Lecture 13.
Hash Tables

Introduction to Algorithms

Da Nang University of Science and Technology

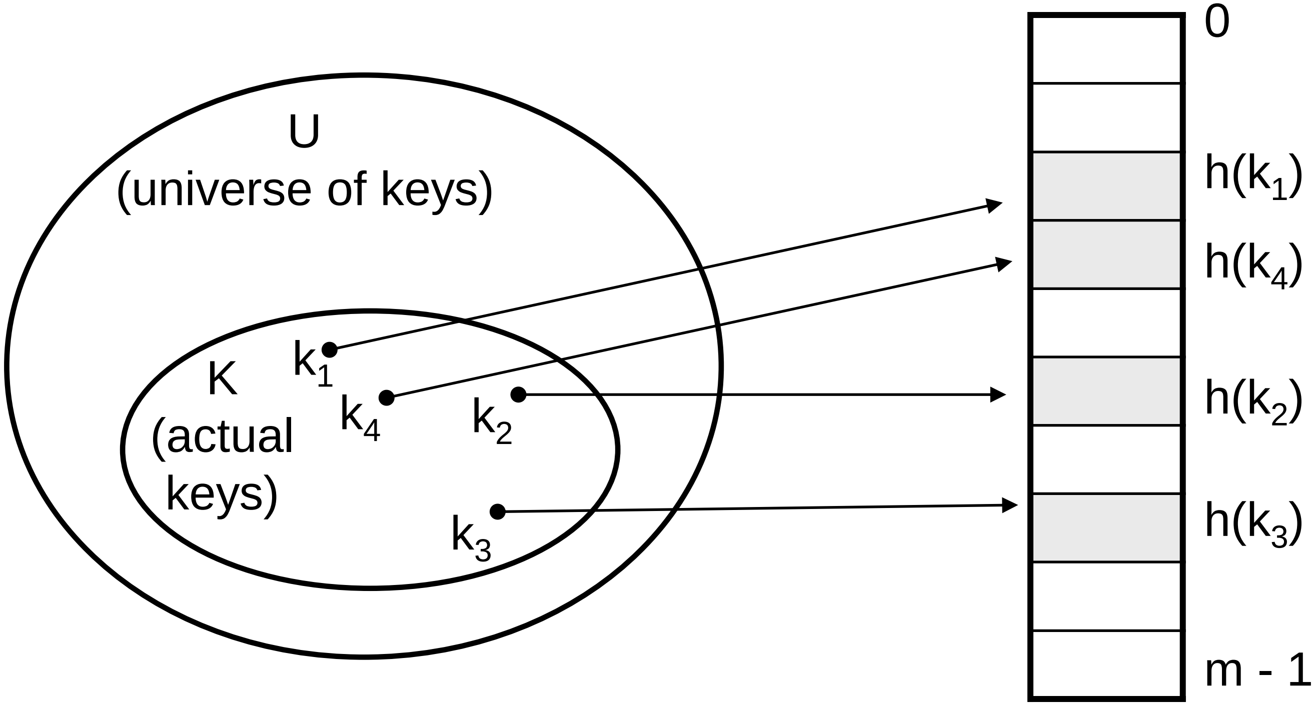
Dang Thien Binh

dtbinh@dut.udn.vn

Hash Tables

- Motivation: symbol tables
  - A compiler uses a *symbol table* to relate symbols to associated data
    - Symbols: variable names, procedure names, etc.
    - Associated data: memory location, call graph, etc.
  - For a symbol table (also called a *dictionary*), we care about search, insertion, and deletion
  - We typically don't care about sorted order

Hash Tables



Hash Tables

- Idea
  - Use a function h to compute the slot for each key
  - Store the element in slot h(k)
- A **hash function** h transforms a key into an index in a hash table T[0...m-1]:

  $h : U \rightarrow \{0, 1, . . . , m - 1\}$
- We say that k **hashes** to slot h(k)
- Advantages:
  - Reduce the range of array indices handled: m instead of |U|
  - Storage is correspondingly reduced

Hash Tables

- More formally:
  - Given a table T and a record x, with key (= symbol) and satellite data, we need to support:
    - Insert (T, x)
    - Delete (T, x)
    - Search(T, x)
  - We want these to be fast, but don't care about sorting the records
- The structure we will use is a *hash table*
  - Supports all the above in O(1) expected time !

Direct-Address Tables

- Suppose:
  - The range of keys is 0..m-1
  - Keys are distinct
- The idea:
  - Set up an array T[0..m-1] in which
    - T[i] = x if $x \in T$ and key[x] = i
    - T[i] = NULL otherwise
  - This is called a *direct-address table*
    - Operations take O(1) time!
    - *So what's the problem?*

Direct-Address Tables

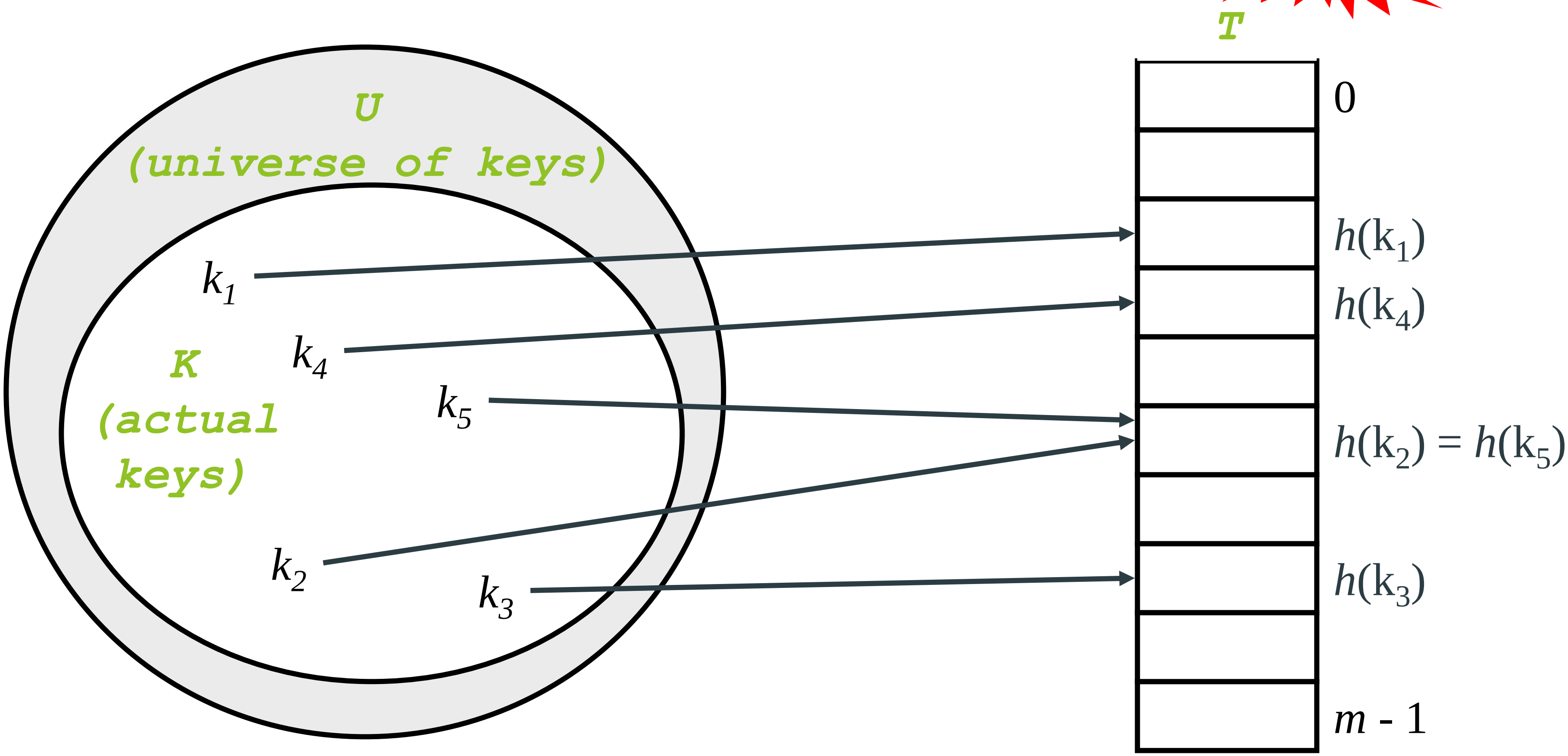
- Dictionary operations are trivial and take O(1) time each:
  - DIRECT-ADDRESS-SEARCH(T, k)
    return T [k]
  - DIRECT-ADDRESS-INSERT(T, x)
    T [key[x]] $\leftarrow$ x
  - DIRECT-ADDRESS-DELETE(T, x)
    T [key[x]] $\leftarrow$ NIL

