



Hash Tables



SDP-4

Dictionary

□ **Dictionary:**

- Dynamic-set data structure for storing items indexed using *keys*.
- Supports operations **Insert, Search, and Delete**.
- Applications:
 - Symbol table of a compiler.
 - Memory-management tables in operating systems.
 - Large-scale distributed systems.

□ **Hash Tables:**

- Effective way of implementing dictionaries.
- Generalization of ordinary arrays.



Direct-address Tables

- Direct-address Tables are **ordinary arrays**.
- **Facilitate direct addressing**.
 - Element whose key is k is obtained by indexing into the k^{th} position of the array.
- **Applicable** when we can afford to allocate an array with one position for every possible key.
 - i.e. when the universe of keys U is small.
- **Dictionary operations** can be implemented to take **$O(1)$ time**.
 - Details in Sec. 11.1.



Hash Tables

□ **Notation:**

□ U – Universe of all possible keys.

□ K – Set of keys actually stored in the dictionary.

□ $|K| = n$.

□ **When U is very large,**

□ Arrays are not practical.

□ $|K| \ll |U|$.

□ **Use a table of size proportional to $|K|$ – The hash tables.**

□ However, we lose the direct-addressing ability.

□ Define functions that map keys to slots of the hash table.



Hashing

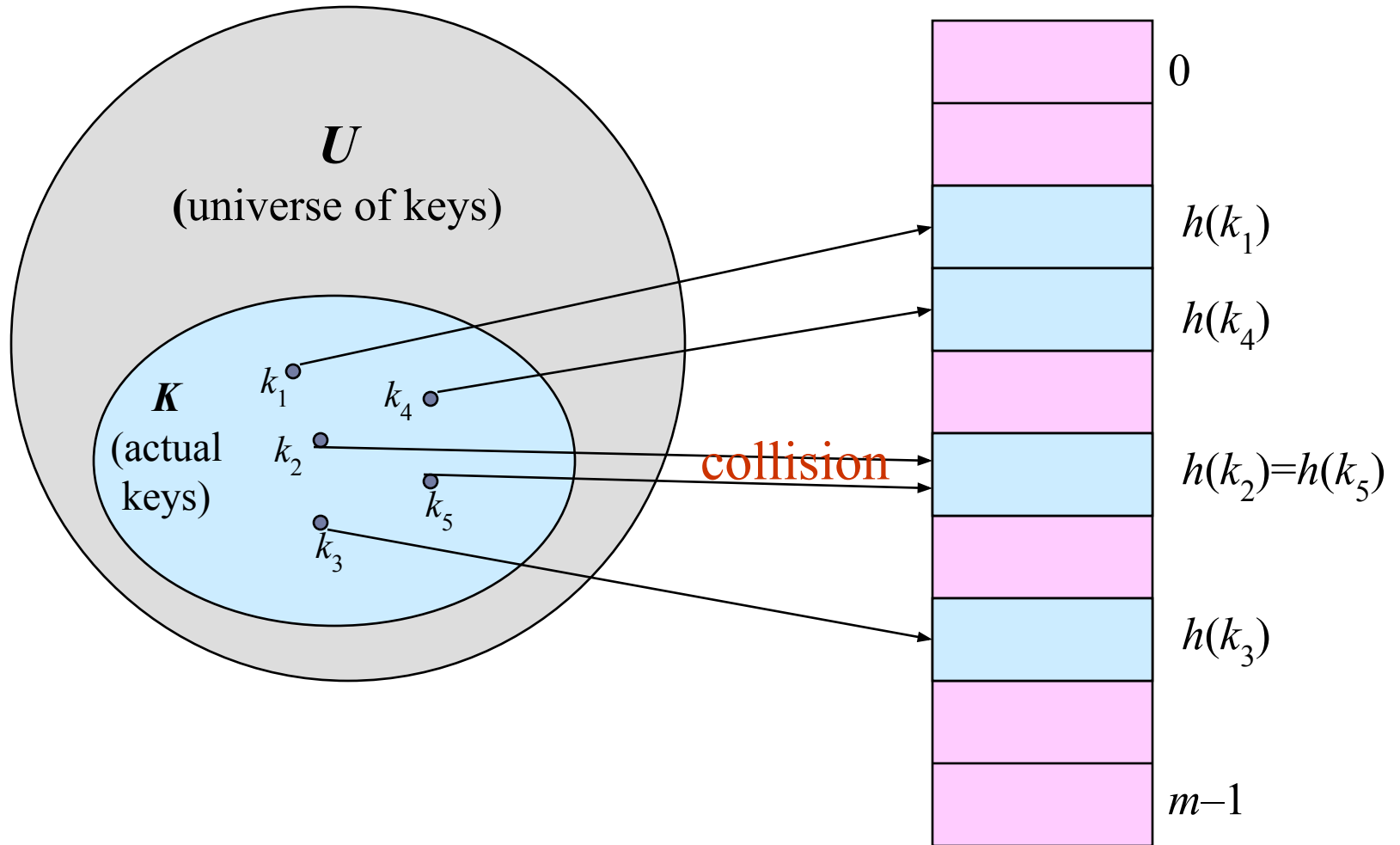
- **Hash function h** : Mapping from U to the slots of a hash table $T[0..m-1]$.

