# 7. Hash Tables

# Goals

- Understand hash tables and hash functions
- Learn how to resolve collisions
- Understand search time in hash tables

# Time Complexity of Search

- Array
  - $O(n)$
- Binary search trees
  - Worst-case $\Theta(n)$
  - Average-case $\Theta(\log n)$
- Balanced binary search trees (red-black tree)
  - Worst-case $\Theta(\log_2 n)$
- B-trees
  - Worst-case $\Theta(\log_k n)$
- Hash table
  - Average-case $\Theta(1)$

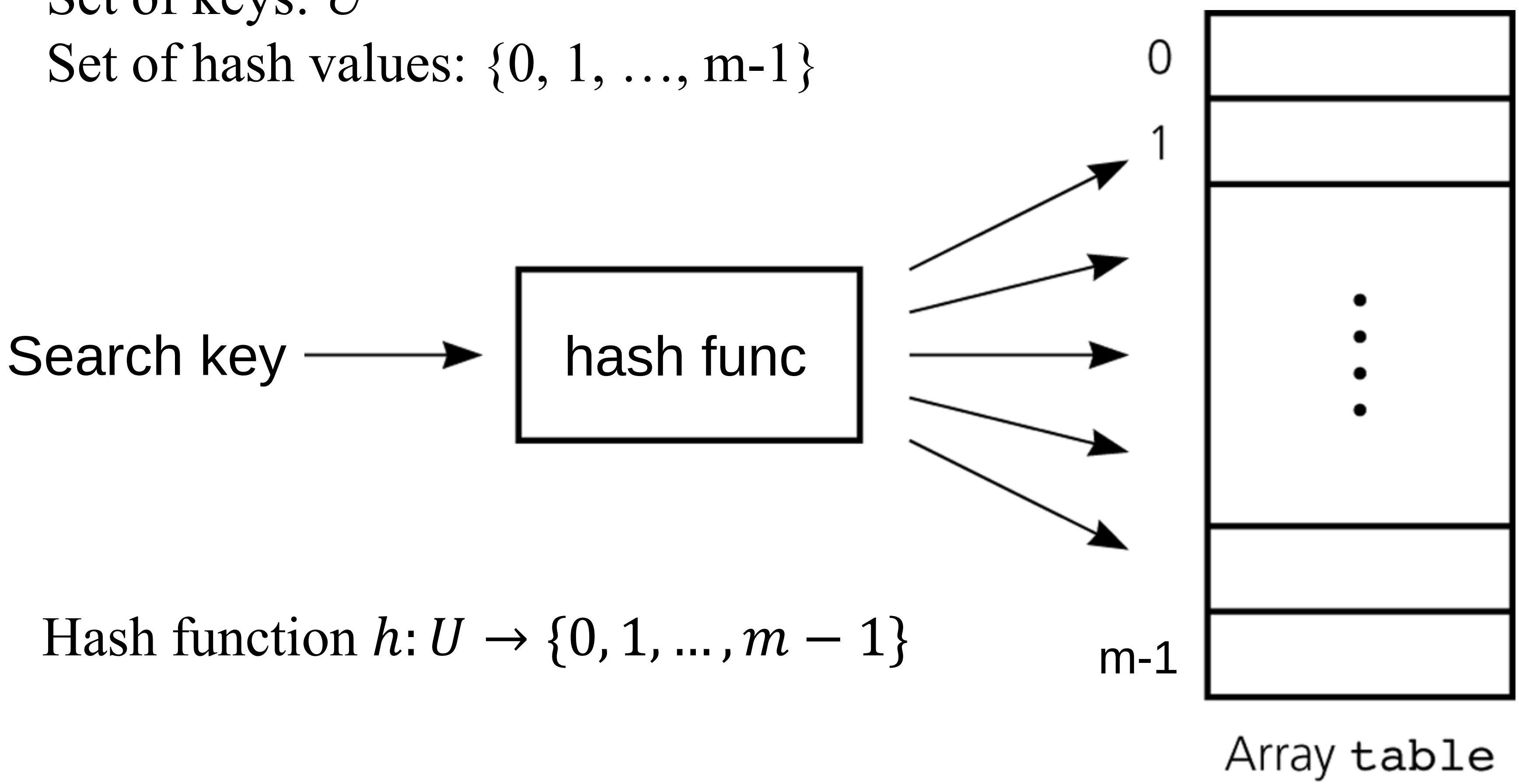
# Hash Tables

- Data structure in which the location for a key is determined by the value of the key
- Search, insert, delete in expected O(1) time
- Useful in applications where fast response is important.
- Don’t support operations like finding minimum or maximum