The Problem With Direct Addressing

- Direct addressing works well when the range *m* of keys is relatively small
- But what if the keys are 32-bit integers?
  - Problem 1: direct-address table will have 2^{32} entries, more than 4 billion
  - Problem 2: even if memory is not an issue, the time to initialize the elements to NULL may be …
- Solution: map keys to smaller range 0..*m*-1
- This mapping is called a *hash function*

Hash Functions

- Next problem: *collision*



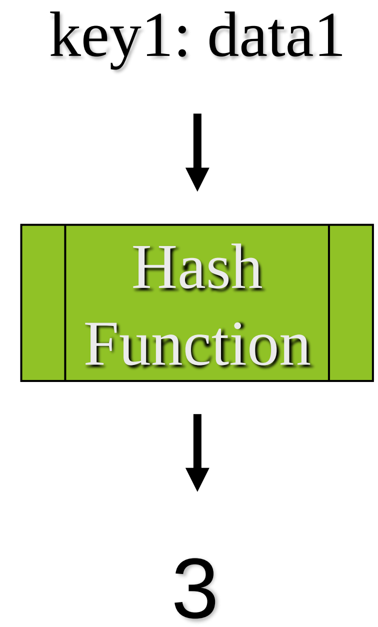
Resolving Collisions

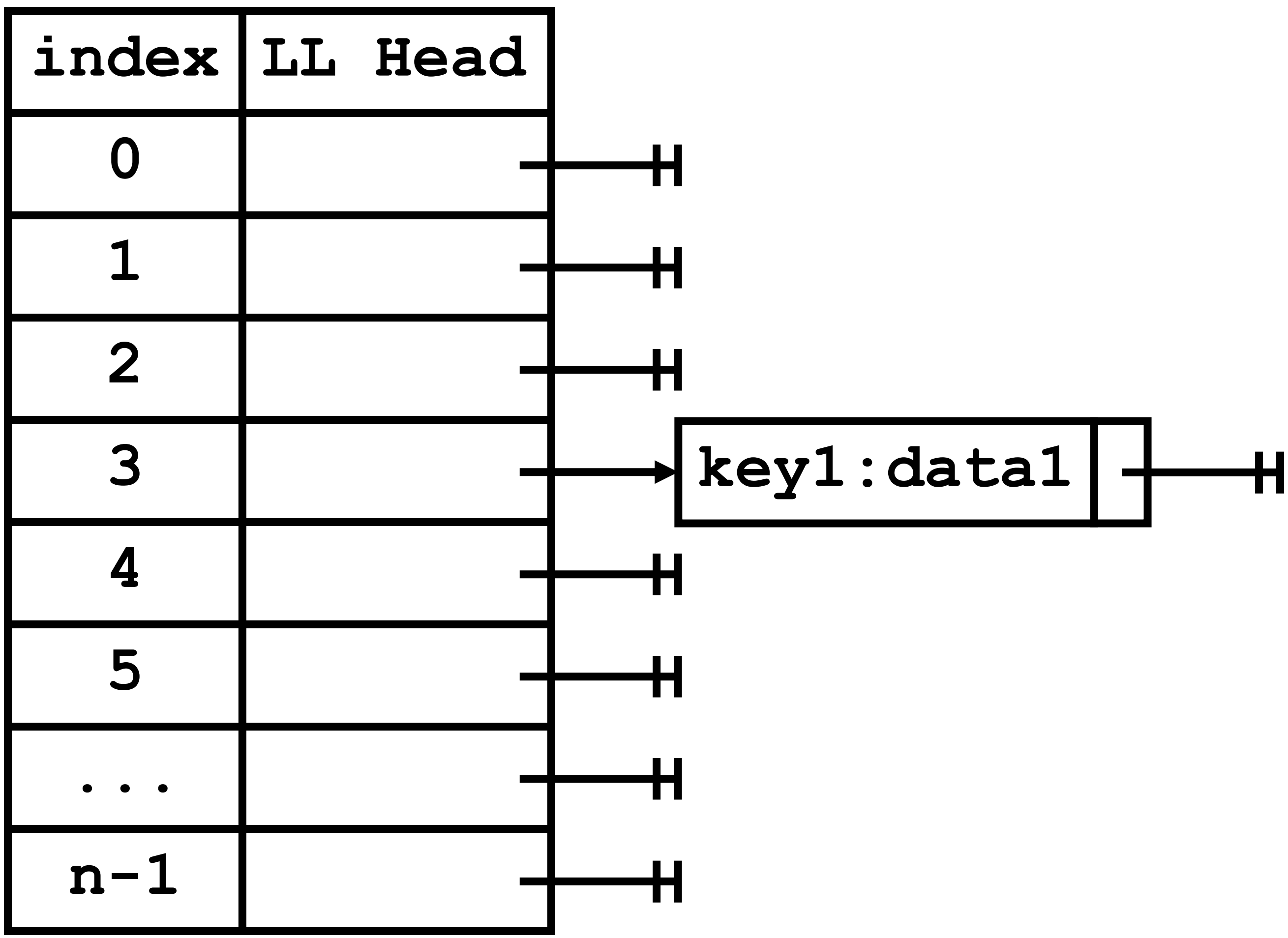
- *How can we solve the problem of collisions?*
  - Solution 1: *chaining*
  - Solution 2: *open addressing*

