Data Structures & Algorithms

Week 3 - Hashing (HashTables, HashMaps, Dictionaries)

Subodh Sharma, Rahul Garg

{svs,rahulgarg}@iitd.ac.in

Recap: Hash Functions

Recap: Hash Functions

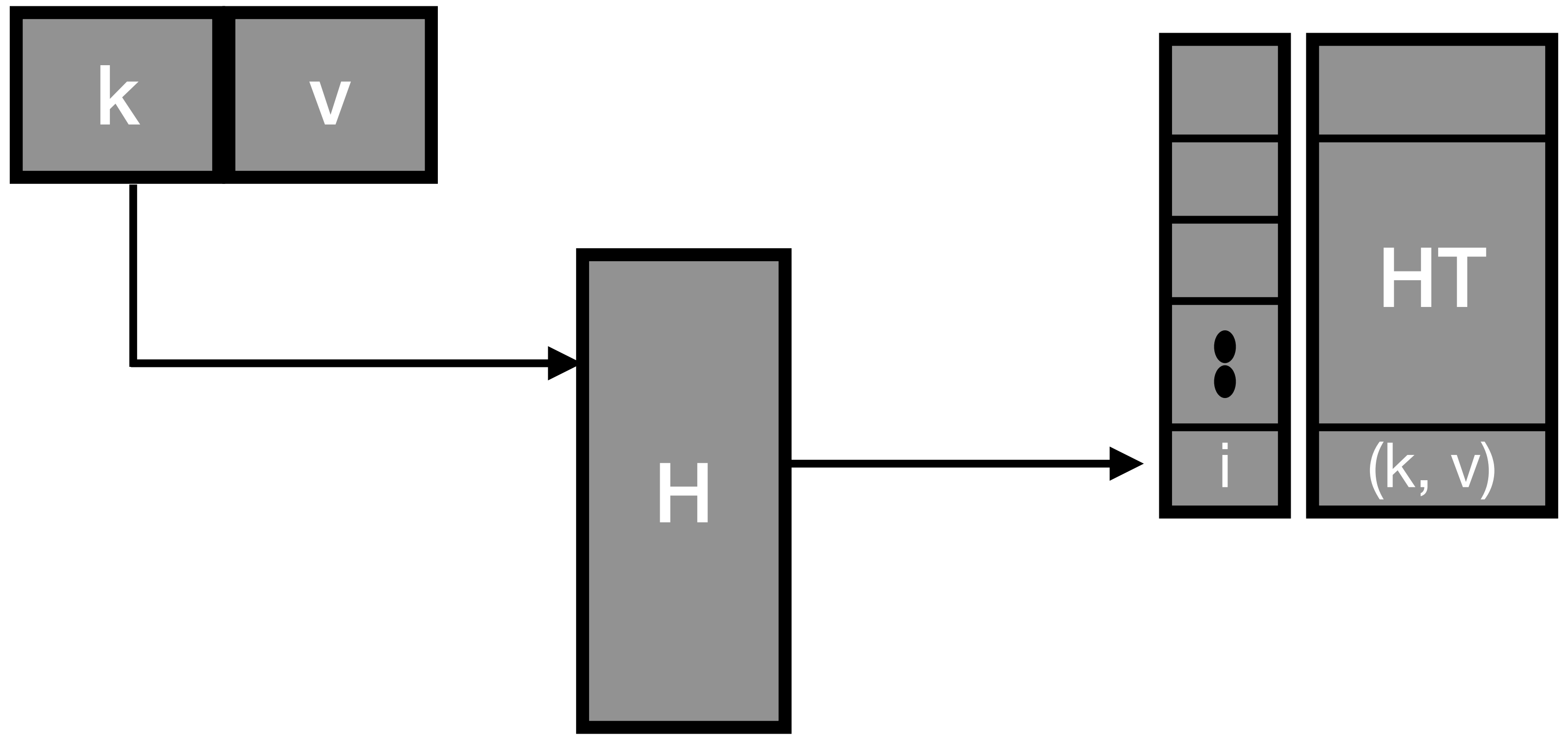
- A hash function **H** takes a key **k** and generates a hash value **H(k) = i**

Recap: Hash Functions

- A hash function **H** takes a key **k** and generates a hash value **H(k) = i**
- Hash Table **HT** stores key-value pair **(k,v)** at **HT[I]**