# Hash Function

Set of keys: $U$

Set of hash values: {0, 1, …, m-1}




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

# Hash Table

Insert: 25, 13, 16, 15, 7

| | |
|---|---|
| 0 | 13 |
| 1 | |
| 2 | 15 |
| 3 | 16 |
| 4 | |
| 5 | |
| 6 | |
| 7 | 7 |
| 8 | |
| 9 | |
| 10 | |
| 11 | |
| 12 | 25 |

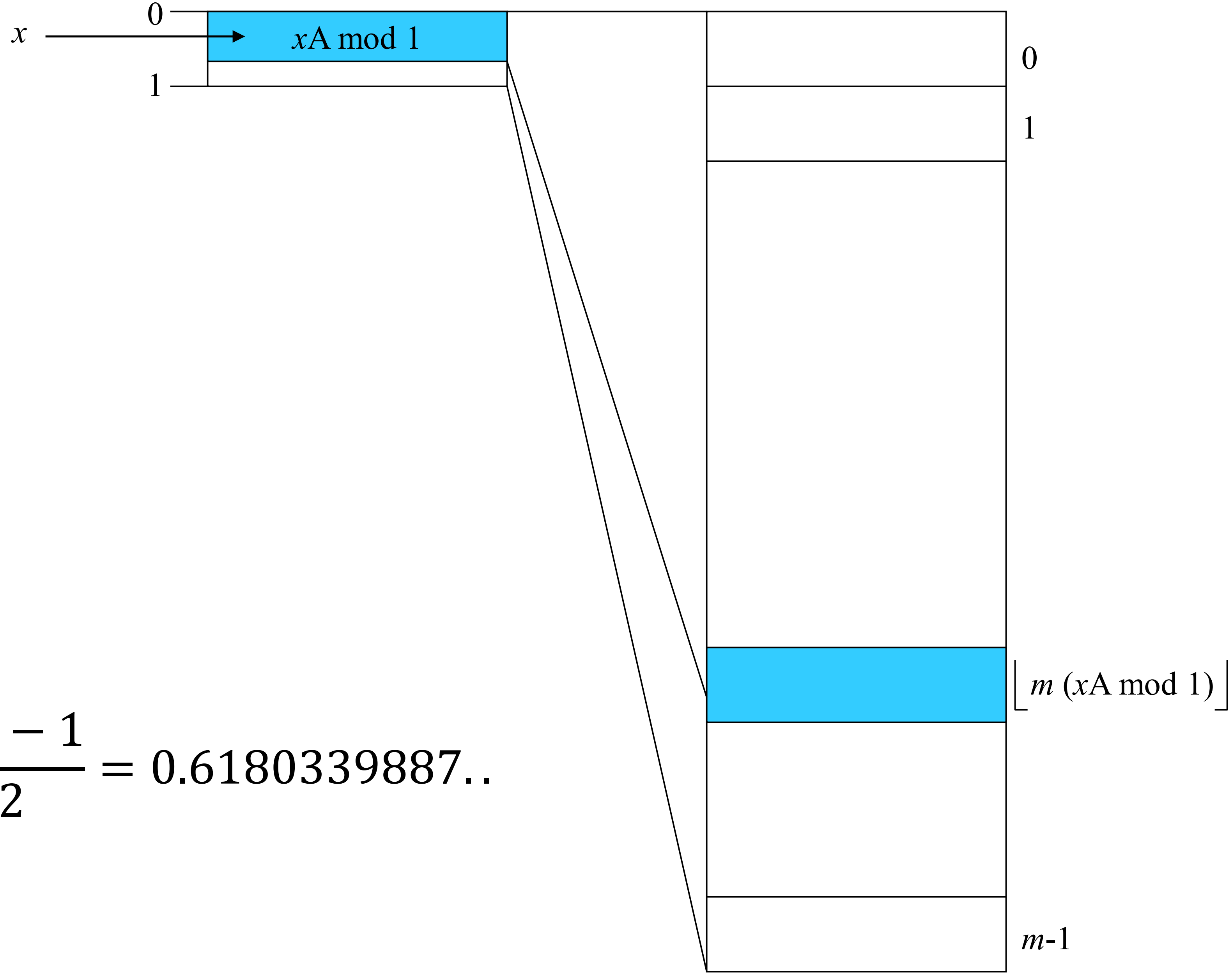
Hash function:
$h(x) = x \bmod 13$

# Hash Function

- Keys should be evenly distributed over hash table.
- Computation of hash function should be simple.
- Commonly used methods
  - Division method
  - Multiplication method

# Hash Function

- Division method
  - $h(x) = x \bmod m$
