Chapter 11. Hash Tables

Joon Soo Yoo

May 6, 2025

Assignment

- Read §11.1, §11.2, §11.3.1
- Problems
  - §11.1 - 2
  - §11.2 - 2, 6

Chapter 11: Hash Tables

- **Chapter 11.1: Direct-Address Tables**
- Chapter 11.2: Hash Tables
- Chapter 11.3: Hash Functions
- Chapter 11.4: Open Addressing
- Chapter 11.5: Practical Considerations

Chapter 11.1: Direct-Address Tables

What is Direct Addressing?

- Simple and fast data structure when the universe of keys U is **small**.
- Use an array $T[0 \ldots m-1]$, where each index corresponds to a key in U.
- Example: If $U = \{0, 1, 2, 3, 4, 5\}$, make array $T[0 \ldots 5]$.

Direct Addressing Example

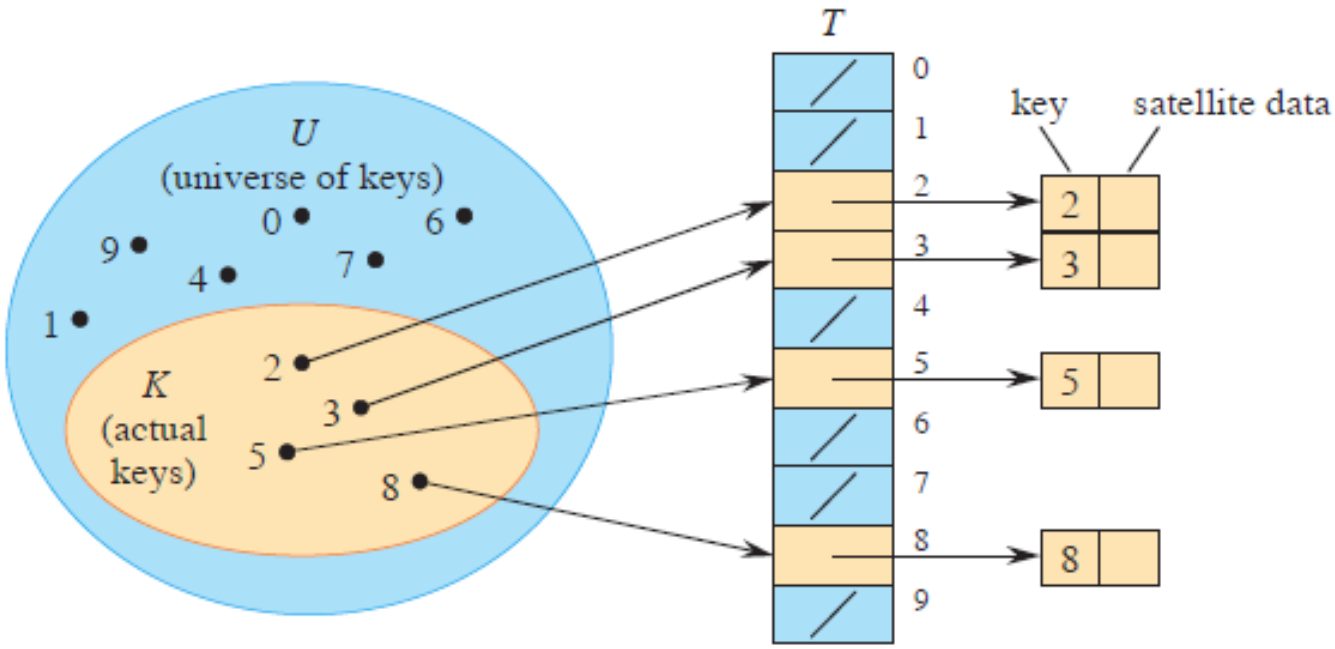

Figure 11.1 How to implement a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index into the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, in blue, contain NIL.

Direct-Address Tables Operations

Operations (All in $O(1)$ time):

- `DIRECT-ADDRESS-SEARCH`(T, k): Return $T[k]$.
- `DIRECT-ADDRESS-INSERT`(T, k): Set $T[k] = x$.
- `DIRECT-ADDRESS-DELETE`(T, k): Set $T[k] =$ NIL.

Drawback of Direct Addressing

When does it become impractical?