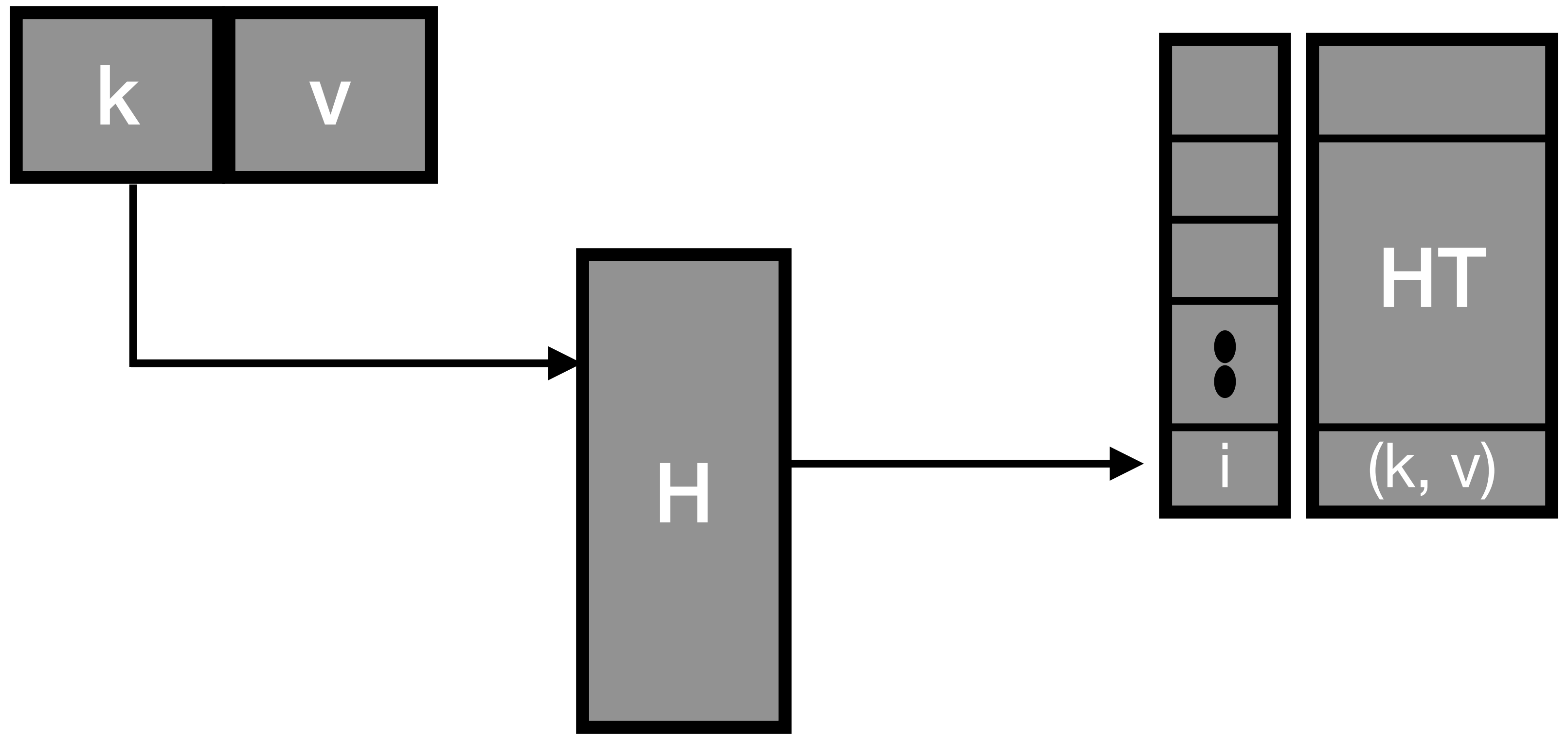
Recap: Hash Functions

- A hash function **H** takes a key **k** and generates a hash value **H(k) = i**
- Hash Table **HT** stores key-value pair **(k,v)** at **HT[I]**



