Insertion in Red-Back Trees + Hash Tables

Instructor: Krishna Venkatasubramanian

CSC 212

Based on slides from Prof. David Luebke at VirginiaTech

Red-Black Trees (Recap)

- What is a Red-Black Tree?
- What are main properties of Red-Black Trees?
- How tall is a Red-Back Tree?
- What is rotation in a Red-Black Tree?

Red-Black Trees: Insertion

- **Case 0:** New node is (z) is always RED .Insert as you would in BST
- **Case 1:** If the new node (z) is RED, and its parent (z.p) is BLACK you don't need to do anything.
- **Case 2:** If a new node (z) is RED and its parent (z.p) is RED, then:
  - **2.a**: if the uncle (y) is BLACK, a **rotation** needs to be performed
    - **2.a.i.** If the insertion path from grand-parent -> parent -> node **BOTH LEFT** then
      - Do RIGHT rotation **around grandparent (z.p.p)**
      - Color flip parent (z.p), grandparent (z.p.p)
    - **2.a.ii.** If the insertion path from grand-parent -> parent -> node **BOTH RIGHT** then
      - Do LEFT rotation **around grandparent** (z.p.p)
      - Color flip parent (z.p), grandparent (z.p.p)
    - **2.a.iii.** If the insertion path from grand-parent -> parent -> node is **LEFT then RIGHT** do:
      - Do LEFT rotation **around parent** (z.p)
      - Do RIGHT rotation around (z)
      - Color flip parent (z.p), grandparent (z.p.p)
    - **2.a.iV.** If the insertion path from grand-parent -> parent -> node is **RIGHT then LEFT** do:
      - Do RIGHT rotation **around parent** (z.p)
      - Do LEFT rotationaround (z)
      - Color flip parent (z.p), grandparent (z.p.p)
  - **2.b:** If the Uncle (y) is RED, a flip parent (z.p), uncle (y) and grandparent (z.p.p) color
- **Case 3:** If the Root is RED, change it to BLACK.

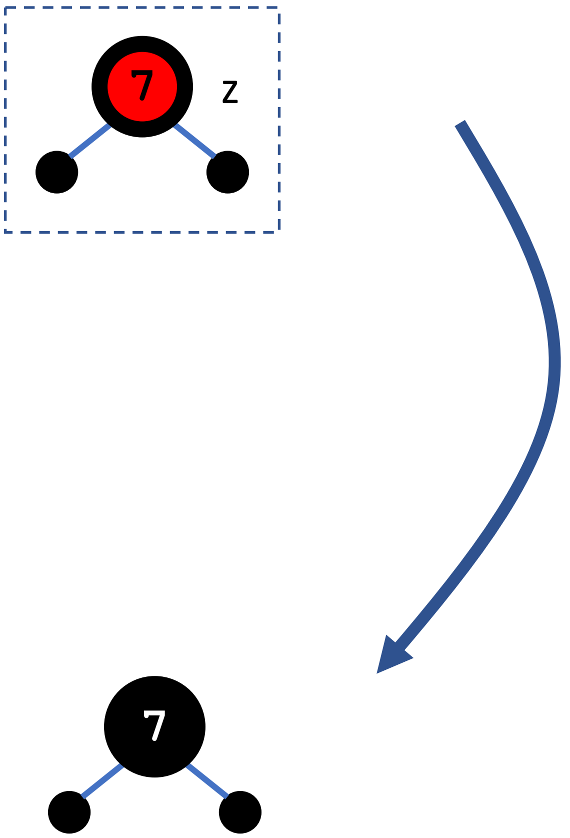
FROM: https://cathyatseneca.gitbook.io/data-strutures-and-algorithms/red-black-trees

Red-Black Trees: Insertion Example

- Insert 7

- **Case 0:** New node is (z) is always RED .Insert as you would in BST

- **Case 3:** If the Root is RED, change it to BLACK.

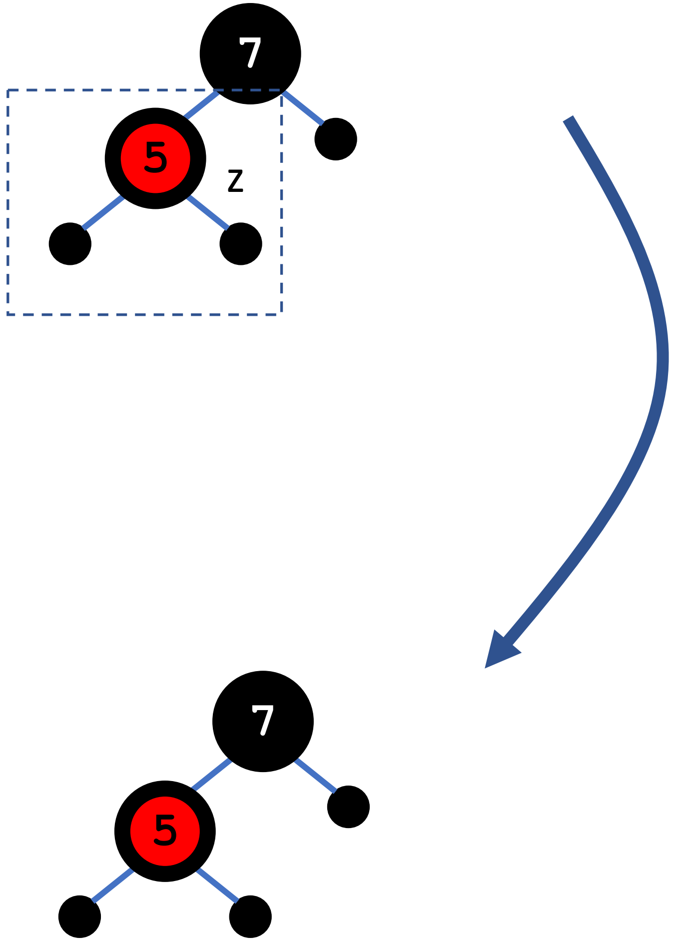


Red-Black Trees: Insertion Example

- Insert 5

- **Case 0:** New node is (z) is always RED .Insert as you would in BST

- **Case 1:** If the new node (z) is RED, and its parent (z.p) is BLACK you don't need to do anything.