Chaining

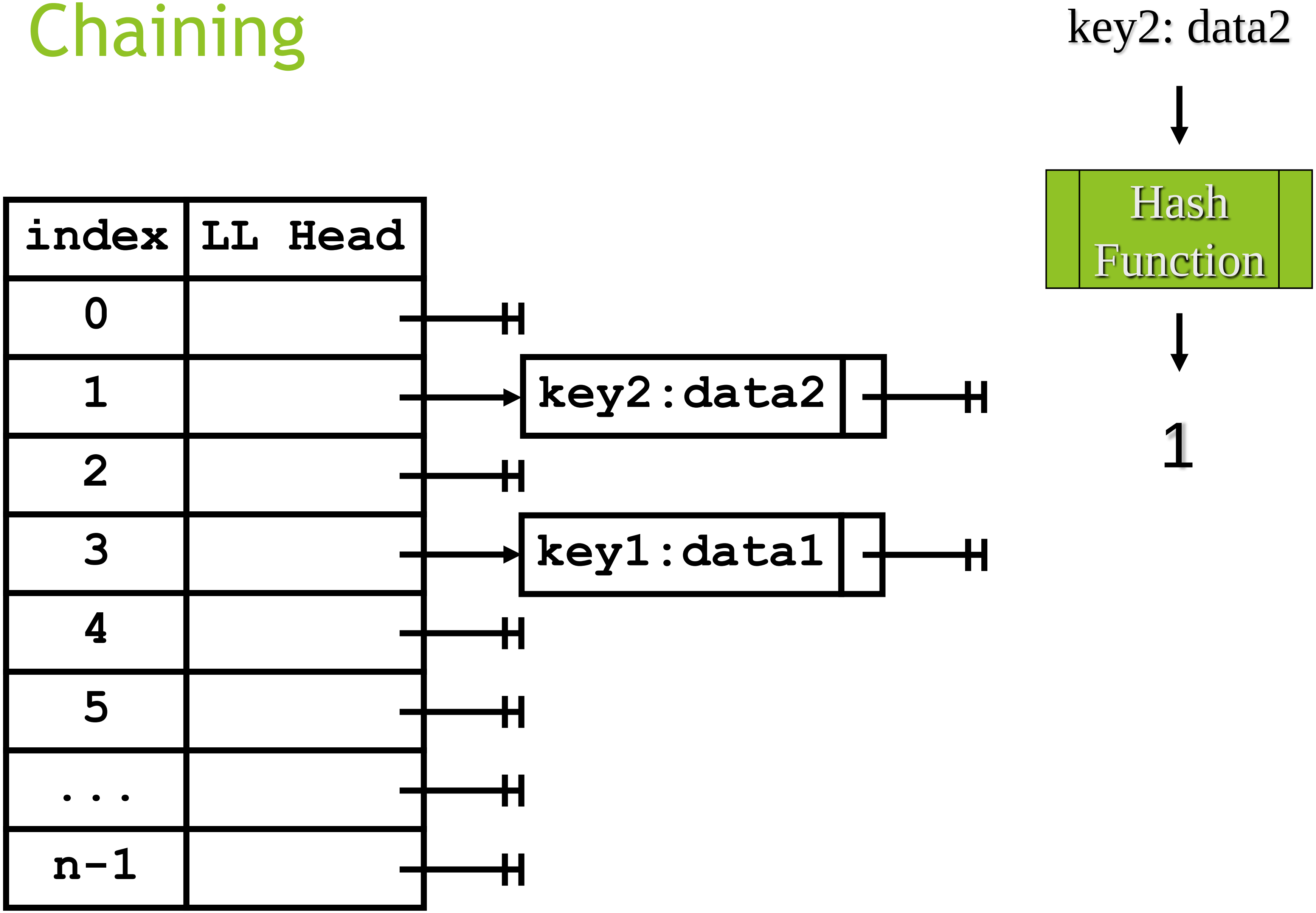
index	LL Head
0	
1	
2	
3	
4	
5	
...	
n-1	