- If the universe U is **large**, direct addressing wastes too much memory.
- Example: $U = \{0, 1, 2, \ldots, 2^{64} - 1\} \rightarrow$ impossible to allocate array.
- Most entries are NIL $\rightarrow$ **huge memory waste**.

Conclusion:

- **Direct addressing is only suitable when U is small.**
- For large U, we need to use **Hash Tables** $\rightarrow$ map keys to small index space.

Chapter 11: Hash Tables

- Chapter 11.1: Direct-Address Tables
- **Chapter 11.2: Hash Tables**
- Chapter 11.3: Hash Functions
- Chapter 11.4: Open Addressing
- Chapter 11.5: Practical Considerations

Hash Tables: Motivation and Introduction

- **Direct addressing has limits**:
  - If the universe U of keys is large or infinite, creating an array of size $|U|$ is impractical.
  - Memory usage becomes wasteful, especially when the actual number of stored keys $|K|$ is small.

- **Hash tables offer a better solution**:
  - Storage reduces to $\Theta(|K|)$ while maintaining $O(1)$ average-case search time.
  - Uses a **hash function** $h\colon U \to \{0, 1, \ldots, m-1\}$ to map keys to table slots.
  - Table size $m \ll |U|$, saving significant memory.

- **Example of a hash function**:
  - Simple version: $h(k) = k \mod m$
  - Not necessarily good (may lead to poor distribution).

Collisions and the Need for Resolution

- **Collision**: Two distinct keys map to the same hash slot.
- **Why collisions are unavoidable**:
  - By pigeonhole principle: $|U| > m \Rightarrow$ at least two keys share a slot.
  - Cannot design a hash function that avoids collisions altogether.
- **Goal of good hash functions**:
  - Make collisions **rare** $\rightarrow$ random-looking hash function.
  - Deterministic: same input always produces the same output.
- **Next topic**:
  - We need a technique to resolve collisions $\rightarrow$ **Chaining** and **Open addressing**.
  - Before that, define **independent uniform hashing** to model randomness.