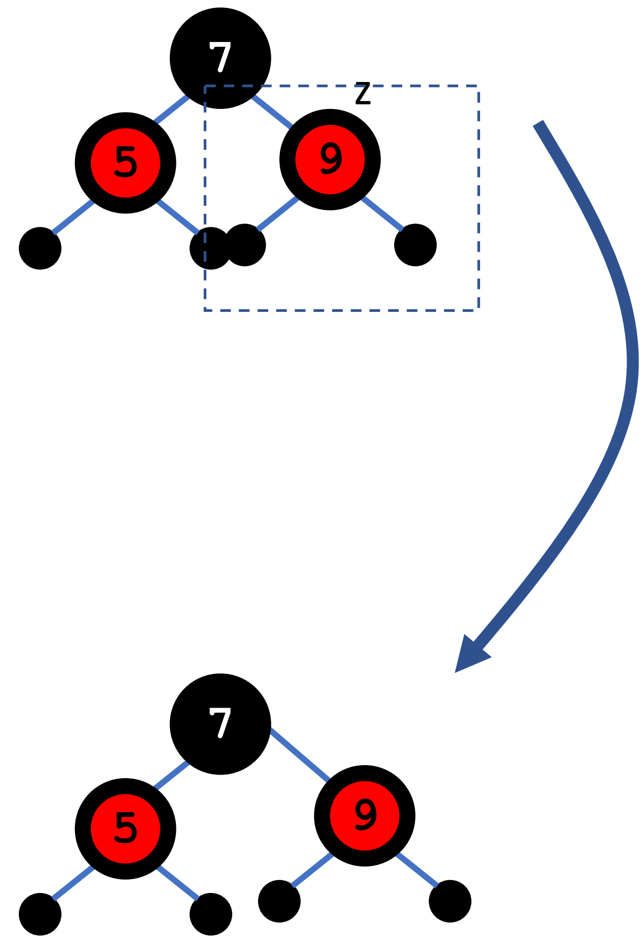


Red-Black Trees: Insertion Example

- Insert 9

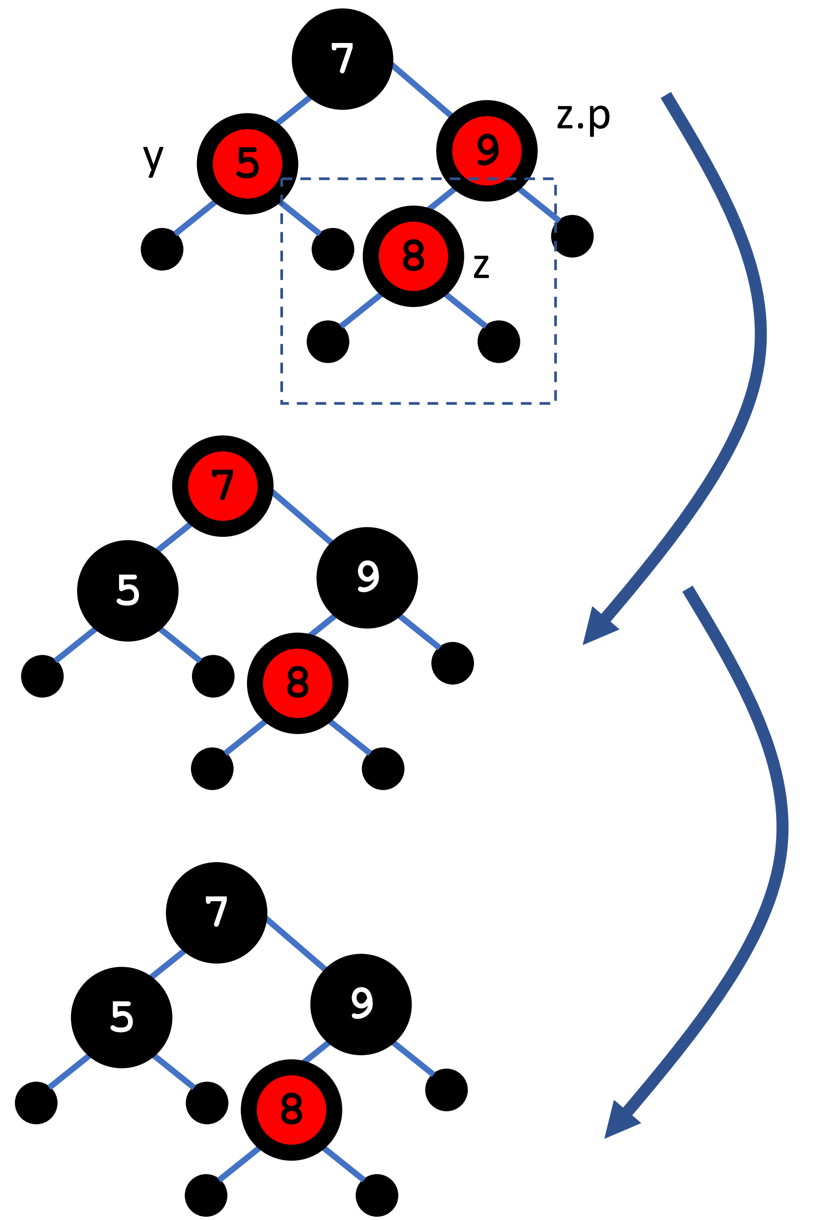
- **Case 0:** New node is (z) is always RED .Insert as you would in BST

- **Case 1:** If the new node (z) is RED, and its parent (z.p) is BLACK you don't need to do anything.



Red-Black Trees: Insertion Example

- Insert 8
- **Case 0:** New node is (z) is always RED .Insert as you would in BST
- **Case 2:** If a new node (z) is RED and its parent (z.p) is RED, then:
  - **2.b:** If the Uncle (y) is RED, a flip parent (z.p), uncle (y) and grandparent (z.p.p) color
- **Case 3:** If the Root is RED, change it to BLACK.