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

- With arrays, key k maps to slot $A[k]$.
- With hash tables, key k maps or “**hashes**” to slot $T[h[k]]$.
- $h[k]$ is the **hash value** of key k .



Hashing



Issues with Hashing

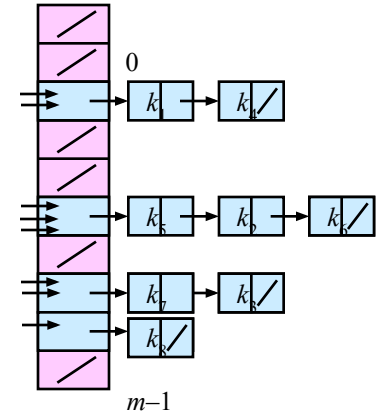
- Multiple keys can hash to the same slot – collisions are possible.
 - Design hash functions such that collisions are minimized.
 - But avoiding collisions is impossible.
 - Design collision-resolution techniques.
- Search will cost $\Theta(n)$ time in the worst case.
 - However, all operations can be made to have an expected complexity of $\Theta(1)$.



Methods of Resolution

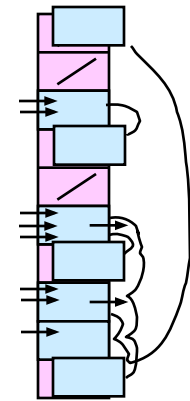
□ Chaining:

- Store all elements that hash to the same slot in a linked list.
- Store a pointer to the head of the linked list in the hash table slot.