Collisions Diagram

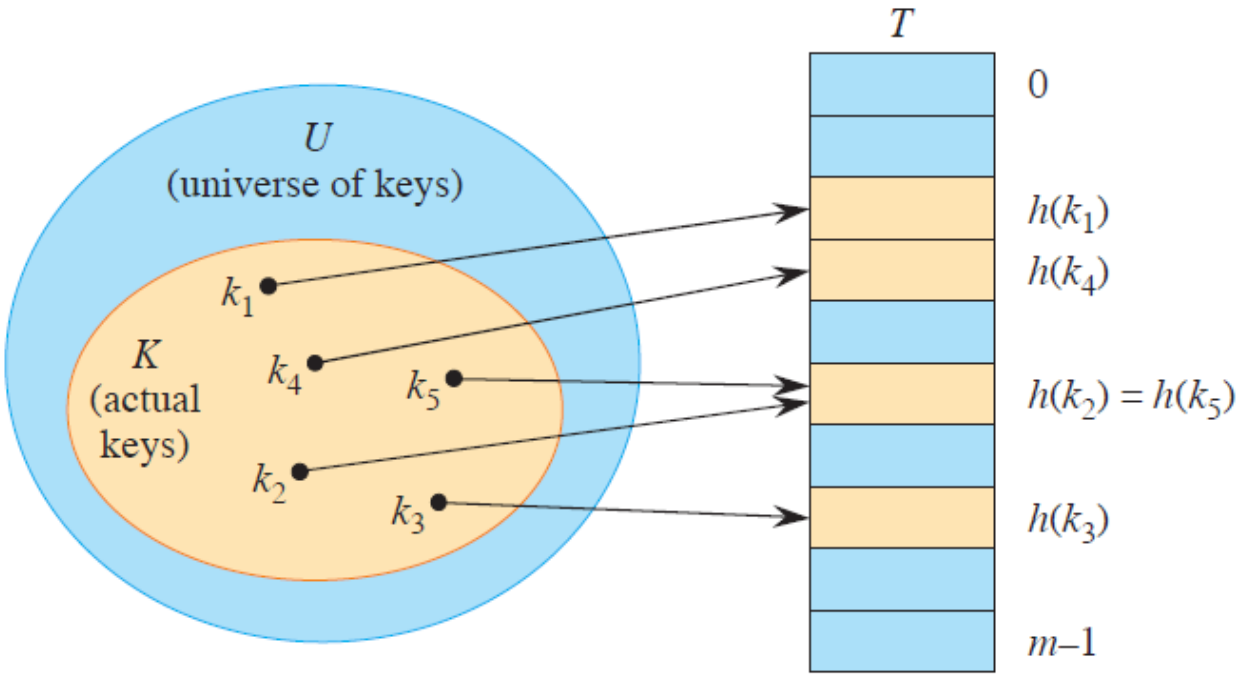

Figure 11.2 Using a hash function h to map keys to hash-table slots. Because keys k_2 and k_5 map to the same slot, they collide.

Independent Uniform Hashing

Ideal Hash Function Concept

- Each key k in universe $U \rightarrow$ hash function $h(k)$ outputs a value in $\{0, 1, 2, \ldots, m-1\}$.
- Values are chosen:
  - **Uniformly**: Each slot equally likely.
  - **Independently**: Hash value of k_1 gives no clue about k_2.
- Same key always produces same hash $\rightarrow$ deterministic per key.

Independent Uniform Hashing: Ideal vs. Impractical

Why Ideal?

- Avoids clustering $\rightarrow$ collisions are purely random.
- Ensures good average-case performance.
- Analysis becomes simple and clean.

Why Impractical?

- Requires storing random values for **every possible key** in U.
- If $|U|$ is large (e.g., 2^{64}), storage becomes infeasible.
- $\rightarrow$ **Would need infinite or impractical amount of memory.**

Practical Implications

- Perfect independent uniform hashing is impossible.
- We design hash functions that are **"close enough"**:
  - Universal hashing
  - Randomized hashing
  - Multiplication and shift methods

Collision Resolution by Chaining: Overview

- When multiple keys hash to the same slot, a **collision** occurs.
- **Chaining** resolves collisions by storing keys in a **linked list** at each slot.
- If slot $T[j]$ has no keys $\rightarrow$ `NIL`.

Idea: Divide n elements into m subsets (expected size $n/m = \alpha$), each managed independently as a list.