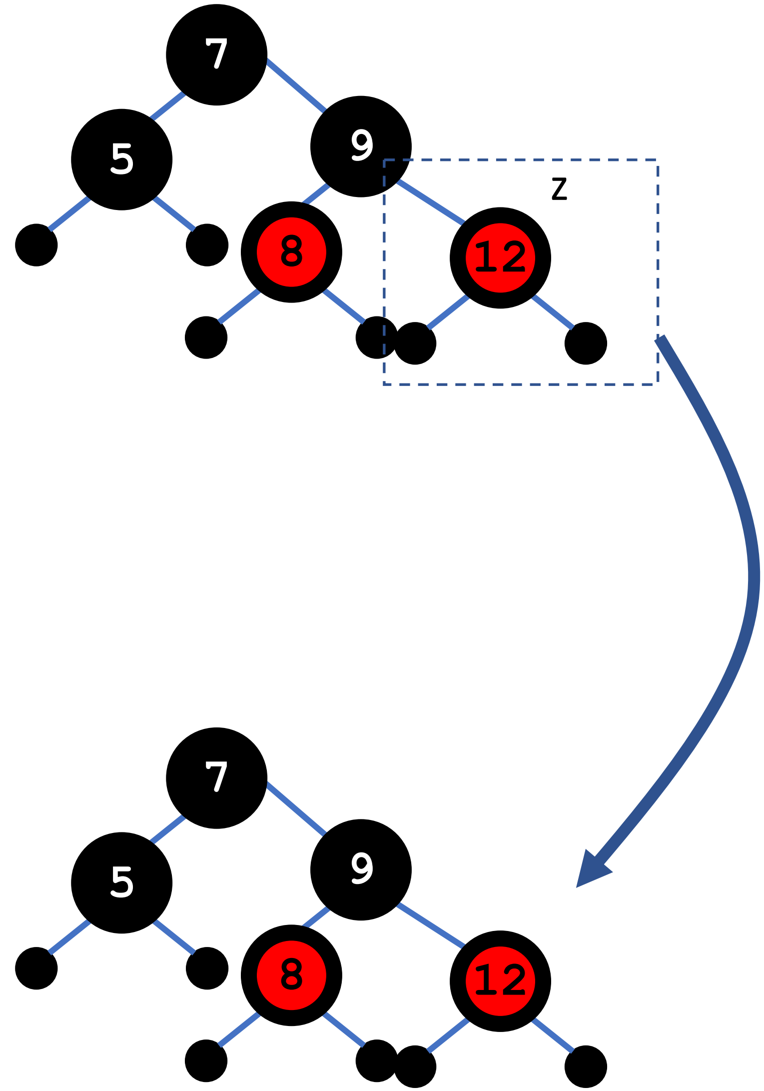


Red-Black Trees: Insertion Example

- Insert 12

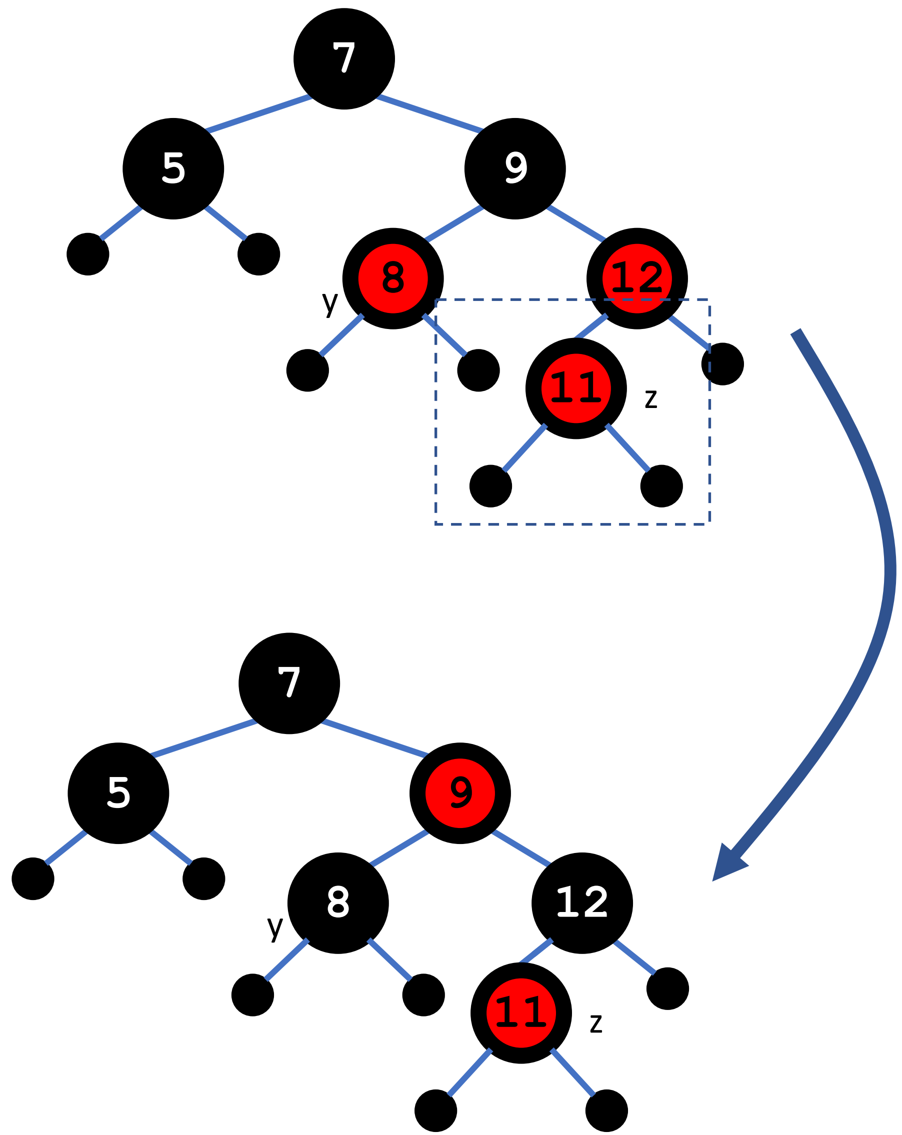
- **Case 0:** New node is (z) is always RED .Insert as you would in BST

- **Case 1:** If the new node (z) is RED, and its parent (z.p) is BLACK you don't need to do anything.



Red-Black Trees: Insertion Example

- Insert 11
- **Case 0:** New node is (z) is always RED .Insert as you would in BST
- **Case 2:** If a new node (z) is RED and its parent (z.p) is RED, then:
  - **2.b:** If the Uncle (y) is RED, a flip parent (z.p), uncle (y) and grandparent (z.p.p) color

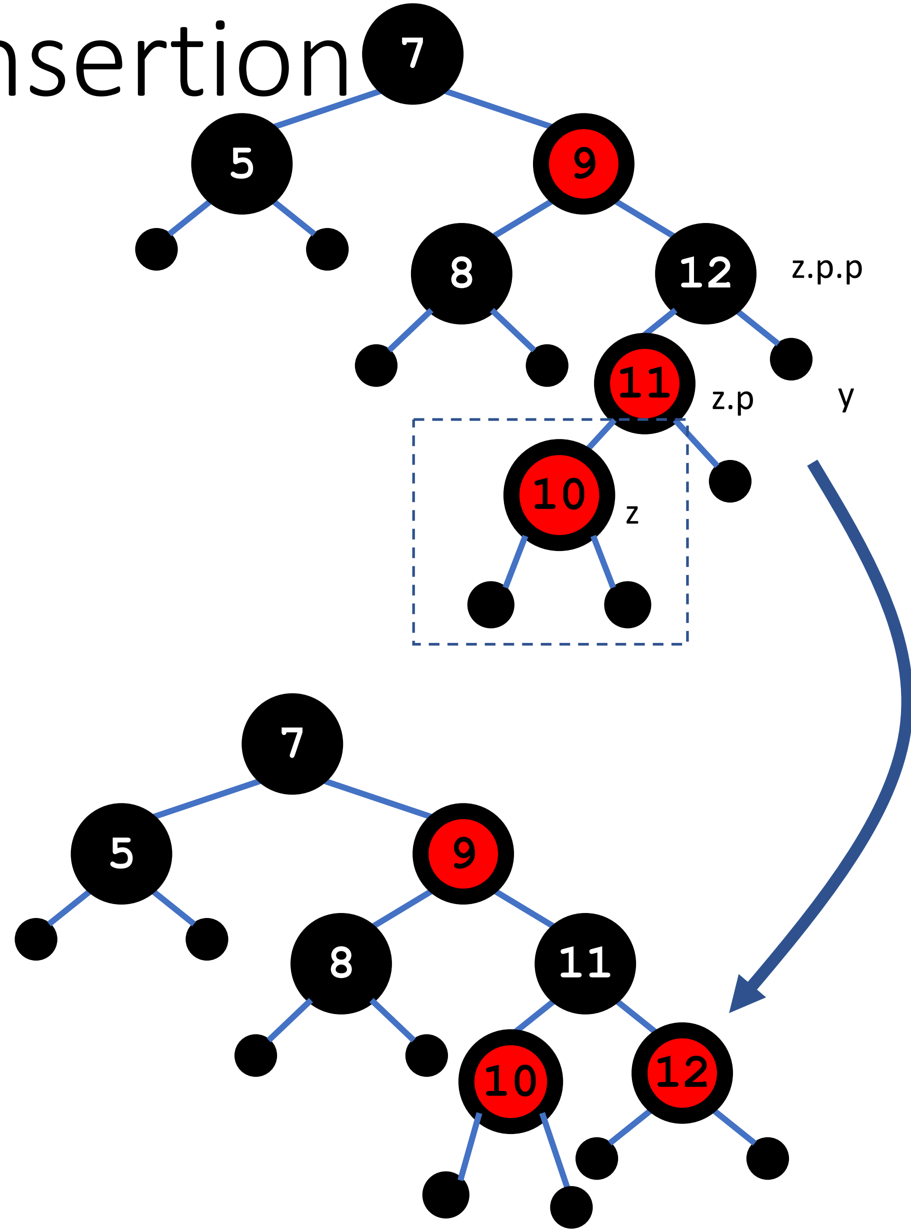


Red-Black Trees: Insertion Example

- Insert 10

- **Case 0:** New node is (z) is always RED .Insert as you would in BST

- **Case 2:** If a new node (z) is RED and its parent (z.p) is RED, then:
  - **2.a**: if the uncle (y) is BLACK, a **rotation** needs to be performed
    - **2.a.i.** If the insertion path from grand-parent -> parent -> node **BOTH LEFT** then
      - Do RIGHT rotation **around grandparent** (z.p.p)
      - Color flip parent (z.p), grandparent (z.p.p)