Chaining


key1: data1
Hash Function
3

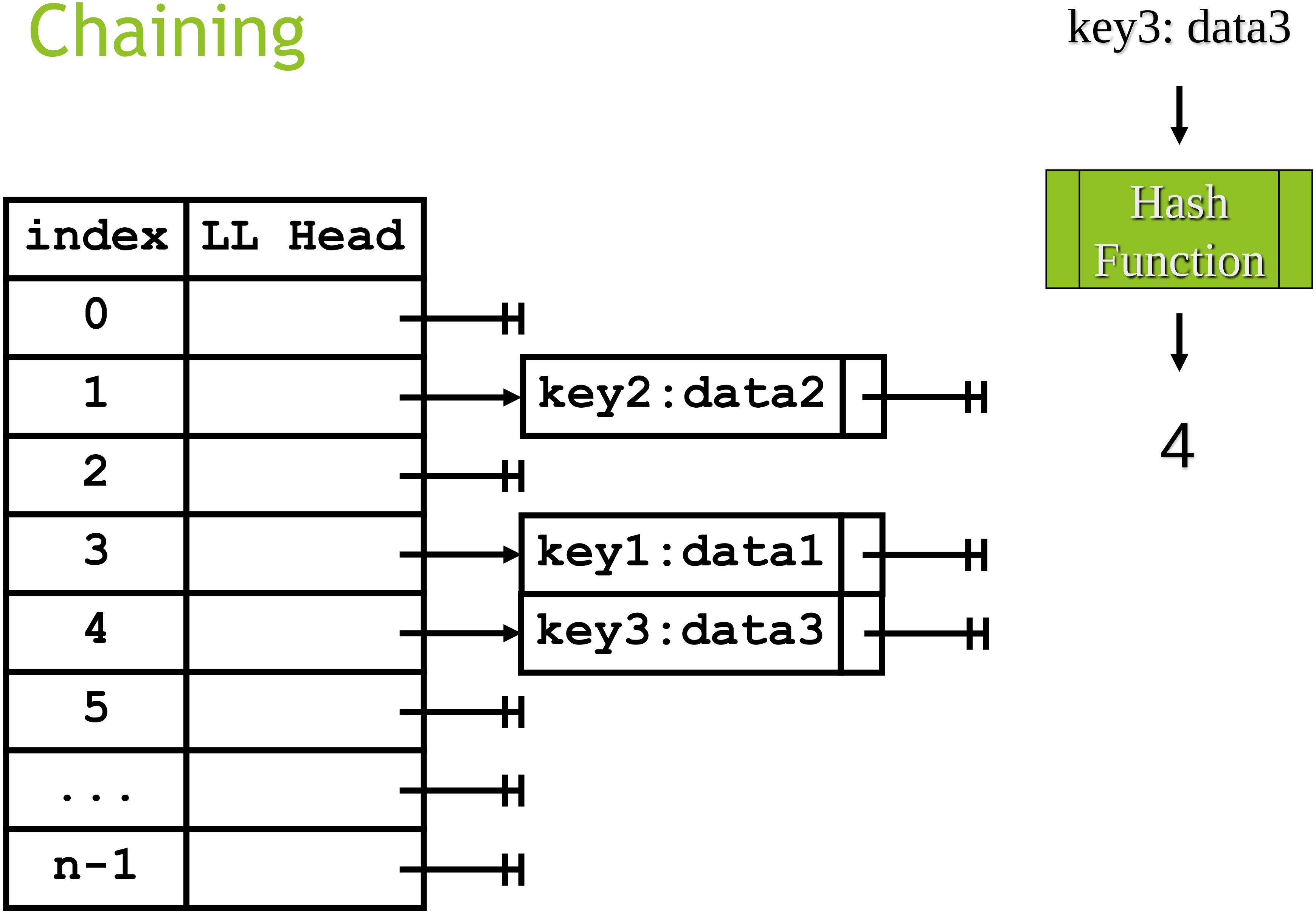

index
LL Head
0
1
2
3
4
5
...
n-1
key1:data1

Chaining



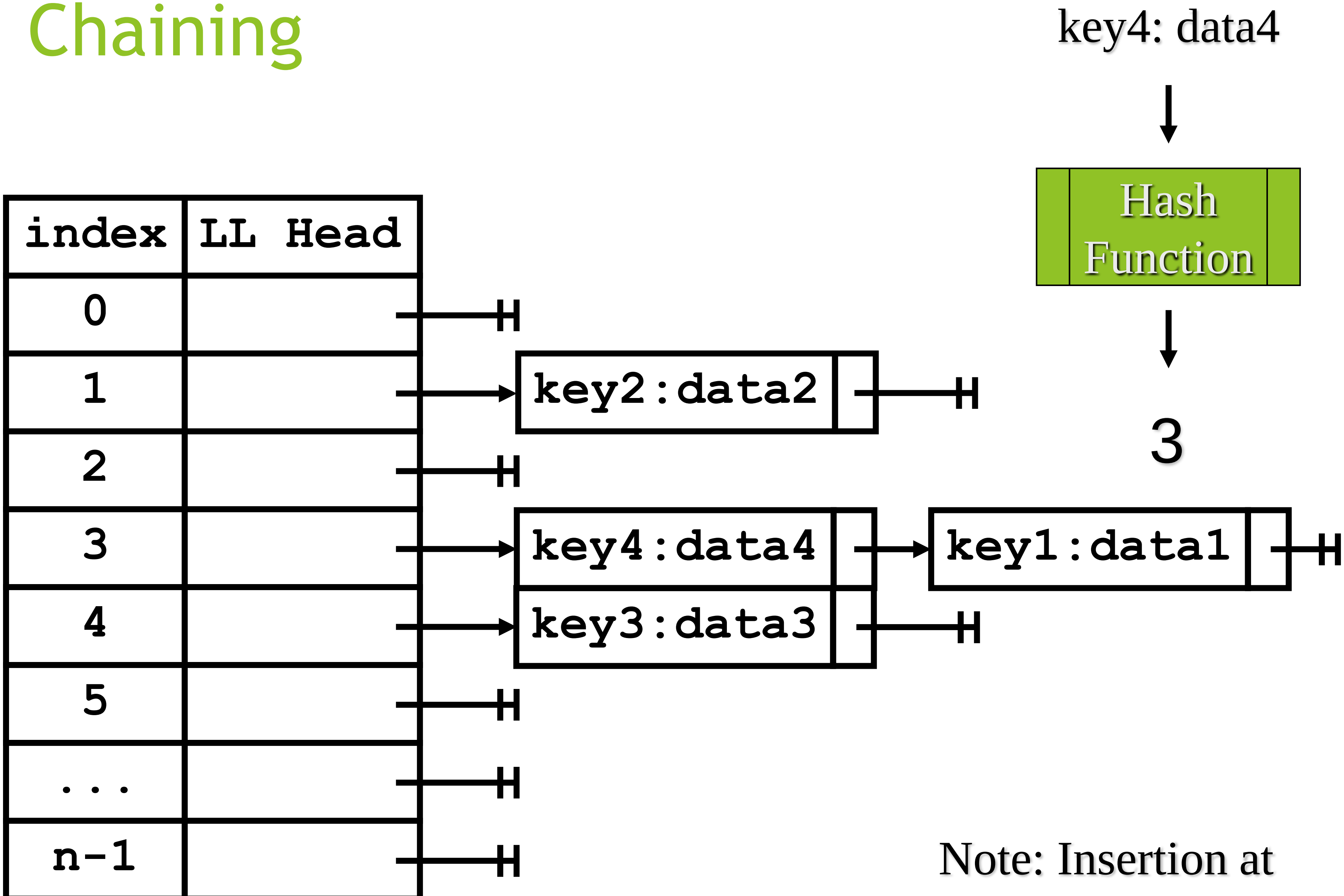
key2: data2
Hash
Function
1
index
LL Head
0
1
2
3
4
5
...
n-1
key2:data2
key1:data1

Chaining



key3: data3
Hash Function
4
index
LL Head
0
1
2
3
4
5
...
n-1
key2:data2
key1:data1
key3:data3