Chaining Diagram

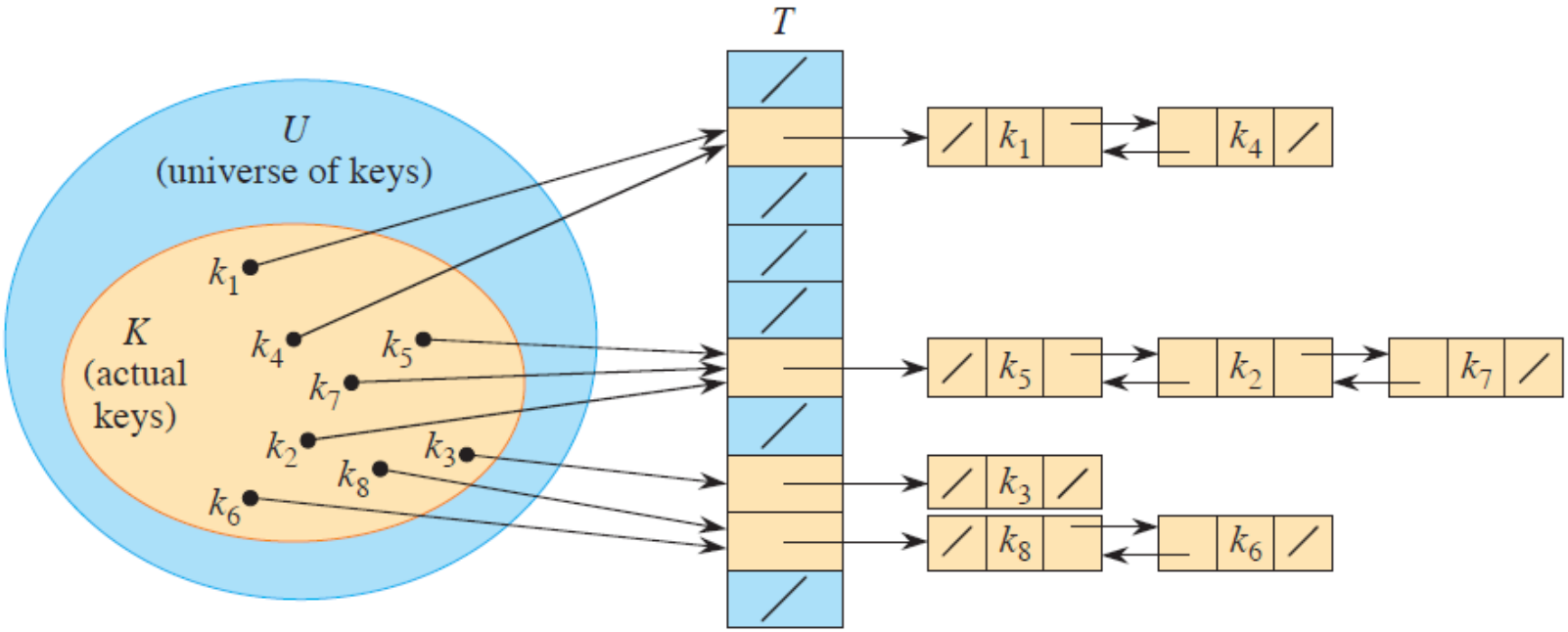

Figure 11.3 Collision resolution by chaining. Each nonempty hash-table slot $T[j]$ points to a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_2) = h(k_7)$. The list can be either singly or doubly linked. We show it as doubly linked because deletion may be faster that way when the deletion procedure knows which list element (not just which key) is to be deleted.

Chaining Example (Conceptual)

- $T[0 \ldots m-1]$ is the hash table array.
- $T[j]$ points to a linked list storing all keys k such that $h(k) = j$.
- This reduces storage from $|U|$ to $\Theta(|K|)$.

Hash Table Slot $\rightarrow$ Linked List of Keys

Hash Table Operations with Chaining

```
CHAINED-HASH-INSERT(T, x)
1  LIST-PREPEND(T[h(x.key)], x)

CHAINED-HASH-SEARCH(T, k)
1  return LIST-SEARCH(T[h(k)], k)

CHAINED-HASH-DELETE(T, x)
1  LIST-DELETE(T[h(x.key)], x)
```

Hash Table Operations with Chaining Time

Insertion (CHAINED-HASH-INSERT)

- Insert at the front of the linked list $\rightarrow \Theta(1)$ time.

Search (CHAINED-HASH-SEARCH)

- Search through the linked list $\rightarrow$ Expected $\Theta(1+\alpha)$ time.

Deletion (CHAINED-HASH-DELETE)

- If doubly linked list $\rightarrow \Theta(1)$ time.

Chaining: Performance Analysis Overview

- Hash table T with m slots, storing n elements.
- Define **load factor**:
$$\alpha = \frac{n}{m}$$
- α represents the average number of elements per chain.
- Example: $n = 10$, $m = 5 \Rightarrow \alpha = 2 \rightarrow$ each chain has ~ 2 elements on average.

Worst-Case Scenario

- All n elements hash to the same slot $\rightarrow$ one long chain.
- Searching time becomes:

$$\Theta(n)$$

- No better than using a simple linked list.
- **Hashing is NOT optimized for the worst case.**

Average-Case Scenario: Key Assumptions

- To analyze average-case, assume:
  - **Uniform hashing**: each key is equally likely to hash to any slot.
  - **Independent hashing**: keys hash independently of each other.
- Combined, this is called:

Independent Uniform Hashing

$\rightarrow$ Ideal assumption for analysis.

Analyzing Unsuccessful Search: Setup

Scenario: Searching for a key k that is not stored in the table.

- Hash table $T[0 \ldots m-1]$ with m slots.
- Each slot j has a linked list of length n_j.
- Total elements stored:

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

- **Load factor:**

$$\alpha = \frac{n}{m}$$

$\rightarrow$ average number of elements per slot.

Analyzing Unsuccessful Search: Observation

Observation:

- Hash function $h(k)$ gives a random slot j.
- Expected number of elements in $T[j]$ is:

$$E[n_j] = \alpha$$

- To search unsuccessfully $\rightarrow$ must scan entire linked list at $T[j]$.
- Scanning takes $O(\alpha)$ time.