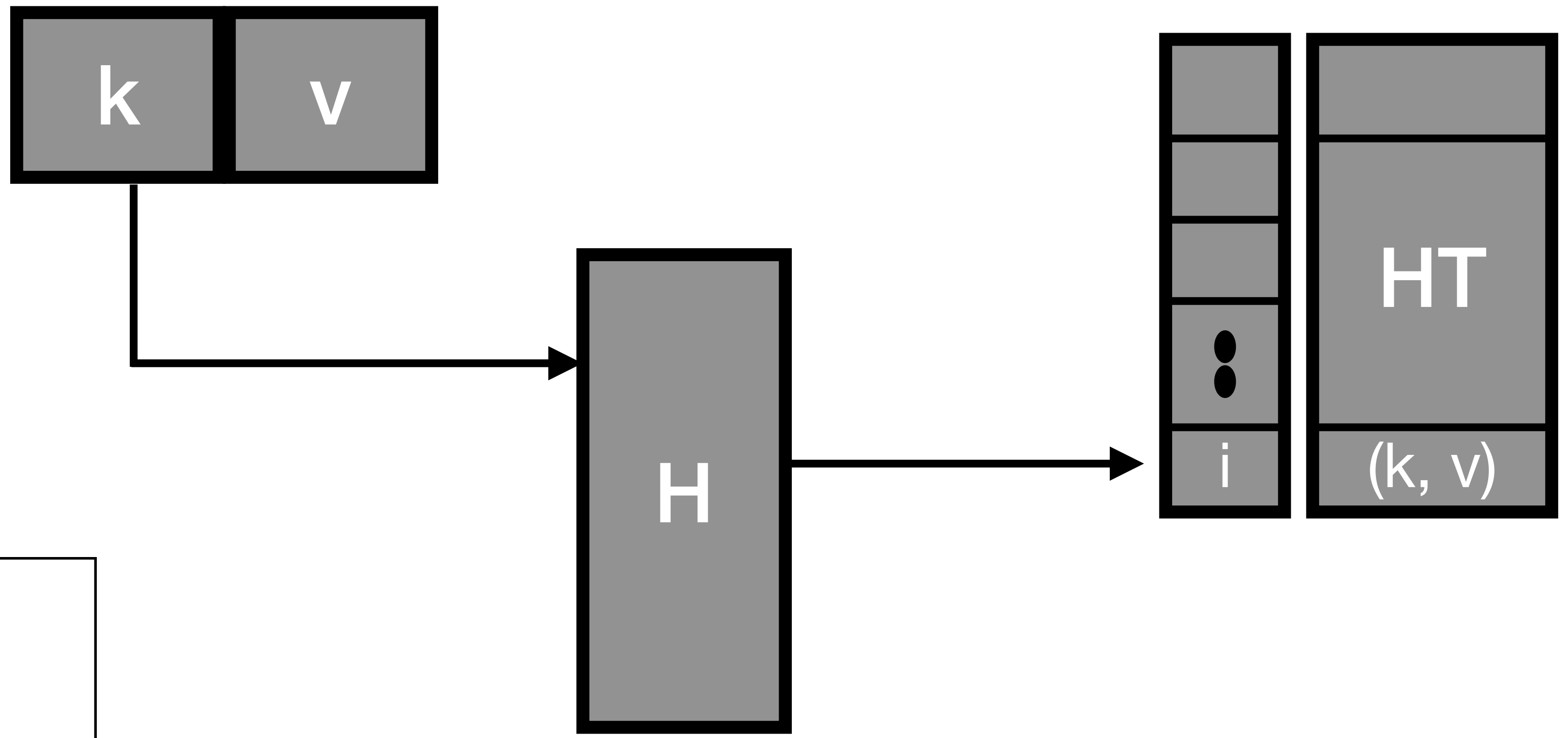
Recap: Hash Functions

- A hash function **H** takes a key **k** and generates a hash value **H(k) = i**
- Hash Table **HT** stores key-value pair **(k,v)** at **HT[I]**
- **Motivation:** Fast storage and retrieval



Recap: Hash Functions

- A hash function **H** takes a key **k** and generates a hash value **H(k) = i**
- Hash Table **HT** stores key-value pair **(k,v)** at **HT[I]**
- **Motivation:** Fast storage and retrieval



	LL, no dup	**Sorted Array wit h Binary Search**	**Hash Table**
Insert	O(n)	O(log n)	O(1)
Delete	O(n)	O(log n)	O(1)
Contains	O(n)	O(log n)	O(1)

Collision (Recap)

Collision (Recap)

- There are ~500 students in the COL106 class. Assume we have an **HT** of 500.

Collision (Recap)

- There are ~500 students in the COL106 class. Assume we have an **HT** of 500.
- Hash function: last two digits of the entry # of students

Collision (Recap)

- There are ~500 students in the COL106 class. Assume we have an **HT** of 500.
- Hash function: last two digits of the entry # of students
- Where does 2022CS10110 and 2022CS50310 go?

Collision (Recap)

- There are ~500 students in the COL106 class. Assume we have an **HT** of 500.
- Hash function: last two digits of the entry # of students
- Where does 2022CS10110 and 2022CS50310 go?
  - To location 10 in **HT**. This is **COLLISION!**

Collision (Recap)

- There are ~500 students in the COL106 class. Assume we have an **HT** of 500.
- Hash function: last two digits of the entry # of students
- Where does 2022CS10110 and 2022CS50310 go?
  - To location 10 in **HT**. This is **COLLISION!**
- **Collision Resolution:**

Collision (Recap)

- There are ~500 students in the COL106 class. Assume we have an **HT** of 500.
- Hash function: last two digits of the entry # of students
- Where does 2022CS10110 and 2022CS50310 go?
  - To location 10 in **HT**. This is **COLLISION!**
- **Collision Resolution:**
  - **Open Hashing**: Chaining

Collision (Recap)

- There are ~500 students in the COL106 class. Assume we have an **HT** of 500.
- Hash function: last two digits of the entry # of students
- Where does 2022CS10110 and 2022CS50310 go?
  - To location 10 in **HT**. This is **COLLISION!**
- **Collision Resolution:**
  - **Open Hashing**: Chaining
  - **Closed Hashing:** Linear probing, Quadratic probing, Double Hashing

A Good Hash Function: Reqs

A Good Hash Function: Reqs

- **Uniform Distribution**: Distribution of keys uniformly across the HT
    - This minimises collision, improves HT utilisation!

A Good Hash Function: Reqs

- **Uniform Distribution**: Distribution of keys uniformly across the HT
  - This minimises collision, improves HT utilisation!
- **Collision Resistant:** Computationally infeasible to find $x, y : H(x) = H(y) \wedge x \neq y$

A Good Hash Function: Reqs

- **Uniform Distribution**: Distribution of keys uniformly across the HT
    - This minimises collision, improves HT utilisation!
- **Collision Resistant:** Computationally infeasible to find $x, y : H(x) = H(y) \wedge x \neq y$
- **Deterministic and Fast computation**

A Good Hash Function: Reqs

- **Uniform Distribution**: Distribution of keys uniformly across the HT
    - This minimises collision, improves HT utilisation!
- **Collision Resistant:** Computationally infeasible to find $x, y : H(x) = H(y) \wedge x \neq y$
- **Deterministic and Fast computation**
- **Using all of the input data:** Every part of input affects the output hash
    - $\exists i \in \mathbb{N} : x_i \neq y_i \Rightarrow P(H(x) \neq H(y)) > 0$