Red-Black Trees: Deletion

- And you thought insertion was tricky…
- We will not cover RB delete in class
  - If you want you can read section 13.4 of CR book on your own
  - I would recommend read for the overall picture, not the details

Hash Tables

Motivation

- In many applications we might want to store large amounts of data and quickly search them
- Example:
  - URI Student Database
  - Social Security office database for the U.S.
- One way of storing these is in a list
  - E.g., [Student 1, Student 2,……Student n]
- **However search this list will require O(n) complexity**
  - As we have to go through the entire list in the worst case.
- Can we do better?

Hash Tables

- Hash table:
  - Given a table *T* and a record *x*, with **key** (= symbol) and **values** 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

- For example in the URI student database case:
  - x is a student's record, which contains the student information, which contains:
    - **Keys** is the id number
      - In the following discussions we will consider all keys to be (possibly large) natural numbers
      - We use the **notation "x.key" to mean <u>key</u> for record x**
    - **Values:** which is the actual details of student information (name, address, …etc.)

Record X in Table

Key: {3}
Value: {name, Address, Major ….}

Direct Addressing

- 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.key = *i*
    - T[*i*] = NULL otherwise

  - This is called a ***direct-address table***
    - Operations take O(1) time!

The Problem With Direct Addressing

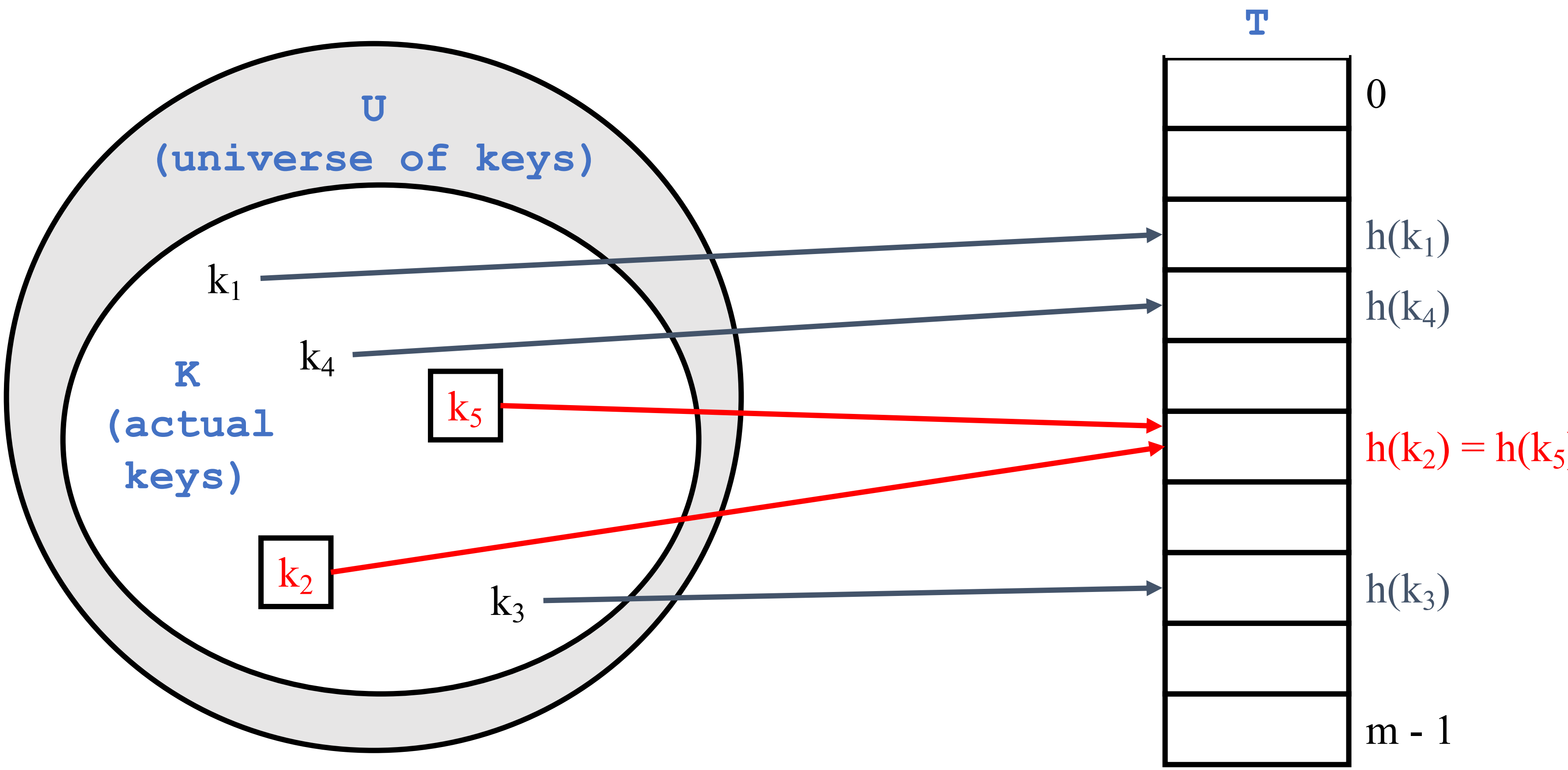
- Direct addressing works well when the range *m* of keys is relatively small

- But what if the keys are 32-bit integers?
  - **Problem 1:** direct-address table will have how many entries?
    - 2^32 = 4 billion

  - **Problem 2:** even if memory is not an issue, the time to initialize the elements to NULL may be an issue

- Solution?:
  - **map keys to smaller range 0..*m*-1**
  - This mapping is called a ***hash function***