  - $m$: size of hash table. prime in most cases.
- Multiplication method
  - $h(x) = (xA \bmod 1) * m$
  - $A$: constant such that $0 < A < 1$
  - $m$: not necessarily prime. typically $2^p$ for some integer $p$




$$A: \frac{\sqrt{5}-1}{2} = 0.6180339887..$$

# Collision

- Two keys may hash to the same slot.
- Methods to resolve collisions
  - Chaining
  - Open Addressing

# Collision

Insert: 25, 13, 16, 15, 7, 29

| | |
|---|---|
| 0 | 13 |
| 1 | |
| 2 | 15 |
| 3 | 16 |
| 4 | |
| 5 | |
| 6 | |
| 7 | 7 |
| 8 | |
| 9 | |
| 10 | |
| 11 | |
| 12 | 25 |

$h(29) = 29 \bmod 13 = 3$

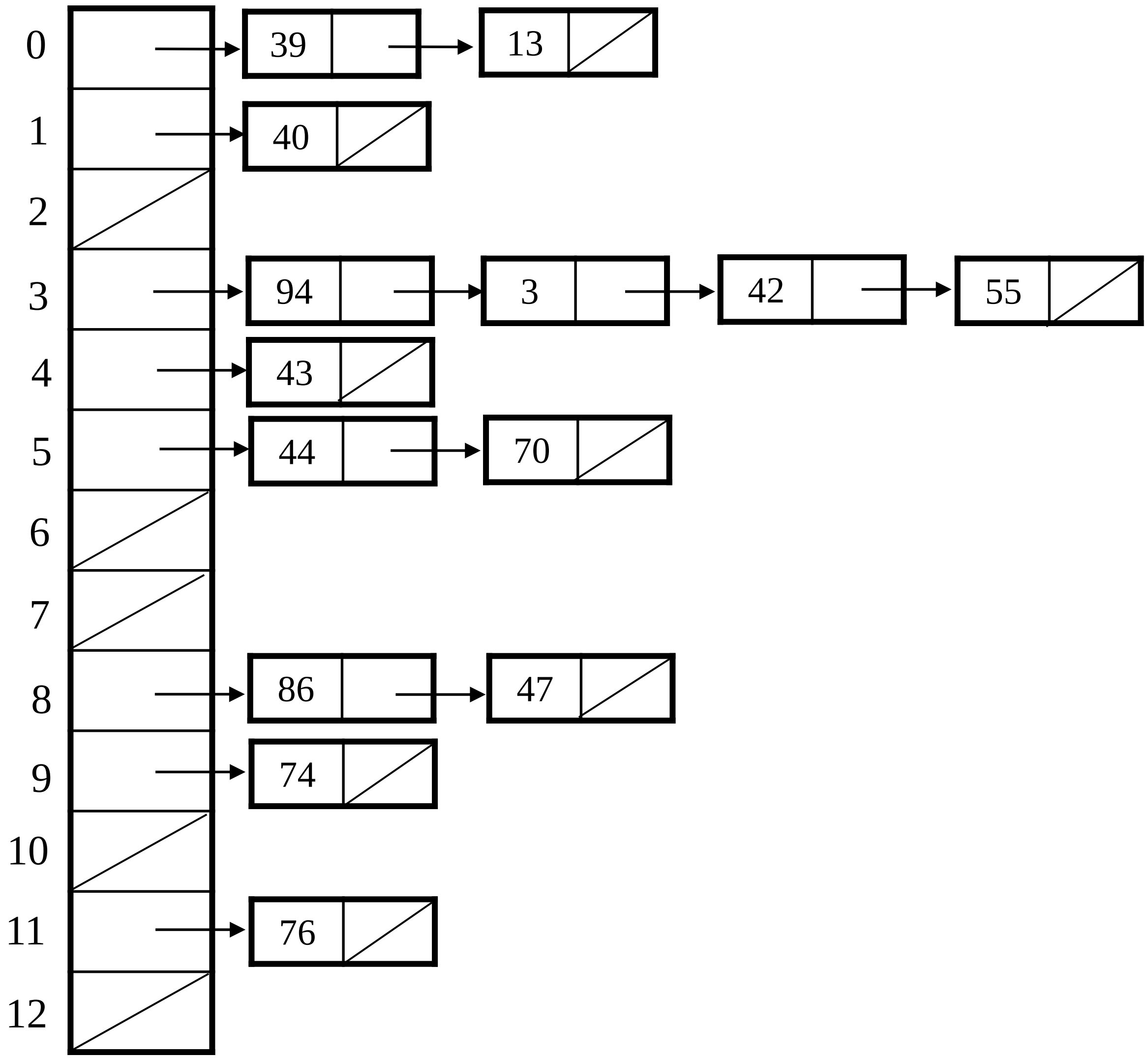
When inserting 29, we've found a collision!