A Good Hash Function: Reqs

- **Uniform Distribution**: Distribution of keys uniformly across the HT
    - This minimises collision, improves HT utilisation!
- **Collision Resistant:** Computationally infeasible to find $x, y : H(x) = H(y) \wedge x \neq y$
- **Deterministic and Fast computation**
- **Using all of the input data:** Every part of input affects the output hash
    - $\exists i \in \mathbb{N} : x_i \neq y_i \Rightarrow P(H(x) \neq H(y)) > 0$
- **Dynamic:** Dynamic resizing of HT should be possibles

Cryptographic Hashing vs Hashing

Cryptographic Hashing vs Hashing

- Cryptographic hashes require additional properties

Cryptographic Hashing vs Hashing

- Cryptographic hashes require additional properties
    - **Preimage Resistance** (One Way property): Let $H(x) = h$. Given h, it is computationally intractable to find x

Cryptographic Hashing vs Hashing

- Cryptographic hashes require additional properties
  - **Preimage Resistance** (One Way property): Let $H(x) = h$. Given h, it is computationally intractable to find x
  - **Second Preimage Resistance**: Given x_1, it should be computationally infeasible to find x_2 s.t. $x_1 \neq x_2 \wedge H(x_1) = H(x_2)$

Cryptographic Hashing vs Hashing

- Cryptographic hashes require additional properties
  - **Preimage Resistance** (One Way property): Let $H(x) = h$. Given h, it is computationally intractable to find x
  - **Second Preimage Resistance**: Given x_1, it should be computationally infeasible to find x_2 s.t. $x_1 \neq x_2 \wedge H(x_1) = H(x_2)$
  - **Avalanche effect:** Small change in the input produces significant change in the output

Cryptographic Hashing vs Hashing

- Cryptographic hashes require additional properties
  - **Preimage Resistance** (One Way property): Let $H(x) = h$. Given h, it is computationally intractable to find x
  - **Second Preimage Resistance**: Given x_1, it should be computationally infeasible to find x_2 s.t. $x_1 \neq x_2 \wedge H(x_1) = H(x_2)$
  - **Avalanche effect:** Small change in the input produces significant change in the output
    - $\forall x, y : d_H(x, y) = 1 \Rightarrow P(H(x)_j \neq H(y)_j) \geq 0.5$

Collision Resolution (1): Chaining

Collision Resolution (1): Chaining

- For *find/insert/delete an element **e***

Collision Resolution (1): Chaining

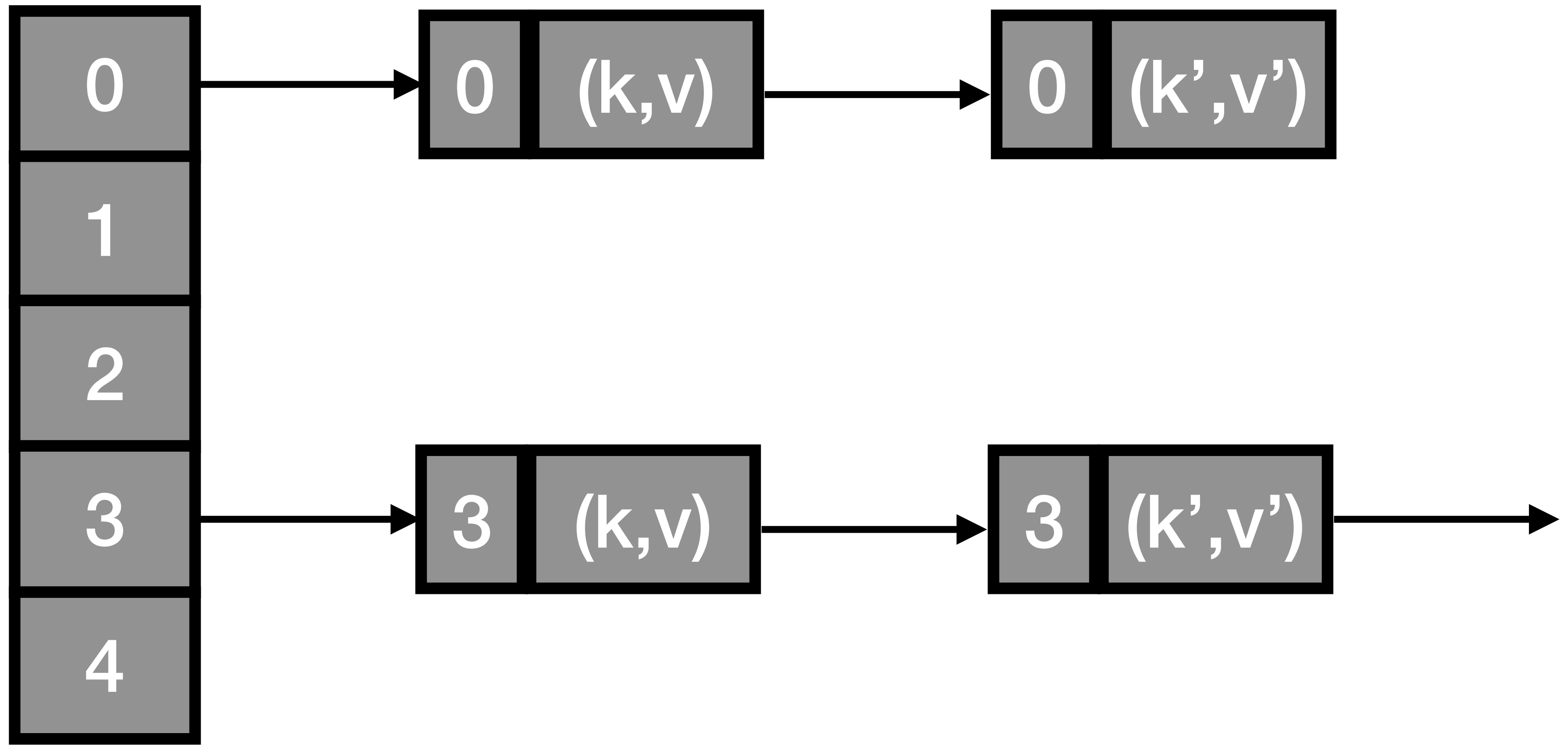
- For *find/insert/delete an element* ***e***
  - Use **H(.)** to look up the position of **e** in **HT**