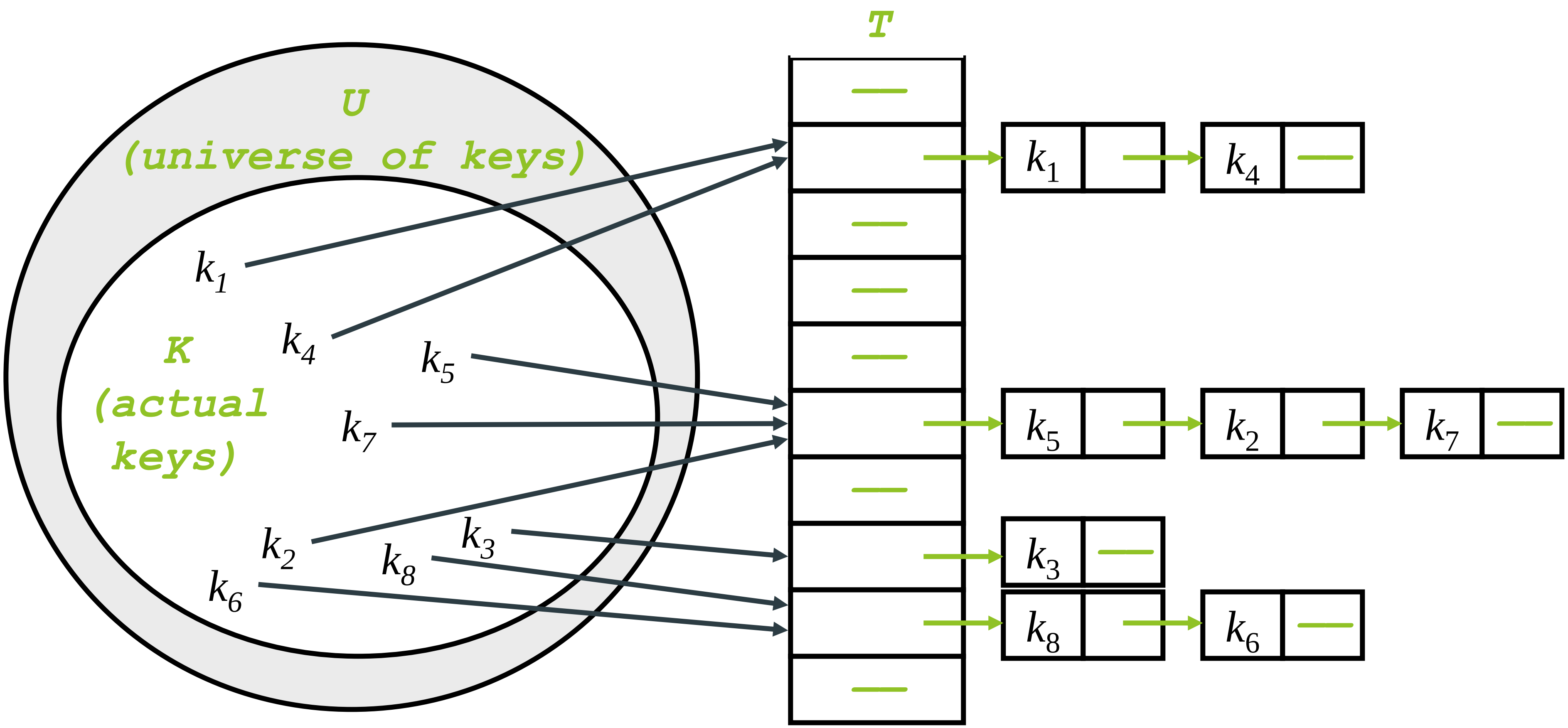
Chaining



Note: Insertion at beginning of list

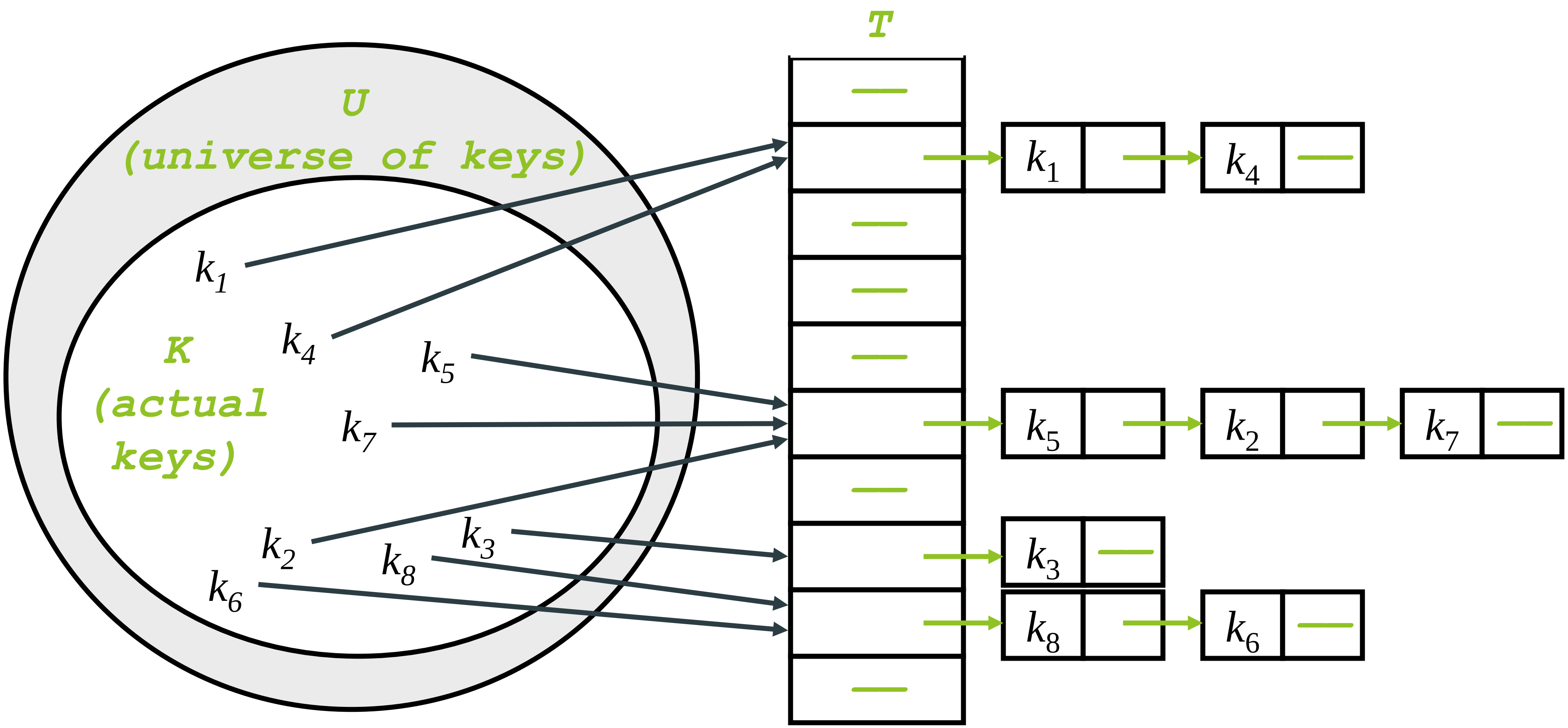
Chaining

- Chaining puts elements that hash to the same slot in a linked list:



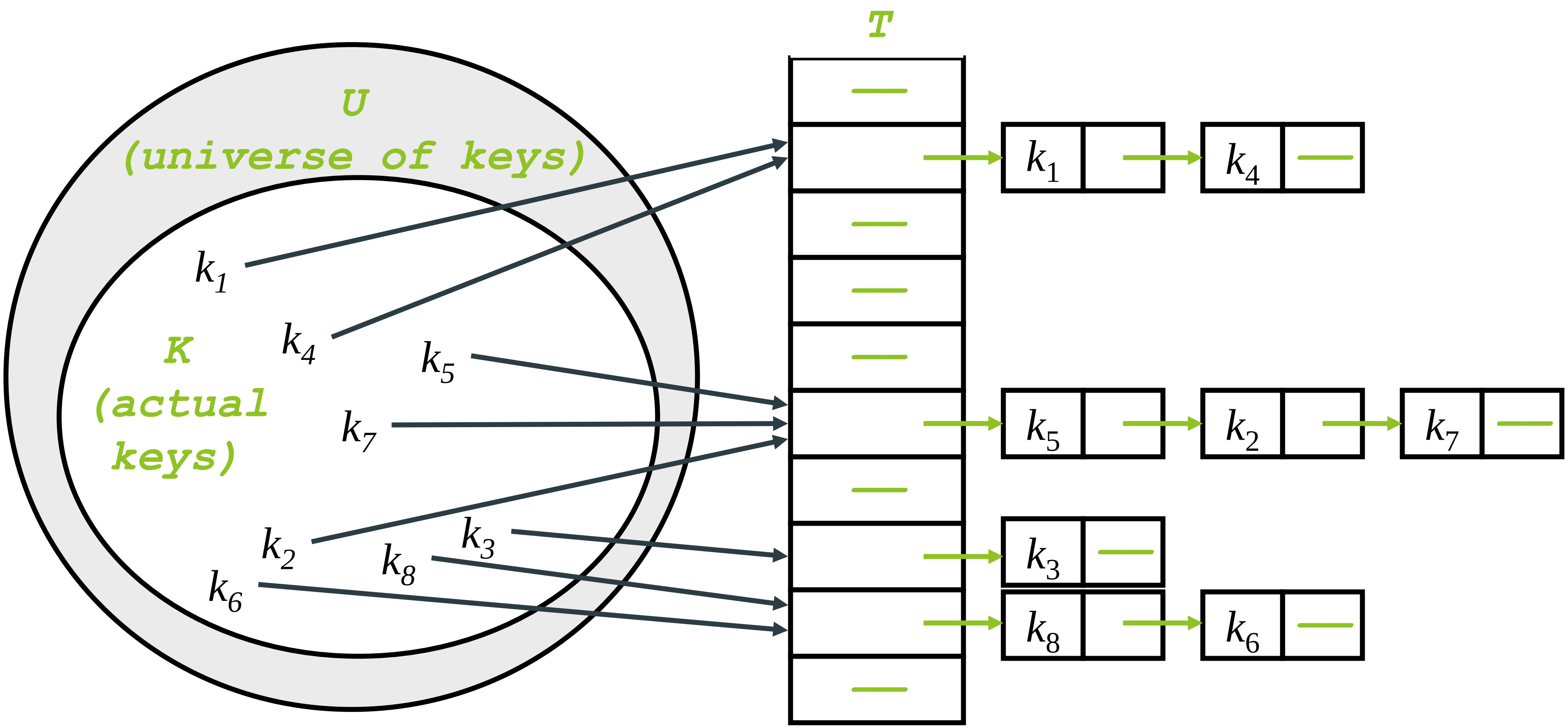
Chaining

- *How do we insert an element?*



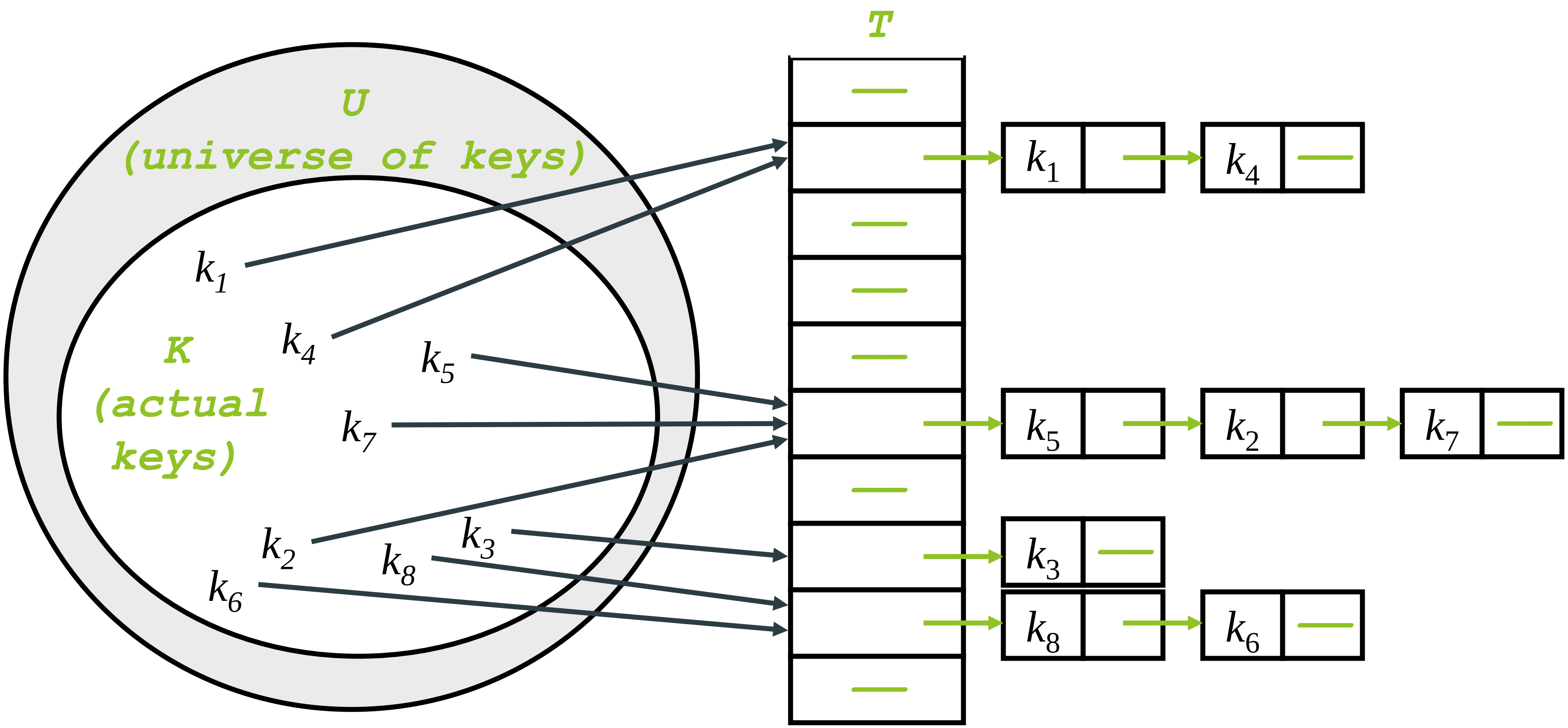
Chaining

- *How do we delete an element?*
  - *Do we need a doubly-linked list for efficient delete?*



Chaining

- *How do we search for an element with a given key?*



Chaining

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

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

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

Practice Problems

- Draw a hash table after we insert the keys 5; 28; 19; 15; 20; 33; 12; 17; 10 with collisions resolved by chaining. Let the table have 9 slots, T[0,..,8], and the hash function be h(k) = k mod 9.

i	0	1	2	3	4	5	6	7	8
T[i]	[]	[10, 19, 28]	[20]	[12]	[]	[5]	[33, 15]	[]	[17]

Analysis of Chaining

- Assume *simple uniform hashing*: each key in table is equally likely to be hashed to any slot
- Given *n* keys and *m* slots in the table, the *load factor* $\alpha = n\ /\ m$ = average # keys per slot
- *What will be the average cost of an unsuccessful search for a key?*
  - $O(1+\alpha)$
- *What will be the average cost of a successful search?*
  - $O(1 + \alpha/2) = O(1 + \alpha)$

Analysis of Chaining

- So, the cost of searching
  - O(1 + α)
- *If the number of keys n is proportional to the number of slots in the table, what is α?*
  - α = O(1)
  - In other words, we can make the expected cost of searching constant if we make α constant

What does this analysis mean? 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, $\alpha = n/m = O(m)/m = O(1)$.

Choosing A Hash Function

- Clearly choosing the hash function well is crucial

- *What are desirable features of the hash function?*
  - Should distribute keys uniformly into slots
  - Should not depend on patterns in the data

Interpreting keys as natural numbers

Most hash functions assume that the universe of keys is the set $\mathbb{N} = \{0, 1, 2, \ldots\}$ of natural numbers. Thus, if the keys are not natural numbers, we find a way to interpret them as natural numbers. For example, we can interpret a character string as an integer expressed in suitable radix notation. Thus, we might interpret the identifier `pt` as the pair of decimal integers $(112, 116)$, since $\texttt{p} = 112$ and $\texttt{t} = 116$ in the ASCII character set; then, expressed as a radix-128 integer, `pt` becomes $(112 \cdot 128) + 116 = 14452$. In the context of a given application, we can usually devise some such method for interpreting each key as a (possibly large) natural number. In what follows, we assume that the keys are natural numbers.

Hash Functions: The Division Method

- Idea
  - Map a key k into one of the m slots by taking the remainder of k divided by m

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

- Advantage
  - fast, requires only one operation
- Disadvantage
  - Certain values of m are bad: power of 2 and non-prime numbers

The Division Method

- A good choice for m: a prime number, not too close to an exact power of 2
- e.g., allocate a hash table, with collisions resolved through chaining
  - n = 2000-character strings (8 bits/character)
  - Choose m roughly n/3: m = 701 (prime near 2000/3, not near a power of 2)
  - h(k) = k mod 701

Hash Functions: The Multiplication Method

Idea:

- Multiply key k by a constant A, $0 < A < 1$
- Extract the fractional part of kA
- Multiply the fractional part by m
- Take the floor of the result

$$h(k) = \lfloor m \underbrace{(k\,A \bmod 1)}_{\text{fractional part of } kA \,=\, kA - \lfloor kA \rfloor} \rfloor$$

- **Disadvantage:** Slower than division method
- **Advantage:** Value of m is not critical: typically 2^p

The Multiplication Method

- For a constant A, $0 < A < 1$:
- $h(k) = \lfloor m\,\underbrace{(kA - \lfloor kA \rfloor)}_{\textit{Fractional part of } kA} \rfloor$
- Choose $m = 2^P$
- Choose A not too close to 0 or 1
- Knuth: Good choice for $A = (\sqrt{5} - 1)/2$

Hash Functions: Universal Hashing

- As before, when attempting to foil a malicious adversary: randomize the algorithm
- *Universal hashing*:
  - Guarantees good performance on average, no matter what keys adversary chooses
  - Suppose we want the hash function to uniformly distribute hash values over the hash table of size m
  - Given h(x), we want Prob{h(x)=h(y)}=1/m
  - The # of functions |f| in H s.t. h(x)=h(y) for any x, y in U
  - |f| / |H| = 1/m equivalent to |f| = |H| / m

Let $\mathcal{H}$ be a finite collection of hash functions that map a given universe U of keys into the range $\{0, 1, \ldots, m-1\}$. Such a collection is said to be ***universal*** if for each pair of distinct keys $k, l \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(k) = h(l)$ is at most $|\mathcal{H}|/m$. In other words, with a hash function randomly chosen from $\mathcal{H}$, the chance of a collision between distinct keys k and l is no more than the chance $1/m$ of a collision if $h(k)$ and $h(l)$ were randomly and independently chosen from the set $\{0, 1, \ldots, m-1\}$.

A Universal Hash Function

- Choose table size m to be prime
- Decompose key x into r+1 bytes, so that $x = \{x_0, x_1, \ldots, x_r\}$
  - Only requirement is that max value of byte < m
  - Let $a = \{a_0, a_1, \ldots, a_r\}$ denote a sequence of r+1 elements chosen randomly from $\{0, 1, \ldots, m - 1\}$
  - Define corresponding hash function h_a
    - $$h_a(x) = \sum_{i=0}^{r} a_i x_i \bmod m$$

e.g. Universal Hash Functions

We now define the hash function h_{ab} for any $a \in \mathbb{Z}_p^*$ and any $b \in \mathbb{Z}_p$ using a linear transformation followed by reductions modulo p and then modulo m:

$$h_{ab}(k) = ((ak + b) \bmod p) \bmod m . \tag{11.3}$$

For example, with $p = 17$ and $m = 6$, we have $h_{3,4}(8) = 5$. The family of all such hash functions is

$$\mathcal{H}_{pm} = \left\{h_{ab} : a \in \mathbb{Z}_p^* \text{ and } b \in \mathbb{Z}_p\right\} . \tag{11.4}$$

Each hash function h_{ab} maps $\mathbb{Z}_p$ to $\mathbb{Z}_m$. This class of hash functions has the nice property that the size m of the output range is arbitrary—not necessarily prime—a feature which we shall use in Section 11.5. Since we have $p - 1$ choices for a and p choices for b, the collection $\mathcal{H}_{pm}$ contains $p(p - 1)$ hash functions.