Analyzing Unsuccessful Search: Total Time

Total time for unsuccessful search:

$$\underbrace{O(1)}_{\text{compute hash}} + \underbrace{O(\alpha)}_{\text{scan linked list}} = \boxed{\Theta(1+\alpha)}$$

Summary: Unsuccessful search depends on:

- O(1) hash computation
- O(α) average list length

Theorem 11.1: Unsuccessful Search Time

Theorem

In a hash table using chaining and independent uniform hashing, the expected time for an **unsuccessful search** *is:*

$$\boxed{\Theta(1+\alpha)}$$

where $\alpha = \frac{n}{m}$ *is the load factor.*

Proof of Theorem 11.1

Proof.

- Unsuccessful search $\rightarrow$ key not in table $\rightarrow$ hashes to a random slot.
- By uniform hashing:

$$E[\text{length of slot list}] = \alpha$$

- Searching scans the entire list:

$$O(\alpha) + O(1) = \Theta(1 + \alpha)$$

Successful Search in Hashing with Chaining: Overview

Scenario: Searching for a key that is **already stored**.

- Unlike unsuccessful search, the search **cannot go to an empty slot**.
- Longer chains are **more likely to be searched**, because elements in longer chains are more numerous.
- Goal: Compute expected number of elements checked when searching for the target element.

Key idea: Expected cost is still $\Theta(1 + \alpha)$.

Step 1: Setup the Problem

- There are n elements inserted: $x_1, x_2, \ldots, x_n$.
- Each element has a key k_i and is inserted at the **front** of chain $h(k_i)$.
- Searching for $x_i \rightarrow$ need to check all elements **before** x_i **in the chain**.

Observation: Elements before x_i are those inserted **after** x_i that hash to the same slot.

Step 2: Define Indicator Variable X_{ijq}

Definition:

$$X_{ijq} = \begin{cases} 1 & \text{if search is for } x_i, \text{ and } x_j \text{ (with } i < j) \text{ hashes to same slot } q \\ 0 & \text{otherwise} \end{cases}$$

Probability:

$$\mathbb{E}[X_{ijq}] = \frac{1}{n} \cdot \frac{1}{m} \cdot \frac{1}{m} = \frac{1}{nm^2}$$

Interpretation: This captures chance that x_j is before x_i in same slot during search.

Step 3: Define Variable Y_j for Each Element

Definition of Y_j:

$$Y_j = \begin{cases} 1 & \text{if } x_j \text{ appears BEFORE the element being searched in its chain} \\ 0 & \text{otherwise} \end{cases}$$

Mathematically:

$$Y_j = \sum_{q=0}^{m-1} \sum_{i=1}^{j-1} X_{ijq}$$

Idea:

- For each element x_j, check all previously inserted elements x_i $(i < j)$.
- If x_i and x_j hash to the same slot and x_i is being searched $\rightarrow$ x_j is "before" x_i in the chain.
- Y_j counts whether x_j is one of those "before" elements.

Step 4: Define Total Number Before (Variable Z)

Definition:

$$Z = \sum_{j=1}^{n} Y_j$$

Meaning: Total number of elements that appear before the searched element in the chain.

Successful search cost:

$$\text{Total number checked} = Z + 1$$

(+1 to include the target element itself)

Step 5: Calculate Expected Search Cost

Compute:

$$\mathbb{E}[Z+1] = 1 + \sum_{q=0}^{m-1} \sum_{j=1}^{n} \sum_{i=1}^{j-1} \mathbb{E}[X_{ijq}]$$

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

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

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

where $\alpha = \frac{n}{m}$.

Theorem 11.2 (Successful Search Time)

Result:

- In a hash table with chaining and independent uniform hashing:
- A successful search takes $\Theta(1+\alpha)$ time on average.

Explanation:

- $\alpha/2$ expected number of elements before $\rightarrow$ dominated by α.
- $+1$ for accessing target $\rightarrow$ total is $\Theta(1+\alpha)$.

Conclusion:

$\boxed{\Theta(1+\alpha)}$ time on average for successful search.

Chapter 11: Hash Tables

- Chapter 11.1: Direct-Address Tables
- Chapter 11.2: Hash Tables
- **Chapter 11.3: Hash Functions**
- Chapter 11.4: Open Addressing
- Chapter 11.5: Practical Considerations

Hash Functions: Why They Matter

Good hashing needs good hash functions.