Next problem: *collision*



Resolving Collisions

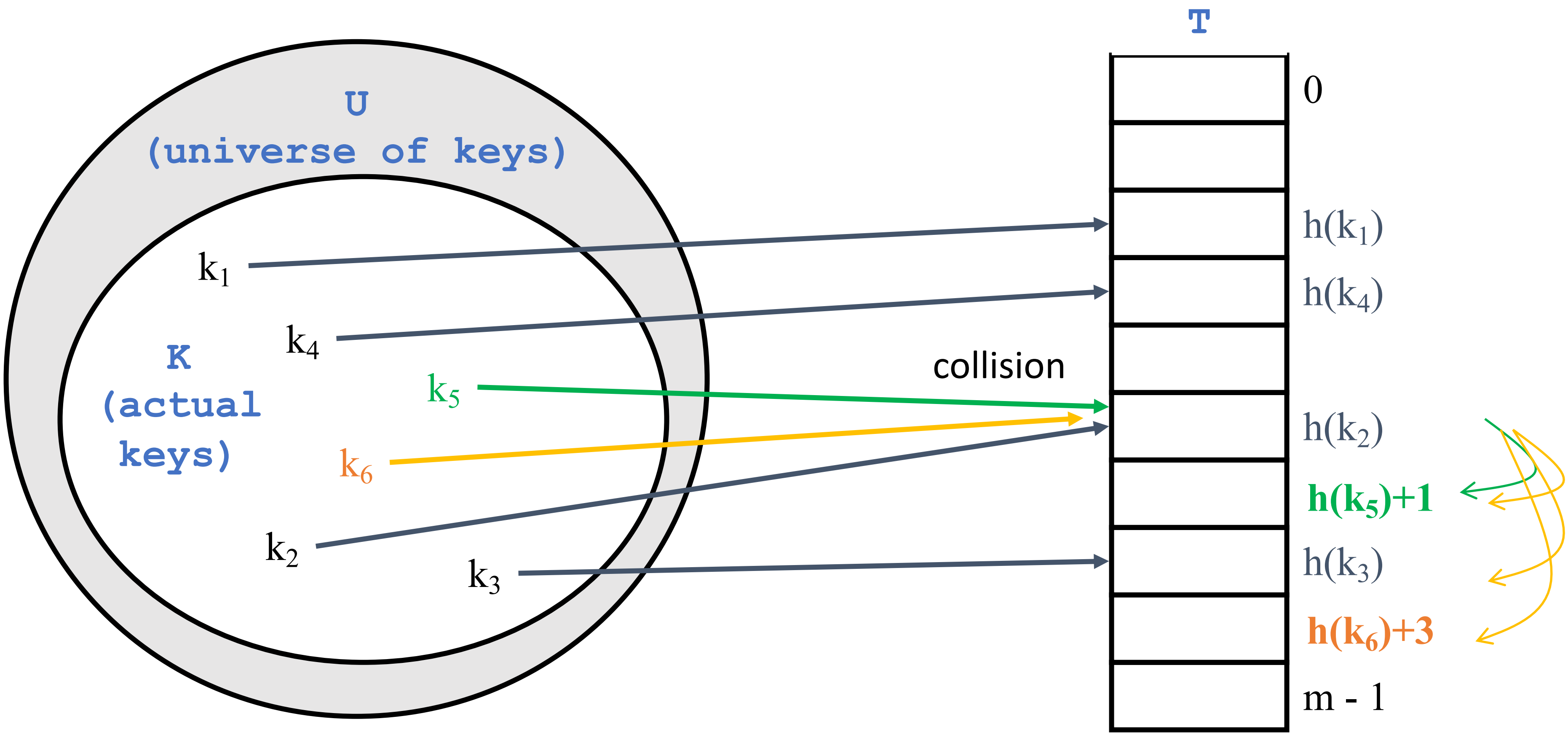
- *How can we solve the problem of collisions?*
- Solution 1: *open addressing + probing*
- Solution 2: *chaining*

Open Addressing + Probing

- 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

Linear Probing

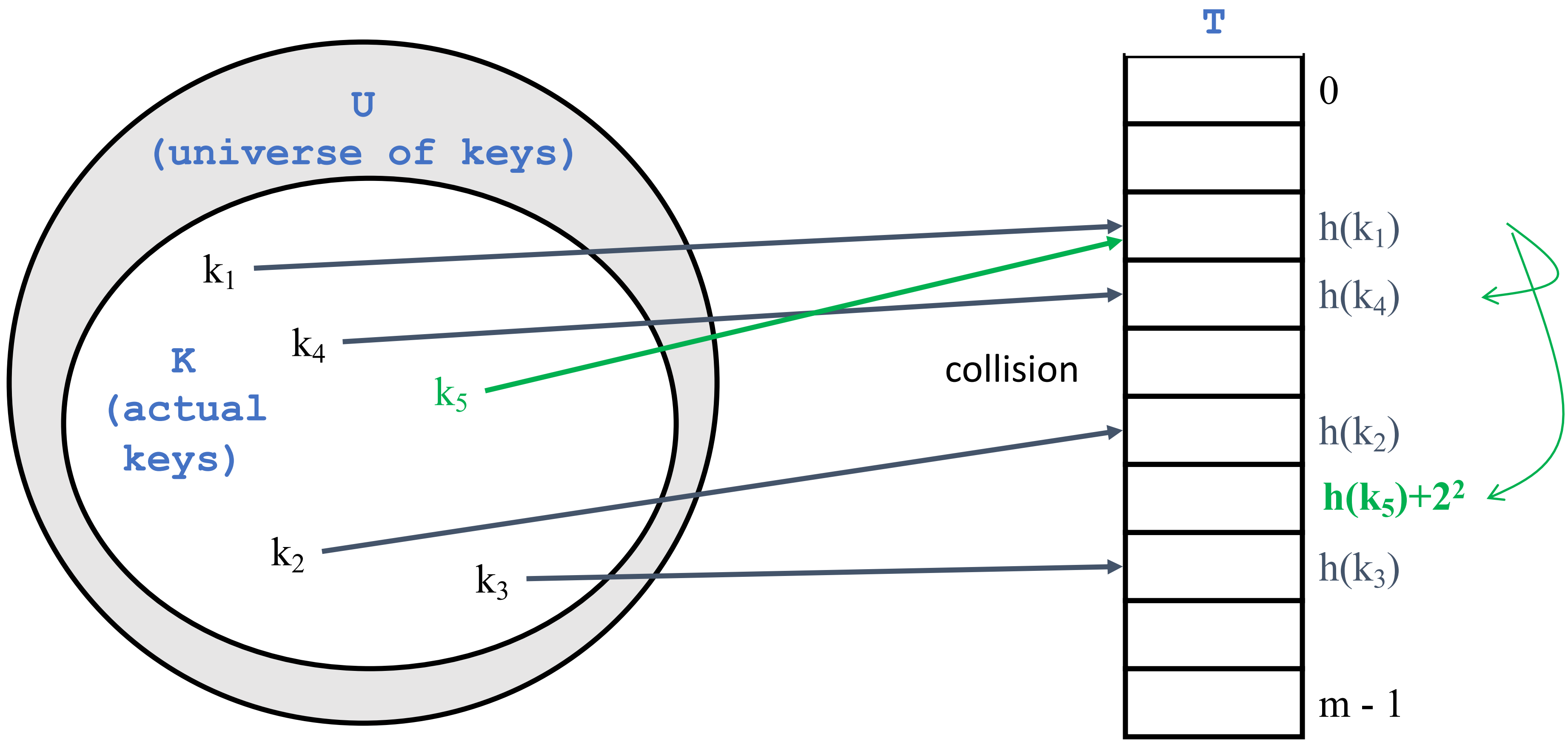
k5 takes 1 probe to find it's slot
K6 takes 3 probes to find it's slot



- If current slot taken, **probe** the next element and continue till you find an empty slot and then take it
- **h(k,i) = h(k)+i mod m** (here, i is the probe count)

