- insert(481) , h(481,1) = 6
- insert(496) , h(496,1) =4
- insert(496) , h(496,2) =1
- insert(496) , h(496,3) =3

Fail probe

0	
1	586
2	133
3	496
4	204
5	
6	481
7	

Linear Probing

- Given an ordinary hash function

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

- the method of linear probing use the hash function :

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

- For $i = 0,1,\ldots,m-1$
- Long runs of occupied slots build up, increasing the average search time!!

Quadratic Probing

- Given an ordinary hash function

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

- the method of quadratic probing use the hash function :

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

- For i = 0,1,…,m-1 and c_1 and c_2 are not equal to 0.

Double Probing

- Given an ordinary hash function

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

- the method of double probing use the hash function :

  $$\boxed{h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m}$$

- For $i = 0,1,\ldots,m-1$ and $h_1(k)$ and $h_2(k)$ are auxiliary hash functions.
- The value $h_2(k)$ must be relatively prime to the hash-table size m.

Analyze Open Addressing

- We have at most one element per slot, thus $n \leq m$, which implies $\alpha \leq 1$.
- We assume the uniform hashing is used.
- The probe sequence $\langle h(k,0)$, $h(k,1)$, … ,$h(k, m-1)\rangle$ used to insert or search for each key k is equally likely to be any permutation of $(0,1,\ldots,m-1)$.

Analyze Open Addressing

- The expected number of probes in an unsuccessful search is at most 1/(1- α)
- Thus inserting an element into an opening address hash table with load factor α requires at most 1/(1- α) probes on average, assuming uniform hashing.
- The expected number of probes in a sucessful search is at most $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$