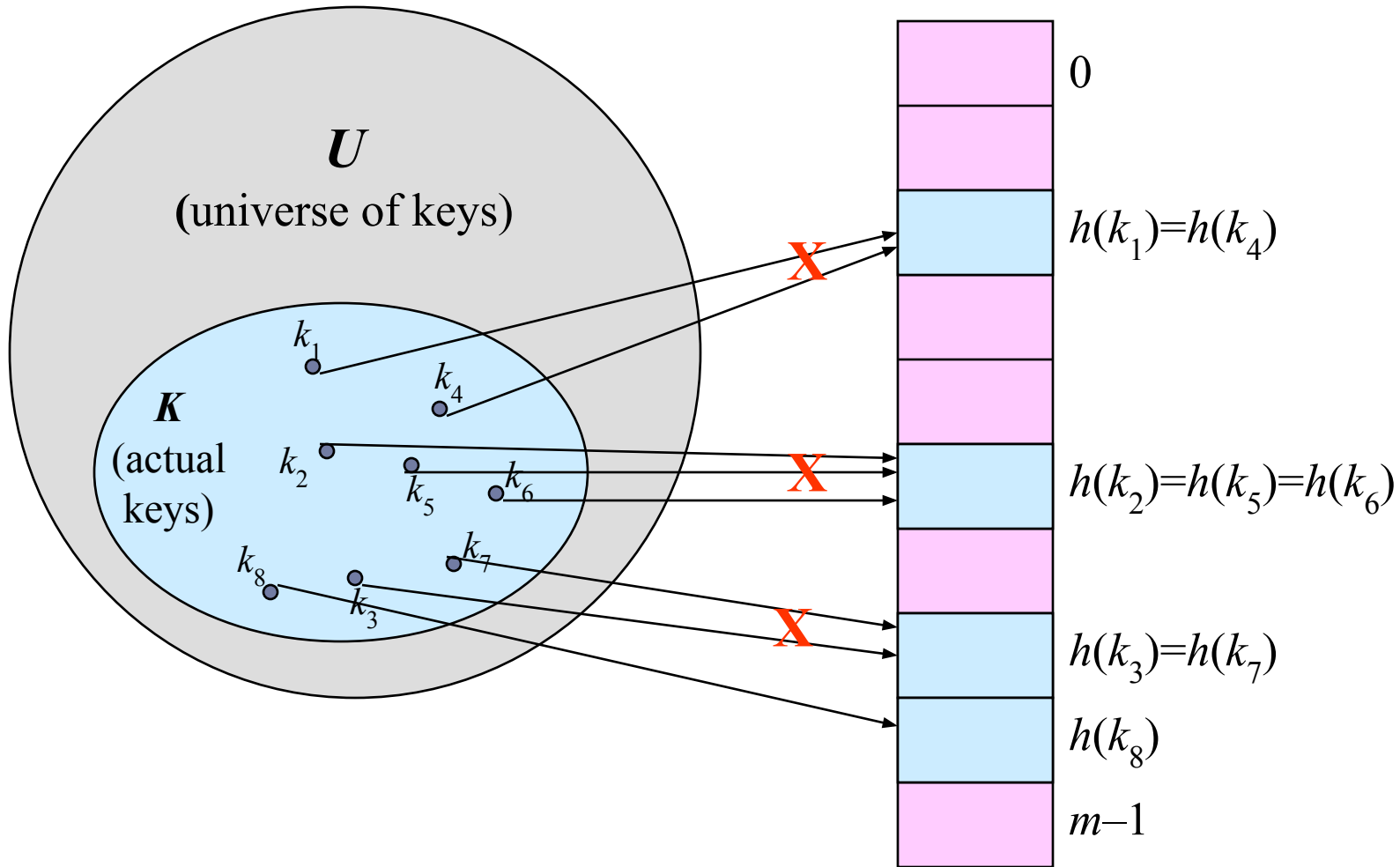


□ Open Addressing:

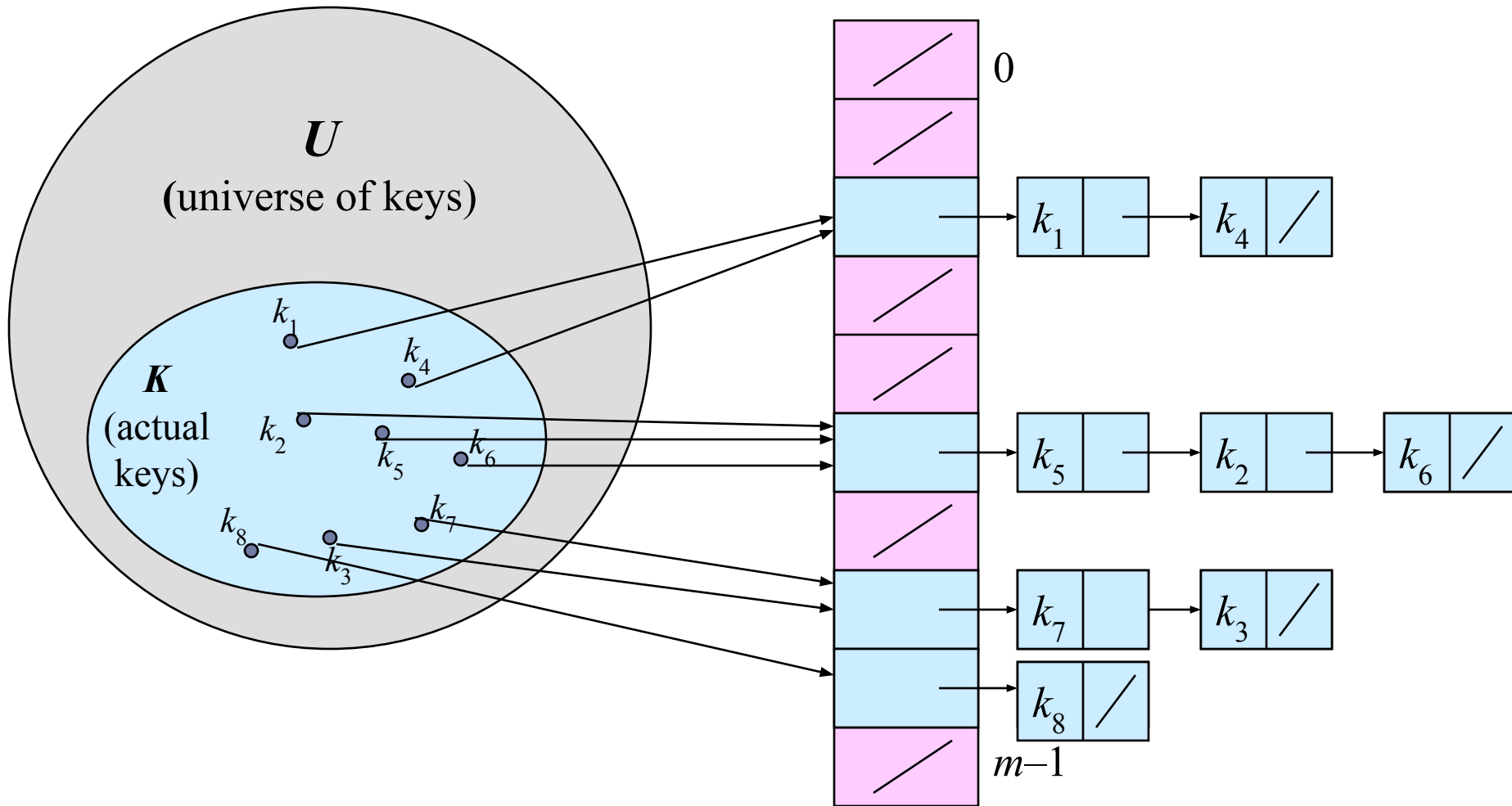
- All elements stored in hash table itself.
- When collisions occur, use a systematic (consistent) procedure to store elements in free slots of the table.



Collision Resolution by Chaining



Collision Resolution by Chaining



Hashing with Chaining

Dictionary Operations:

- **Chained-Hash-Insert** (T, x)
 - Insert x at the head of list $T[h(\text{key}[x])]$.
 - Worst-case complexity – $O(l)$.
- **Chained-Hash-Delete** (T, x)
 - Delete x from the list $T[h(\text{key}[x])]$.
 - Worst-case complexity – proportional to length of list with singly-linked lists. $O(l)$ with doubly-linked lists.
- **Chained-Hash-Search** (T, k)
 - Search an element with key k in list $T[h(k)]$.
 - Worst-case complexity – proportional to length of list.



Analysis on Chained-Hash-Search

- **Load factor** $\alpha = n/m$ = average keys per slot.
 - m – number of slots.
 - n – number of elements stored in the hash table.
- **Worst-case complexity:** $\Theta(n)$ + time to compute $h(k)$.
- Average depends on how h distributes keys among m slots.
- **Assume**
 - *Simple uniform hashing.*
 - Any key is equally likely to hash into any of the m slots, independent of where any other key hashes to.
 - $O(1)$ time to compute $h(k)$.
- Time to search for an element with key k is $\Theta(|T[h(k)]|)$.
- Expected length of a linked list = load factor = $\alpha = n/m$.



Expected Cost of an Unsuccessful Search

Theorem:

An unsuccessful search takes expected time $\Theta(1+\alpha)$.

Proof:

- Any key not already in the table is equally likely to hash to any of the m slots.
- To search unsuccessfully for any key k , need to search to the end of the list $T[h(k)]$, whose expected length is α .
- Adding the time to compute the hash function, the total time required is $\Theta(1+\alpha)$.