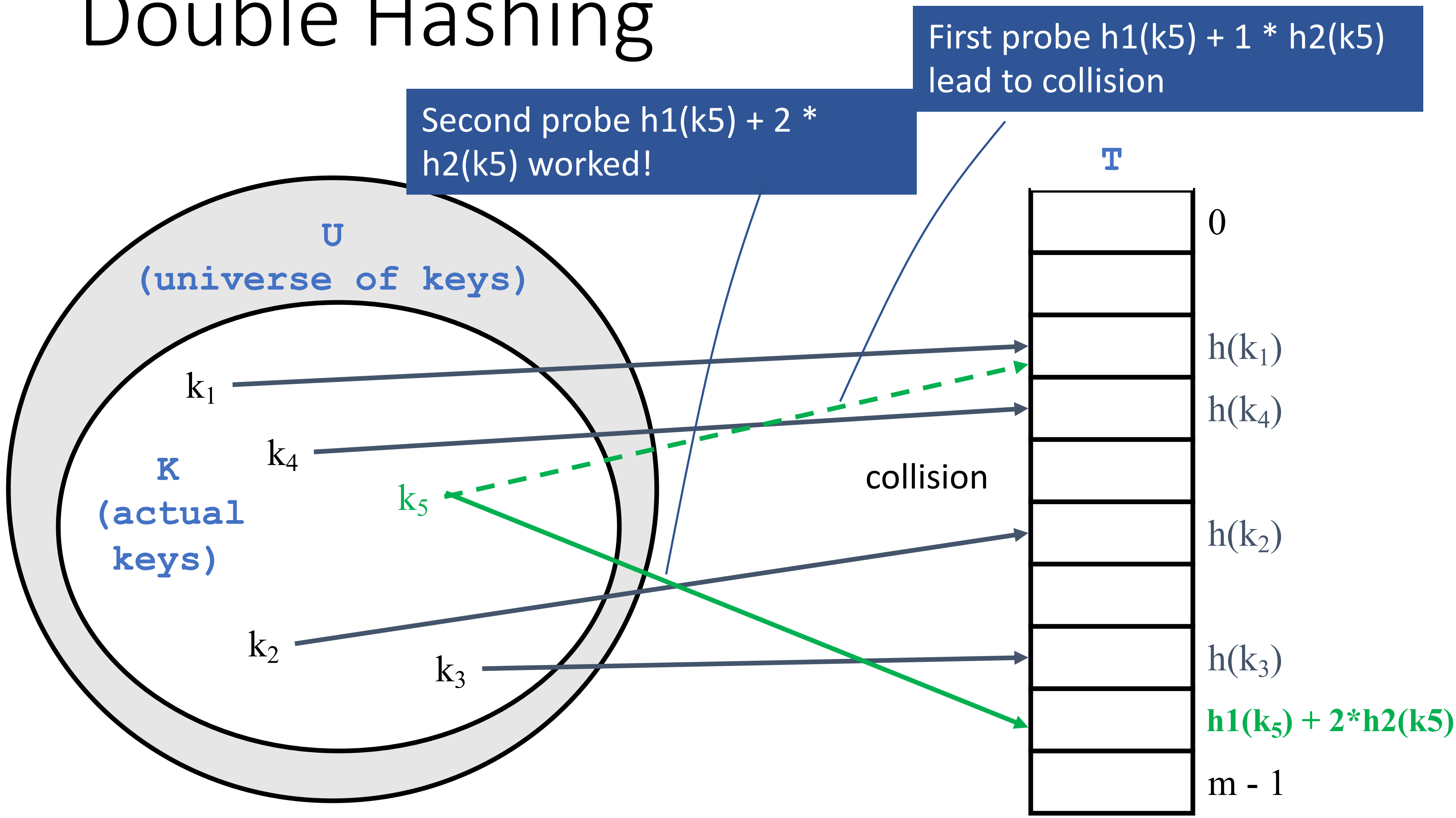
Quadratic Probing

k5 takes 2 probes to find it's slot
The quadratic probing makes it jump



- If current slot taken, **probe** based on **a quadratic equation** and continue till you find an empty slot and then take it
- **$h(k,i) = h(k)+i^2$ mod m** (where i is the probe count) [You can choose any quadratic equation based on i]

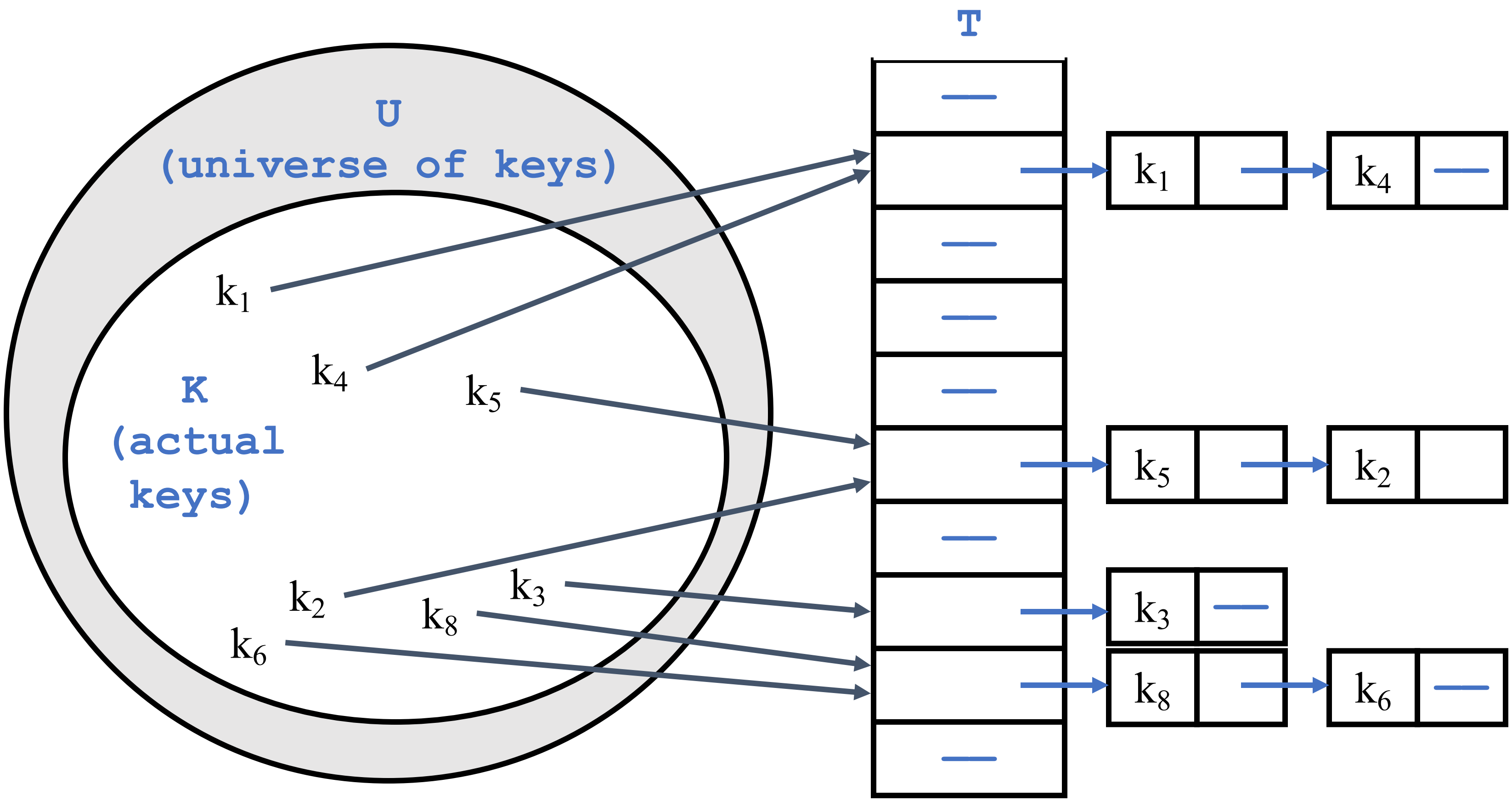
Double Hashing



- Use two hash functions instead of one every time you probe
- **h(k,i) = h1(k)+i * h2(k) mod m** (here, i is the probe count)