Collision Resolution (1): Chaining

- For *find/insert/delete an element* ***e***
  - Use **H(.)** to look up the position of **e** in **HT**
  - Find/insert/delete in the linked list of the hashed slot

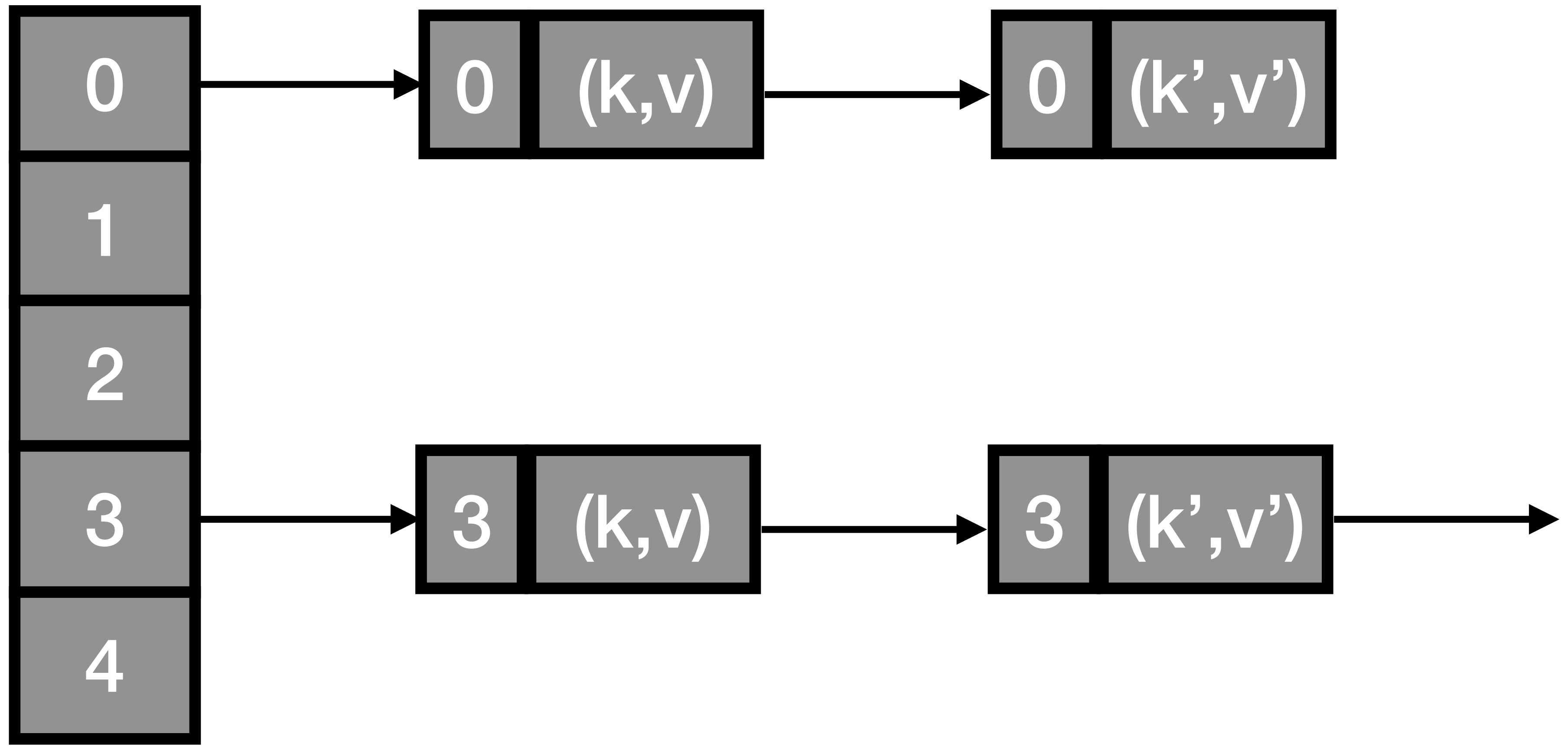
Collision Resolution (1): Chaining

- For *find/insert/delete an element* ***e***
  - Use **H(.)** to look up the position of **e** in **HT**
  - Find/insert/delete in the linked list of the hashed slot