e.g. Universal Hash Functions

$p = 17,\ m = 6$

$h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$

$h_{3,4}(8) = ((3 \cdot 8 + 4) \bmod 17) \bmod 6$

$= (28 \bmod 17) \bmod 6$

$= 11 \bmod 6$

$= 5$

Open Addressing

- If we have enough contiguous memory to store all the keys ($m > N$)

  $\Rightarrow$ store the keys in the table itself
- No need to use the linked lists anymore
- Collision resolution
  - Put the elements that collide in the available empty places in the table

Open Addressing

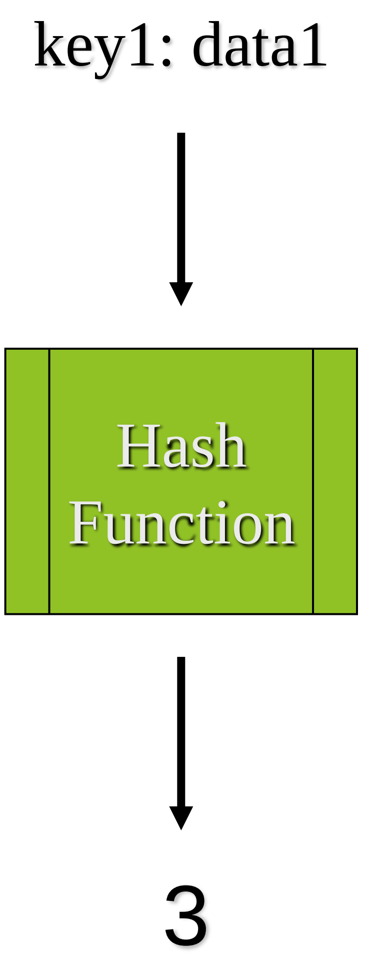
- Basic idea:
  - To insert: if slot is full, try another slot, ..., until an open slot is found (*probing*)
  - To search, follow same sequence of probes as would be used when inserting the element
    - If reach element with correct key, return it
    - If reach a NULL pointer, element is not in table
- Good for fixed sets (adding but no deletion)
  - Example: spell checking
- Table needn't be much bigger than n

Open Addressing

index	data
0	
1	
2	
3	
4	
5	
...	
n-1	

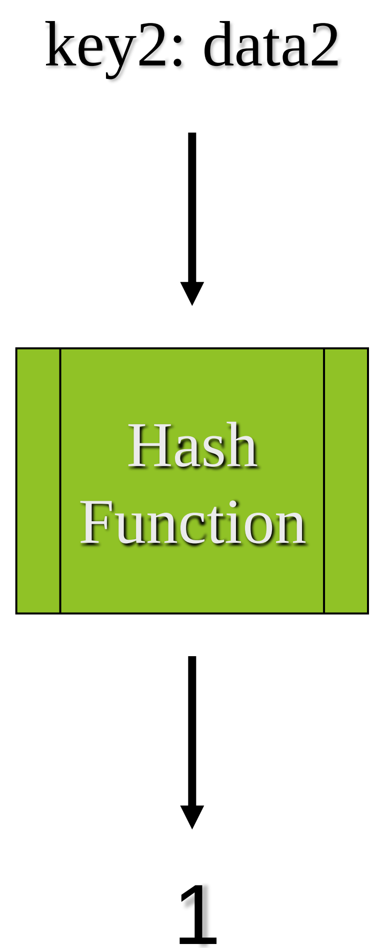
Open Addressing

index	data
0	
1	
2	
3	key1:data1
4	
5	
...	
n-1	



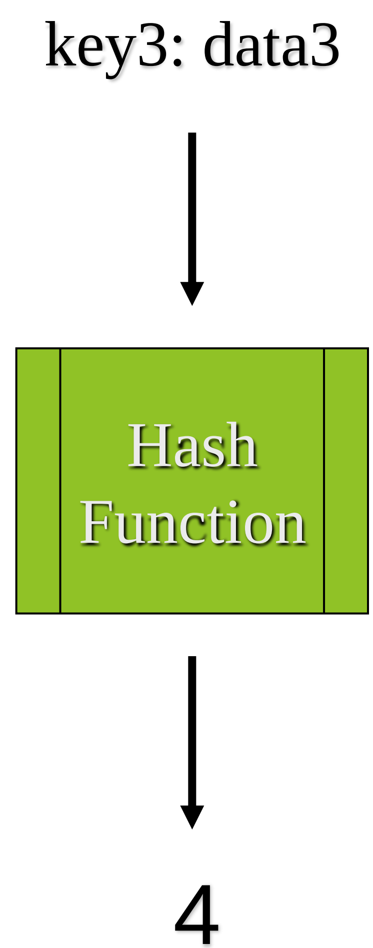
Open Addressing

index	data
0	
1	key2:data2
2	
3	key1:data1
4	
5	
...	
n-1	



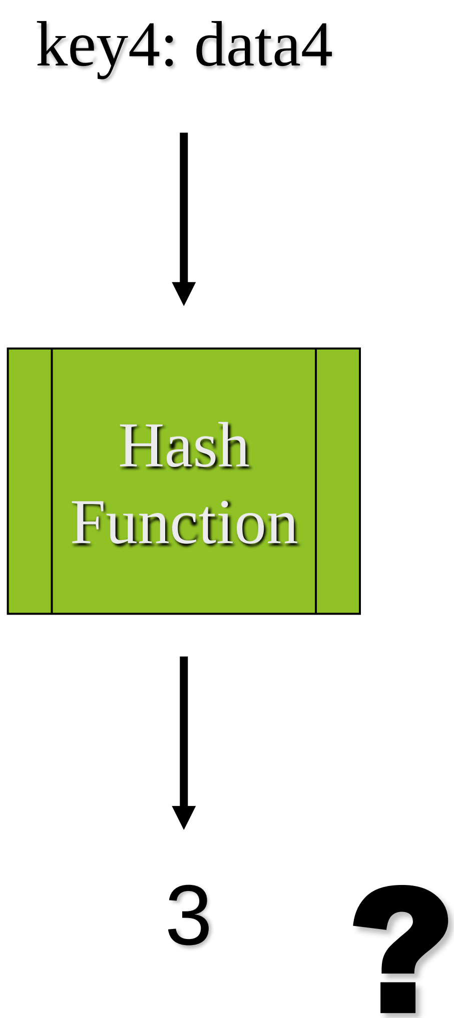
Open Addressing

index	data
0	
1	key2:data2
2	
3	key1:data1
4	key3:data3
5	
...	
n-1	



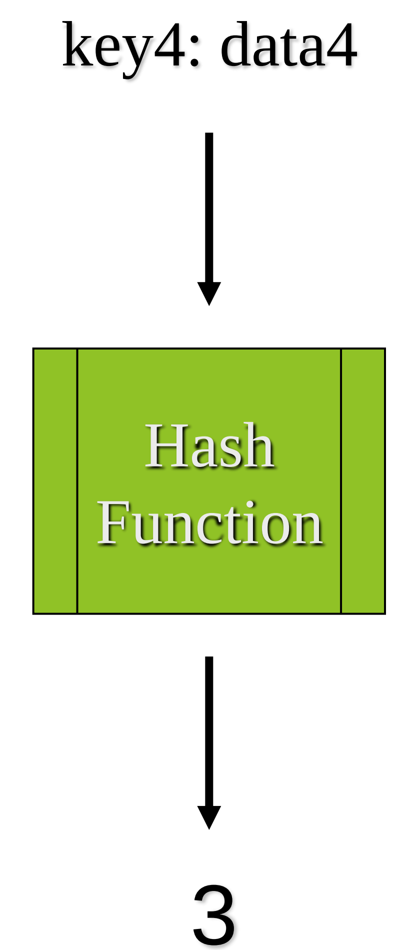
Open Addressing

index	data
0	
1	key2:data2
2	
3	key1:data1
4	key3:data3
5	
...	
n-1	

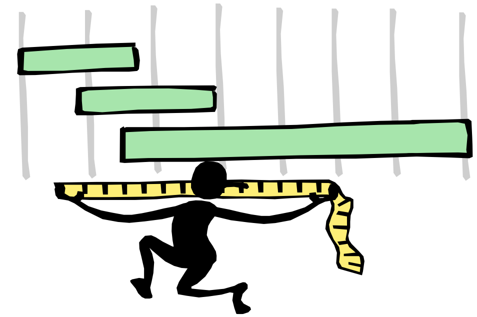


Open Addressing

index	data
0	
1	key2:data2
2	
3	key1:data1
4	key3:data3
5	key4:data4
...	
n-1	



Open Addressing



```
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
```

```
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"
```