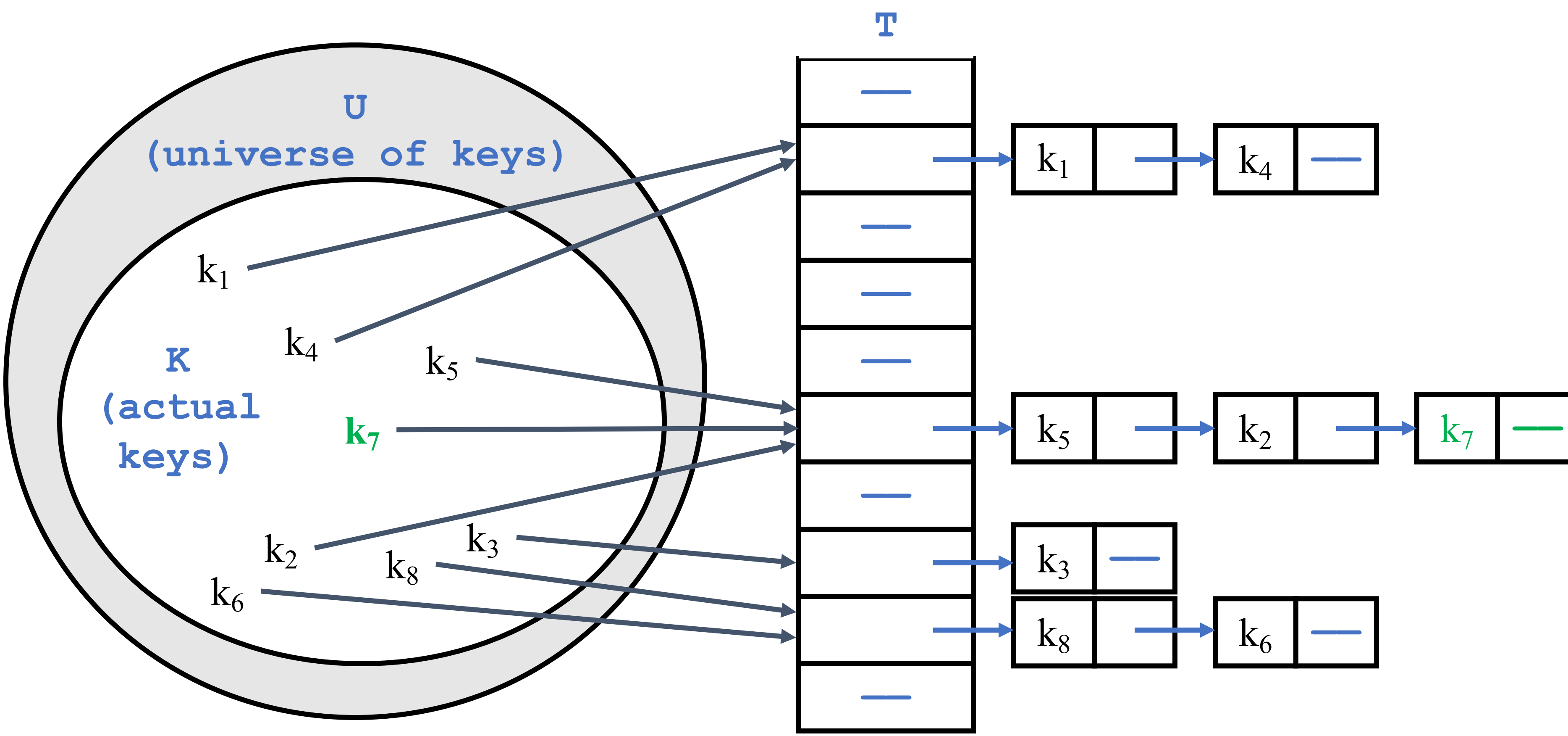
Alternative: Chaining

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



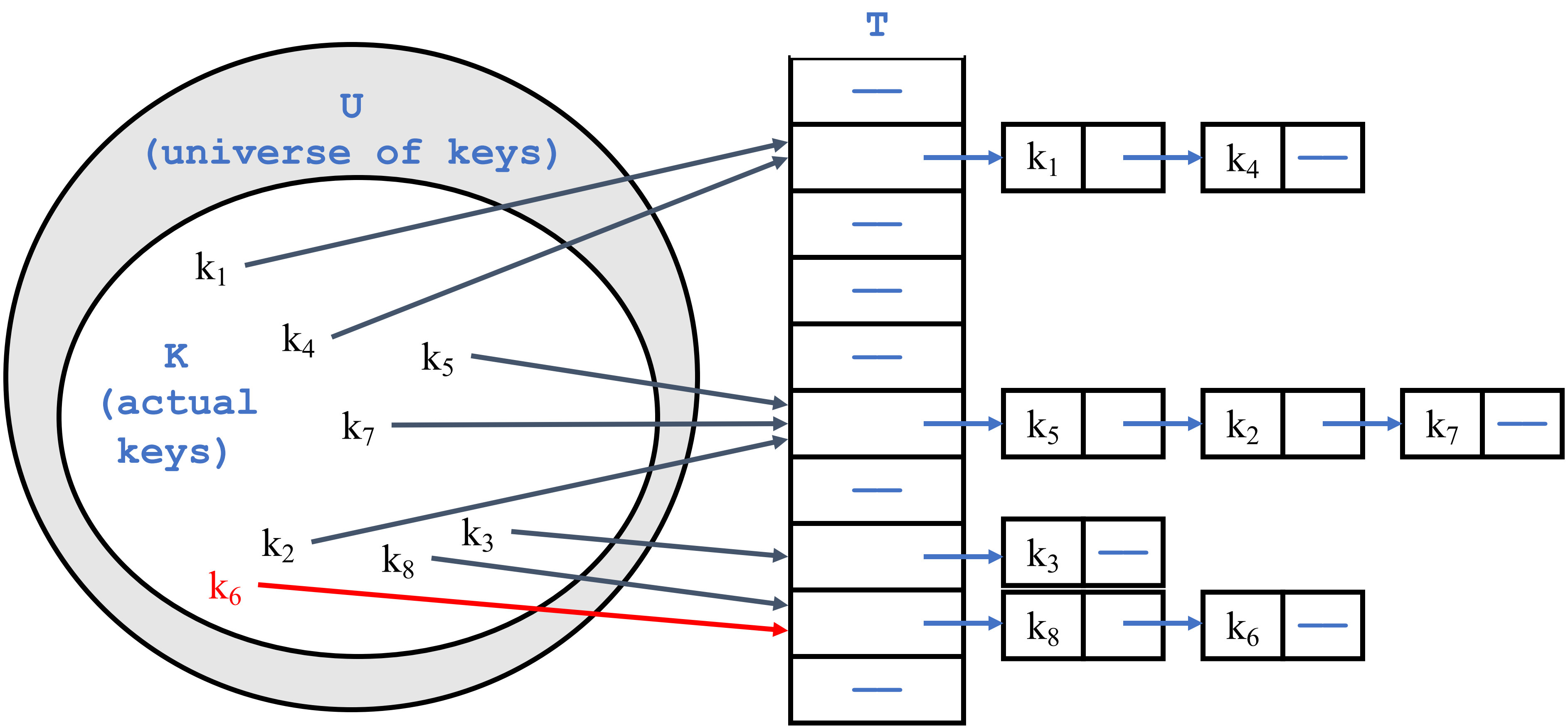
Chaining: Insert

- *How do we **insert** an element?*



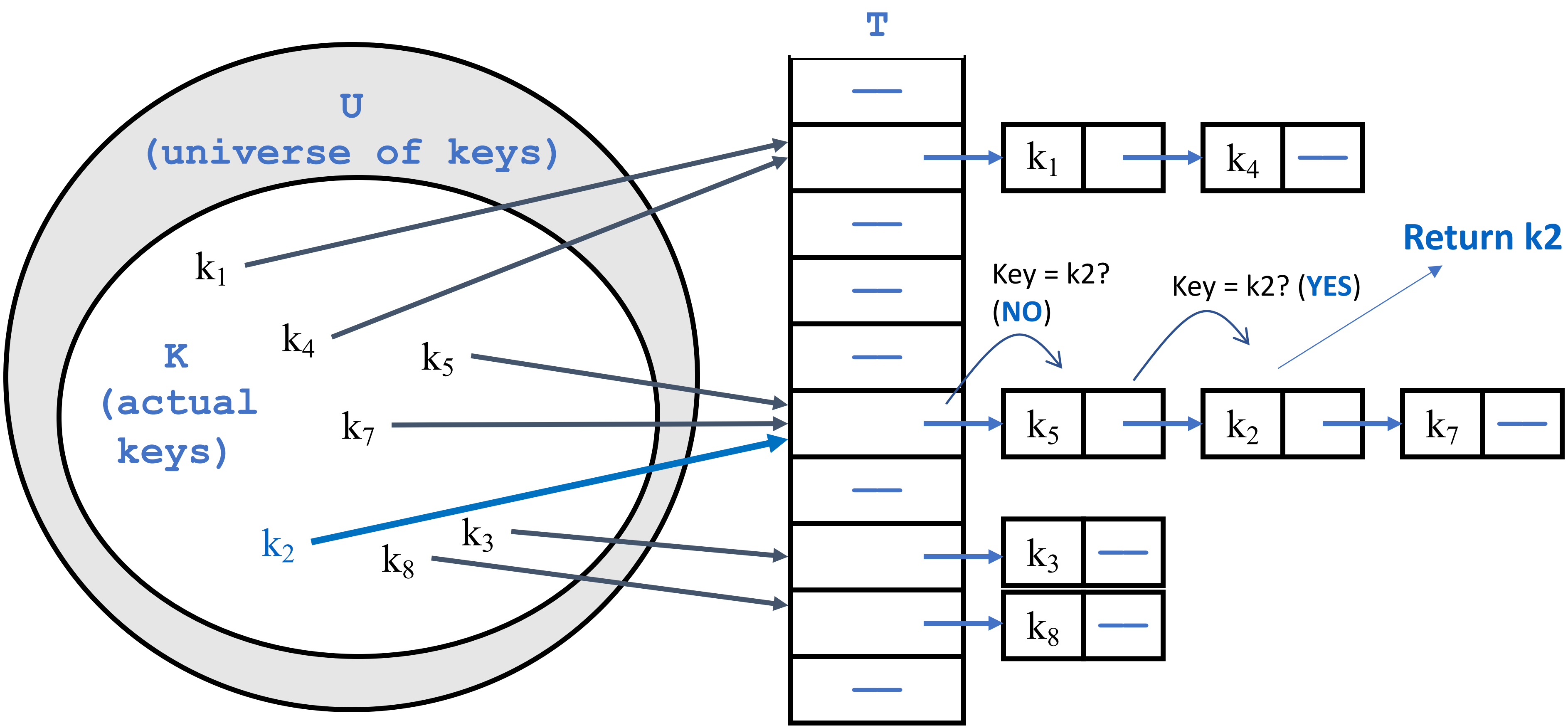
Chaining: Delete

- *How do we **delete** an element?*



Chaining:Search

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



Analysis of Chaining

- Assume ***simple uniform hashing***: each key is equally likely to be hashed to any slot

- Given n keys and m slots: the ***load factor*** $\alpha = n/m$ = average # keys per slot

- *What will be the average cost of an unsuccessful search for a key?*

Analysis of Chaining

- Assume ***simple uniform hashing***: each key is equally likely to be hashed to any slot

- Given *n* keys and *m* slots, the ***load factor*** $\alpha = n/m$ = average # keys per slot

- *What will be the average cost of an unsuccessful search for a key?* A: $O(1+\alpha)$

Analysis of Chaining

- Assume ***simple uniform hashing***: each key is equally likely to be hashed to any slot

- Given *n* keys and *m* slots, the ***load factor*** α = n/m = average # keys per slot

- *What will be the average cost of an unsuccessful search for a key?* A: $O(1+\alpha)$

- *What will be the average cost of a successful search?*

Analysis of Chaining

- Assume ***simple uniform hashing***: each key is equally likely to be hashed to any slot

- Given n keys and m slots, the ***load factor*** $\alpha = n/m$ = average # keys per slot

- *What will be the average cost of an unsuccessful search for a key?* A: $O(1+\alpha)$

- *What will be the average cost of a successful search?*

  A: $O(1 + \alpha/2) = O(1 + \alpha)$

Analysis of Chaining Continued

- So the cost of searching = O(1 + α)

- *If the number of keys n is proportional to the number of slots, what is α?*