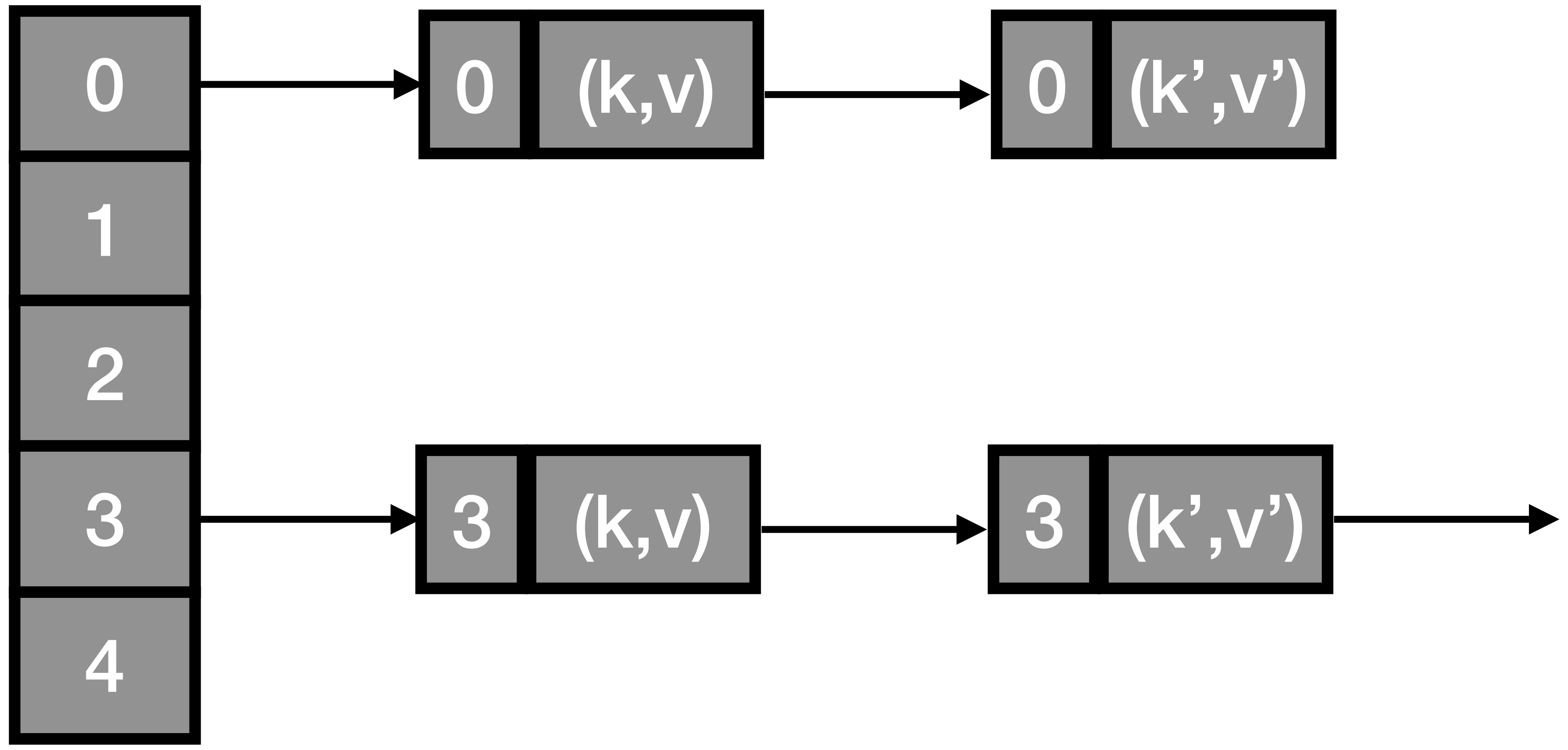
Collision Resolution (1): Chaining

- For *find/insert/delete an element* ***e***
    - Use **H(.)** to look up the position of **e** in **HT**
    - Find/insert/delete in the linked list of the hashed slot
- **Analysis:**