Hash function $h(x) = x \bmod 13$

# Collision Resolution

- Chaining
  - Uses a linked list to store all keys that hash to the same slot
  - Requires linked lists in addition to hash table
- Open addressing
  - Resolves collisions inside hash table
  - Doesn’t require additional space

# Chaining



# Open Addressing

- Successively generate hash values until we find an empty slot
  - $h_0(x), h_1(x), h_2(x), h_3(x), \ldots$
- Commonly used methods
  - linear probing
  - quadratic probing
  - double hashing

# Linear Probing

$h_i(x) = (h(x) + i) \bmod m$

Insert: 25, 13, 16, 15, 7, 28, 31, 20, 1, 38

| | |
|---|---|
| 0 | 13 |
| 1 | |
| 2 | 15 |
| 3 | 16 |
| 4 | 28 |
| 5 | |
| 6 | |
| 7 | 7 |
| 8 | |
| 9 | |
| 10 | |
| 11 | |
| 12 | 25 |

| | |
|---|---|
| 0 | 13 |
| 1 | |
| 2 | 15 |
| 3 | 16 |
| 4 | 28 |
| 5 | 31 |
| 6 | |
| 7 | 7 |
| 8 | 20 |
| 9 | |
| 10 | |
| 11 | |
| 12 | 25 |

| | |
|---|---|
| 0 | 13 |
| 1 | 1 |
| 2 | 15 |
| 3 | 16 |
| 4 | 28 |
| 5 | 31 |
| 6 | 38 |
| 7 | 7 |
| 8 | 20 |
| 9 | |
| 10 | |
| 11 | |
| 12 | 25 |

$h_i(x) = (h(x) + i) \bmod 13$

# Linear probing suffers from primary clustering

Primary clustering: long run of occupied slots

| | |
|---|---|
| 0 | |
| 1 | |
| 2 | 15 |
| 3 | 16 |
| 4 | 28 |
| 5 | 31 |
| 6 | 44 |
| 7 | |
| 8 | |
| 9 | |
| 10 | |
| 11 | 37 |
| 12 | |