- A: α = O(1)
  - In other words, **we can make the expected cost of searching constant if we make α constant**

Choosing A Hash Function

- Clearly choosing the hash function well is crucial
  - *What will a worst-case hash function do?*
    - *O(n)*
  - *What will be the time to search in this case?*
    - *O(n)*

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

Hash Functions: The Division Method

- $h(k) = k \bmod m$
  - In words: hash k into a table with m slots using the slot given by the remainder of k divided by m
- *What happens to elements with adjacent values of k?*
  - *They get stored in adjacent locations (mostly)*
- *What happens if m is a power of 2 (say 2^P)?*
  - Depends only on few least significant bits
  - Higher bits not used
  - Not a good idea to use!

- Pick table size m = prime number not too close to a power of 2 (or 10)

Hash Functions: The Multiplication Method

- For a constant A, $0 < A < 1$:
- h(k) = $\lfloor m\ (k\ A\ mod\ 1)\ \rfloor$

- Slower than division method, but choice of m not so critical
  - Typically choose $m = 2^P$

- Choose A not too close to 0 or 1
  - Good choice for $A = (\sqrt{5} - 1)/2$

Hash Functions: Universal Hashing

- ***Universal hashing***: pick a hash function randomly in a way that is independent of the keys that are actually going to be stored
  - Guarantees good performance on average

Universal Hashing

- Let H be a (finite) collection of hash functions
  - …that map a given universe U of keys…
  - …into the range $\{0, 1, \ldots, m - 1\}$.

- H is said to be *universal* if:
  - for each pair of distinct keys $x, y \in U$, the number of hash functions $h \in H$ for which $h(x) = h(y)$ is $|H|/m$
  - In other words:
    - With a random hash function from H, the chance of a collision between x and y is exactly $1/m$ $(x \neq y)$


That's all Folks!
Any Question ?