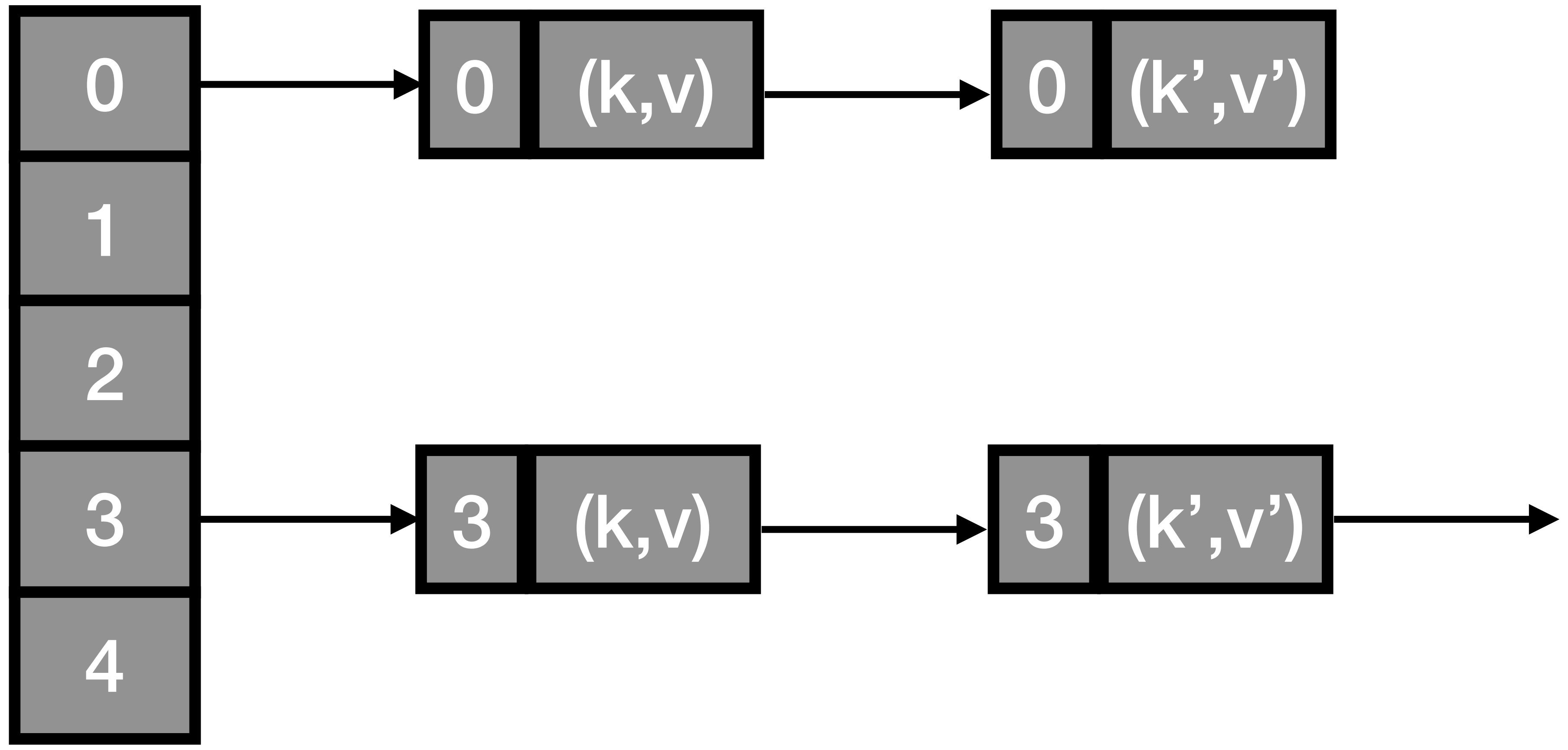
Collision Resolution (1): Chaining

- For *find/insert/delete an element* ***e***
    - Use **H(.)** to look up the position of **e** in **HT**
    - Find/insert/delete in the linked list of the hashed slot
- **Analysis:**
    - Assume that time to compute **H(k)** is $\Theta(1)$