Linear Probing

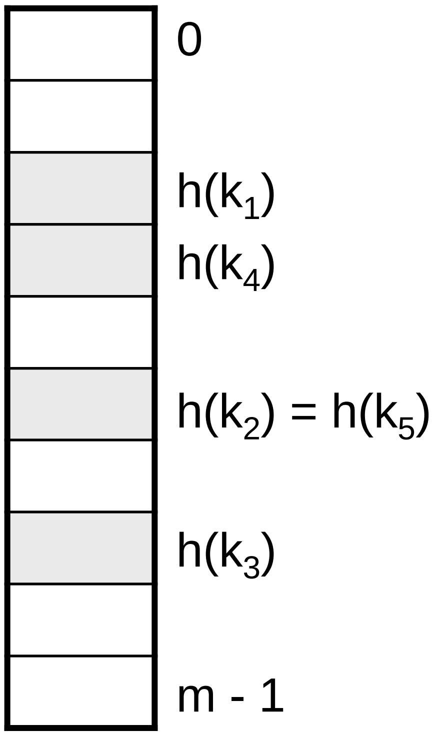
- When there is a collision, check the next available position in the table (probing)

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

- First slot probed: $h_1(k)$, second: $h_1(k) + 1$ and so on

Linear Probing

- Searching for a key
- Three situations:
  - Position in table is occupied with an element of equal key
  - Position in table is empty
  - Position in table occupied with a different element
- Probe the next higher index until the element is found or an empty position is found
- The process wraps around to the beginning of the table



Linear Probing

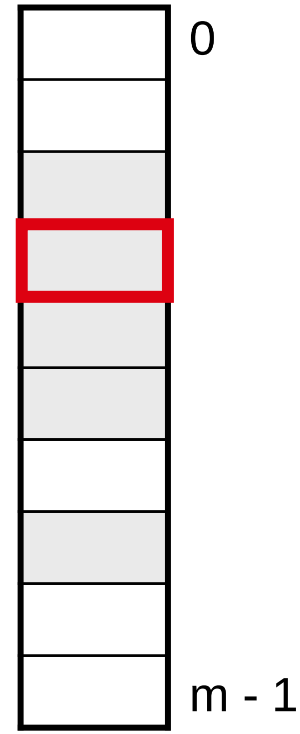
Assume `hash(x) = hash(y) = hash(z) = i`. And assume `x` was inserted first, then `y` and then `z`.
In open addressing: `table[i] = x`, `table[i+1] = y`, `table[i+2] = z`.

Now, assume you want to delete `x`, and set it back to `NULL`.

When later you will search for `z`, you will find that `hash(z) = i` and `table[i] = NULL`, and you will return a wrong answer: `z` is not in the table.

To overcome this, you need to set `table[i]` with a special marker indicating to the search function to keep looking at index `i+1`, because there might be element there which its hash is also `i`.

- Deleting a key
  - Cannot mark the slot as empty
  - Impossible to retrieve keys inserted after that slot was occupied
- Solution
  - Mark the slot with a sentinel value DELETED
- The deleted slot can later be used for insertion
- Searching will be able to find all the keys



Practice Problems

- Consider a hash table of length 11 using open addressing with the primary hash function $h_1(k) = k$. Illustrate the result of inserting 31, 4, 15, 28, 59 using linear probing.
  - Hash function $h(k,i) = (h_1(k) + i) \bmod m = (k+i) \bmod m$
    - h(31,0) = 9 → T[9] = 31
    - h(4,0) = 4 → T[4] = 4
    - h(15,0) = 4 collision
      - h(15,1) = 5 → T[5] = 15
    - h(28,0) = 6 → T[6]= 28
    - h(59,0) = 4 collision
      - h(59,1) = 5 collision
      - h(59,2) = 6 collision
      - h(59,3) = 7 →T[7] = 59

Double Hashing

- Use a first hash function to determine the first slot
- Use a second hash function to determine the increment for the probe sequence

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

- Initial probe: $h_1(k)$, second probe is offset by $h_2(k)$ mod m, so on
- Advantage: avoids clustering
- Disadvantage: harder to delete an element

Double Hashing

$h_1(k) = k \bmod 13$

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

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

- Insert key 14:

$h_1(14) = 14 \bmod 13 = 1$

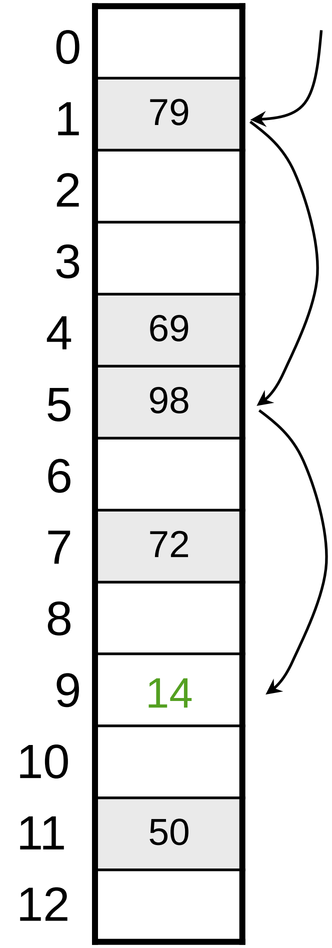
$h_2(14) = 1 + (14 \bmod 11) = 4$

$h(14, 1) = (h_1(14) + h_2(14)) \bmod 13$

$= (1 + 4) \bmod 13 = 5$

$h(14, 2) = (h_1(14) + 2\, h_2(14)) \bmod 13$

$= (1 + 8) \bmod 13 = 9$



Double Hashing

Choosing the second hash function

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

- h_2 should not evaluate to 0 $\Rightarrow$ infinite loop
- h_2 should be relatively prime to the table size m
  - $m = 2\, h_2$: only few slots will be visited
- Solution 1
  - Choose m prime
  - Design h_2 such that it produces an integer less than m
- Solution 2
  - Choose $m = 2^p$
  - Design h_2 such that it always produces an odd number

Practice Problems

- Consider a hash table of length 11 using open addressing. Illustrate the result of inserting 31, 4, 15, 28, 59 using double hashing with functions $h_1(k) = k$ and $h_2(k) = 1 + (k \bmod (m-1))$.
  - Hash function *h(k,i) = (k + i(1 + (k mod (m-1)))) mod m*
    - h(31,0) = 9 → T[9] = 31
    - h(4,0) = 4 → T[4] = 4
    - h(15,0) = 4 collision
      - h(15,1) = 10 → T[10] = 15
    - h(28,0) = 6 → T[6]= 28
    - h(59,0) = 4 collision
      - h(59,1) = 3 → T[3] = 59

Thanks to contributors

Mr. Pham Van Nguyen (2022)

Dr. Thien-Binh Dang (2017 – 2022)

Prof. Hyunseung Choo (2001 – 2022)