Expected Cost of a Successful Search

Theorem:

A successful search takes expected time $\Theta(1+\alpha)$.

Proof:

- The probability that a list is searched is proportional to the number of elements it contains.
- Assume that the element being searched for is equally likely to be any of the n elements in the table.
- The number of elements examined during a successful search for an element x is **1 more than the number of elements that appear before x in x 's list.**
 - These are the elements inserted **after** x was inserted.
- **Goal:**
 - Find the average, over the n elements x in the table, of how many elements were inserted into x 's list after x was inserted.



Expected Cost of a Successful Search

Theorem:

A successful search takes expected time $\Theta(1+\alpha)$.

Proof (contd):

- Let x_i be the i^{th} element inserted into the table, and let $k_i = \text{key}[x_i]$.
- Define indicator random variables $X_{ij} = I\{h(k_i) = h(k_j)\}$, for all i, j .
- Simple uniform hashing $\Rightarrow \Pr\{h(k_i) = h(k_j)\} = 1/m$
 $\Rightarrow E[X_{ij}] = 1/m$.
- Expected number of elements examined in a successful search is:

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

No. of elements inserted after x_i into the same slot as x_i .

Proof – Contd.

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

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

(linearity of expectation)

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

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

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

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

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

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

Expected total time for a successful search

= Time to compute hash function + Time to search

= $O(2 + \alpha/2 - \alpha/2n) = O(1 + \alpha)$.



Expected Cost – Interpretation

- If $n = O(m)$, then $\alpha = n/m = O(m)/m = O(1)$.
⇒ Searching takes constant time on average.
- Insertion is $O(1)$ in the worst case.
- Deletion takes $O(1)$ worst-case time when lists are doubly linked.
- Hence, all dictionary operations take $O(1)$ time on average with hash tables with chaining.



Good Hash Functions

- **Satisfy the assumption of *simple uniform hashing*.**
 - Not possible to satisfy the assumption in practice.
- Often **use heuristics**, based on the domain of the keys, to create a hash function that performs well.
- Regularity in key distribution should not affect uniformity.
Hash value should be independent of any patterns that might exist in the data.
 - E.g. Each key is drawn independently from U according to a probability distribution P :
$$\sum_{k:h(k)=j} P(k) = 1/m \quad \text{for } j = 0, 1, \dots, m-1.$$
 - An example is the division method.



Keys as Natural Numbers

- Hash functions assume that the keys are natural numbers.
- When they are not, have to interpret them as natural numbers.
- **Example:** Interpret a character string as an integer expressed in some radix notation. Suppose the string is CLRS:
 - ASCII values: C=67, L=76, R=82, S=83.
 - There are 128 basic ASCII values.
 - So, $CLRS = 67 \cdot 128^3 + 76 \cdot 128^2 + 82 \cdot 128^1 + 83 \cdot 128^0$
 $= 141,764,947.$



Division Method

- Map a key k into one of the m slots by taking the remainder of k divided by m . That is,

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

- Example: $m = 31$ and $k = 78 \Rightarrow h(k) = 16$.
- **Advantage**: Fast, since requires just one division operation.
- **Disadvantage**: Have to avoid certain values of m .
 - Don't pick certain values, such as $m=2^p$
 - Or hash won't depend on all bits of k .
- **Good choice for m** :
 - Primes, not too close to power of 2 (or 10) are good.



Multiplication Method

- If $0 < A < 1$, $h(k) = \lfloor m (kA \bmod 1) \rfloor = \lfloor m (kA - \lfloor kA \rfloor) \rfloor$
where $kA \bmod 1$ means the fractional part of kA , i.e., $kA - \lfloor kA \rfloor$.
- **Disadvantage:** Slower than the division method.
- **Advantage:** Value of m is not critical.
 - Typically chosen as a power of 2, i.e., $m = 2^p$, which makes implementation easy.
- **Example:** $m = 1000$, $k = 123$, $A \approx 0.6180339887\dots$
$$h(k) = \lfloor 1000(123 \cdot 0.6180339887 \bmod 1) \rfloor$$
$$= \lfloor 1000 \cdot 0.018169\dots \rfloor = 18.$$