- Must be **efficient to compute**.
- Must distribute keys well $\rightarrow$ to avoid long chains.

Main Question:

- How do we design good hash functions?

Ad Hoc (Static) Hashing Methods

Two simple methods:

- **Division method:** $h(k) = k \mod m$
- **Multiplication method:** $h(k) = \lfloor m(kA \mod 1) \rfloor$

Limitation:

- Fixed functions $\rightarrow$ may work well for some datasets.
- **But bad for others $\rightarrow$ clustering can occur.**
- Called **static hashing** $\rightarrow$ no adaptation.

Random Hashing: A Smarter Approach

Idea: Use randomness to avoid bad cases.

- Create a **family of hash functions**.
- At runtime, choose **randomly** from the family.
- Makes it hard for input keys to cause bad collisions.

Benefits:

- Provably good average-case performance.
- Similar idea to **randomized quicksort** $\rightarrow$ avoids worst-case.

Why Universal Hashing? (Motivation)

Problem with just random hashing:

- Randomly picking a hash function *sounds good*, but...
- No formal guarantee on collision probability.
- Could still have bad cases unless we design carefully.

Goal:

- We want a hash function family with **provable guarantees**.
- Specifically: avoid collisions as much as possible.

Universal Hashing: Formal Definition

Definition: Universal Hash Family

- Let $\mathcal{H}$ be a family of hash functions mapping $U \to \{0, 1, \ldots, m-1\}$.
- $\mathcal{H}$ is **universal** if for any two distinct keys $k_1, k_2 \in U$:

$$\Pr_{h \in \mathcal{H}}[h(k_1) = h(k_2)] \leq \frac{1}{m}$$

- Probability is over the random choice of h from $\mathcal{H}$.

Implication:

- Provably bounds the average number of collisions.
- Guarantees predictable average-case performance.

Hashing Strategies Summary

Hash Function Strategies

- **Static hashing** (fixed h):
  - No randomness $\rightarrow$ bad worst-case possible.
- **Random hashing** (random h chosen at runtime):
  - Avoids worst-case inputs $\rightarrow$ good in practice.
- **Universal hashing** (random h from universal family):
  - Provable collision bound $\leq 1/m \rightarrow$ good average case for *any* input.

Static Hashing: Overview

- **Static hashing**: Uses a single, fixed hash function.
- No randomization except from the key distribution.
- Works well if keys are randomly distributed.
- **Problem:** Can perform poorly with adversarial or clustered keys.

Division Method (Static Hashing)

Definition:

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

- Maps key k to slot $0 \ldots m-1$.
- Very fast (only requires a division operation).

Example:

$$k = 100, \quad m = 12$$

$$h(100) = 100 \mod 12 = 4$$

Division Method: Pros and Cons

Advantages:

- Extremely simple and fast.
- Good when m is a prime number.

Disadvantages:

- Poor performance if m is a power of 2 $\rightarrow$ clustering.
- No guarantee of good average-case performance.
- Forces table size m to be prime $\rightarrow$ inflexible for resizing.

Multiplication Method for Hashing

Overview:

- Alternative to division method.
- Works well even when m is not prime.
- Uses high bits of multiplication $\rightarrow$ better randomness.

Hash function definition:

$$h(k) = \lfloor m \cdot (k \cdot A \mod 1) \rfloor$$

- A: A constant in range $0 < A < 1$.
- $k \cdot A \mod 1$: Fractional part of $k \cdot A$.
- Multiply fractional part by m and take floor.

Step-by-Step Process

Given:

- Key k and table size m.
- Constant A (usually suggested by Knuth as $A = \frac{\sqrt{5}-1}{2}$).

Steps:

1. Compute $k \cdot A \rightarrow$ gives a real number.
2. Extract fractional part $\rightarrow k \cdot A - \lfloor k \cdot A \rfloor$.
3. Multiply fractional part by m.
4. Take floor of result $\rightarrow$ gives slot index $h(k)$.

Advantages of Multiplication Method

- **No restriction on m:** Can be power of 2 $\rightarrow$ efficient for computers.
- **Better distribution:** Uses fractional part $\rightarrow$ reduces clustering.
- **Simple to implement:** Only multiplication, subtraction, and floor operation.

Question?