Example of primary clustering

# Quadratic Probing

$$h_i(x) = (h(x) + c_1 i^2 + c_2 i) \bmod m$$

Insert 15, 18, 43, 37, 45, 30

| | |
|---|---|
| 0 | |
| 1 | |
| 2 | 15 |
| 3 | |
| 4 | 43 |
| 5 | 18 |
| 6 | 45 |
| 7 | |
| 8 | 30 |
| 9 | |
| 10 | |
| 11 | 37 |
| 12 | |

$$h_i(x) = (h(x) + i^2) \bmod 13$$

# Quadratic probing suffers from secondary clustering

Secondary clustering: initial hash value determines entire sequence

| | |
|---|---|
| 0 | |
| 1 | |
| 2 | 15 |
| 3 | 28 |
| 4 | |
| 5 | 54 |
| 6 | 41 |
| 7 | |
| 8 | 21 |
| 9 | |
| 10 | |
| 11 | 67 |
| 12 | |

Example of secondary clustering

# Double Hashing

$$h_i(x) = (h(x) + i\,f(x)) \bmod m$$

Insert 15, 19, 28, 41, 67

| | |
|---|---|
| 0 | |
| 1 | |
| 2 | 15 |
| 3 | 67 |
| 4 | |
| 5 | |
| 6 | 19 |
| 7 | |
| 8 | 28 |
| 9 | |
| 10 | 41 |
| 11 | |
| 12 | |

$h_0(15) = h_0(28) = h_0(41) = h_0(67) = 2$

$h_1(67) = 3$

$h_1(28) = 8$

$h_1(41) = 10$

$h(x) = x \bmod 13$

$f(x) = x \bmod 11$

$h_i(x) = (h(x)+i f(x)) \bmod 13$

# Caveat in Deletion

$h(x) = x \bmod 13$

| | |
|---|---|
| 0 | 13 |
| 1 | 1 |
| 2 | 15 |
| 3 | 16 |
| 4 | 28 |
| 5 | 31 |
| 6 | 38 |
| 7 | 7 |
| 8 | 20 |
| 9 | |
| 10 | |
| 11 | |
| 12 | 25 |

(a) Delete 1

| | |
|---|---|
| 0 | 13 |
| 1 | |
| 2 | 15 |
| 3 | 16 |
| 4 | 28 |
| 5 | 31 |
| 6 | 38 |
| 7 | 7 |
| 8 | 20 |
| 9 | |
| 10 | |
| 11 | |
| 12 | 25 |

(b) Search 38, problem!

| | |
|---|---|
| 0 | 13 |
| 1 | DELETED |
| 2 | 15 |
| 3 | 16 |
| 4 | 28 |
| 5 | 31 |
| 6 | 38 |
| 7 | 7 |
| 8 | 20 |
| 9 | |
| 10 | |
| 11 | |
| 12 | 25 |

(c) Mark deletion

# Search Time in Hash Table

- load factor $\alpha$
  - Indicates how much of hash table is filled
  - If $n$ keys are stored in a hash table of size $m$, $\alpha = n/m$.
- Search efficiency in a hash table is related to load factor.

# Search Time in Chaining

- Theorem 1
  - In a hash table with chaining, an unsuccessful search takes average-case $\Theta(1 + \alpha)$ time (under assumption of simple uniform hashing: each key is equally likely to hash into any of *m* slots).
- Theorem 2
  - In a hash table with chaining, a successful search takes average-case $\Theta(1 + \alpha)$ time (expected number of keys examined is $1 + \alpha/2 - \alpha/2n$).

# Search Time in Open Addressing

- Assume uniform hashing
  - Probe sequence of each key (i.e., $h_0(x)$, $h_1(x)$, …, $h_{m-1}(x)$) is equally likely to be any of $m!$ permutations of (0, 1, …, $m$-1)
- Theorem 3
  - In a hash table with open addressing, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$.
- Theorem 4
  - In a hash table with open addressing, the expected number of probes in a successful search is at most $(1/\alpha)\log(1/(1-\alpha))$.

# When Load Factor is High

- If the load factor of a hash table gets high, efficiency of hash table deteriorates.
- General solution: a threshold is set in advance, and if load factor reaches the threshold,
  - Double the size of hash table (allocate a new table)
  - Rehash all keys and store them in new hash table

# Thank you