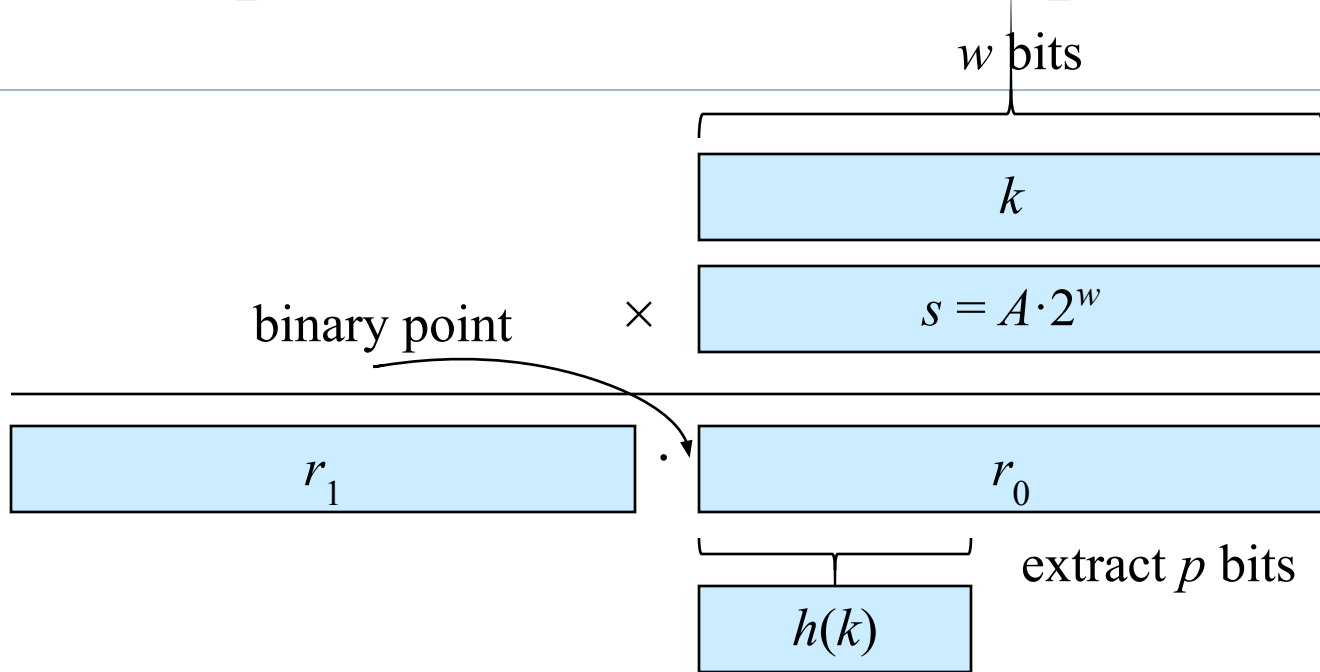


Multiplication Mthd. – Implementation

- Choose $m = 2^p$, for some integer p .
- Let the word size of the machine be w bits.
- Assume that k fits into a single word. (k takes w bits.)
- Let $0 < s < 2^w$. (s takes w bits.)
- Restrict A to be of the form $s/2^w$.
- Let $k \times s = r_1 \cdot 2^w + r_0$.
- r_1 holds the integer part of kA ($\lfloor kA \rfloor$) and r_0 holds the fractional part of kA ($kA \bmod 1 = kA - \lfloor kA \rfloor$).
- We don't care about the integer part of kA .
 - So, just use r_0 , and forget about r_1 .



Multiplication Mthd – Implementation



- We want $\lfloor m (kA \bmod 1) \rfloor$. We could get that by shifting r_0 to the left by $p = \lg m$ bits and then taking the p bits that were shifted to the left of the binary point.
- But, we don't need to shift. Just take the p most significant bits of r_0 .



How to choose A ?

- Another example: On board.
- How to choose A ?
 - The multiplication method works with any legal value of A .
 - But it works better with some values than with others, depending on the keys being hashed.
 - Knuth suggests using $A \approx (\sqrt{5} - 1)/2$.