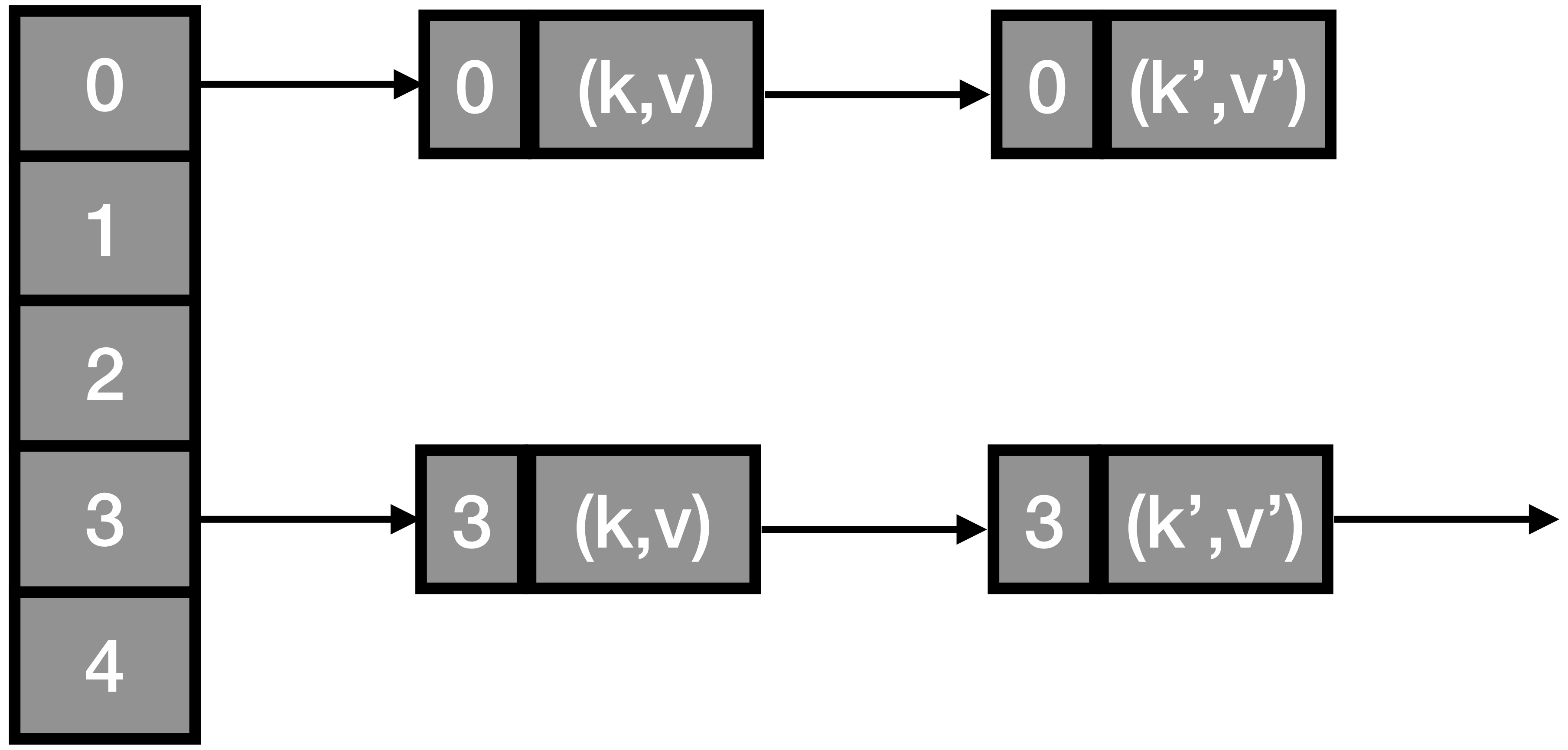
Collision Resolution (1): Chaining

- For *find/insert/delete an element* ***e***
    - Use **H(.)** to look up the position of **e** in **HT**
    - Find/insert/delete in the linked list of the hashed slot
- **Analysis:**
    - Assume that time to compute **H(k)** is $\Theta(1)$
    - **Load factor** of an **HT:** $\alpha = n/m$ with **m** slots holding **n** elements



Collision Resolution (1): Chaining

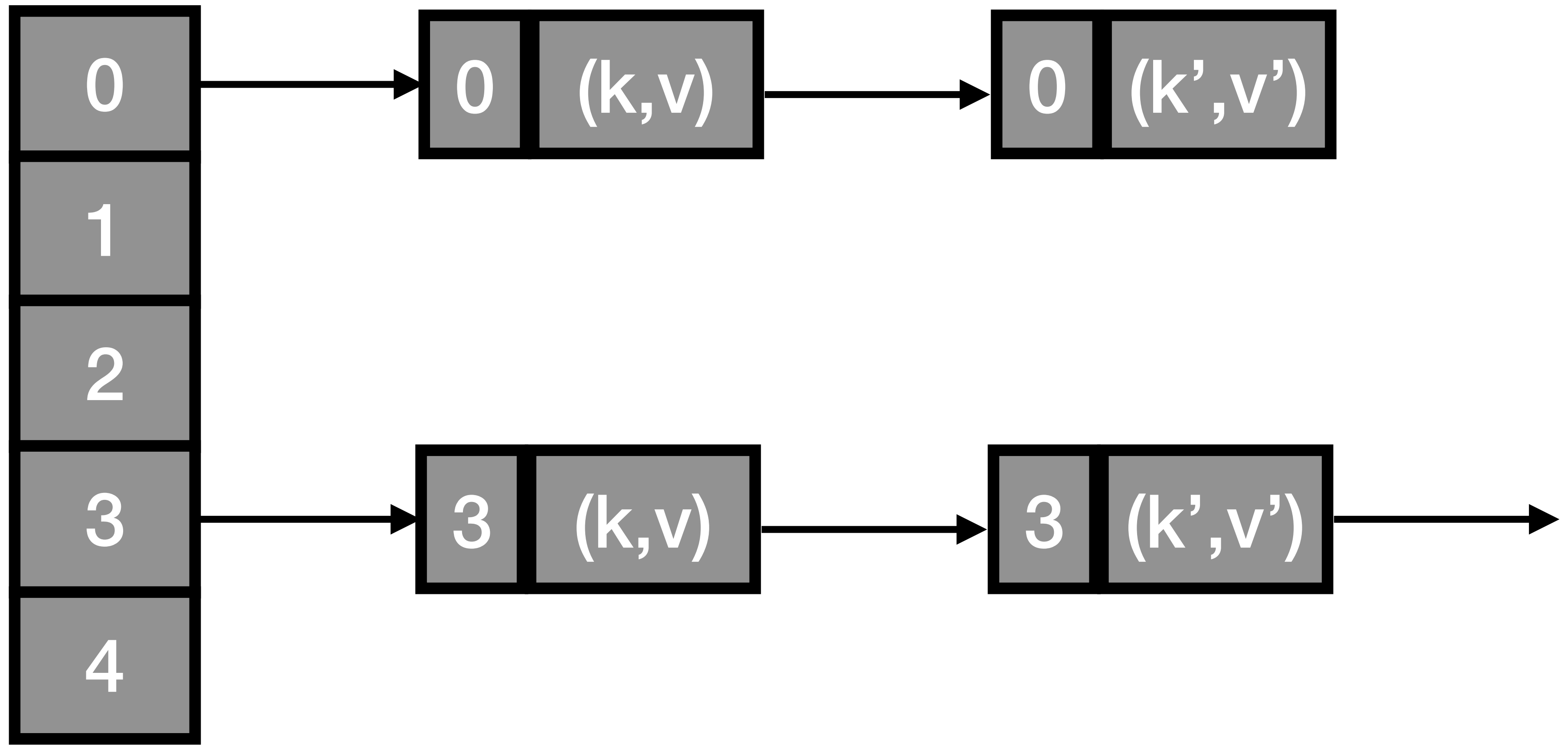
- For *find/insert/delete an element* ***e***
  - Use **H(.)** to look up the position of **e** in **HT**
  - Find/insert/delete in the linked list of the hashed slot
- **Analysis:**
  - Assume that time to compute **H(k)** is $\Theta(1)$
  - **Load factor** of an **HT:** $\alpha = n/m$ with **m** slots holding **n** elements



- Use of **Load factor**?

Collision Resolution (1): Chaining

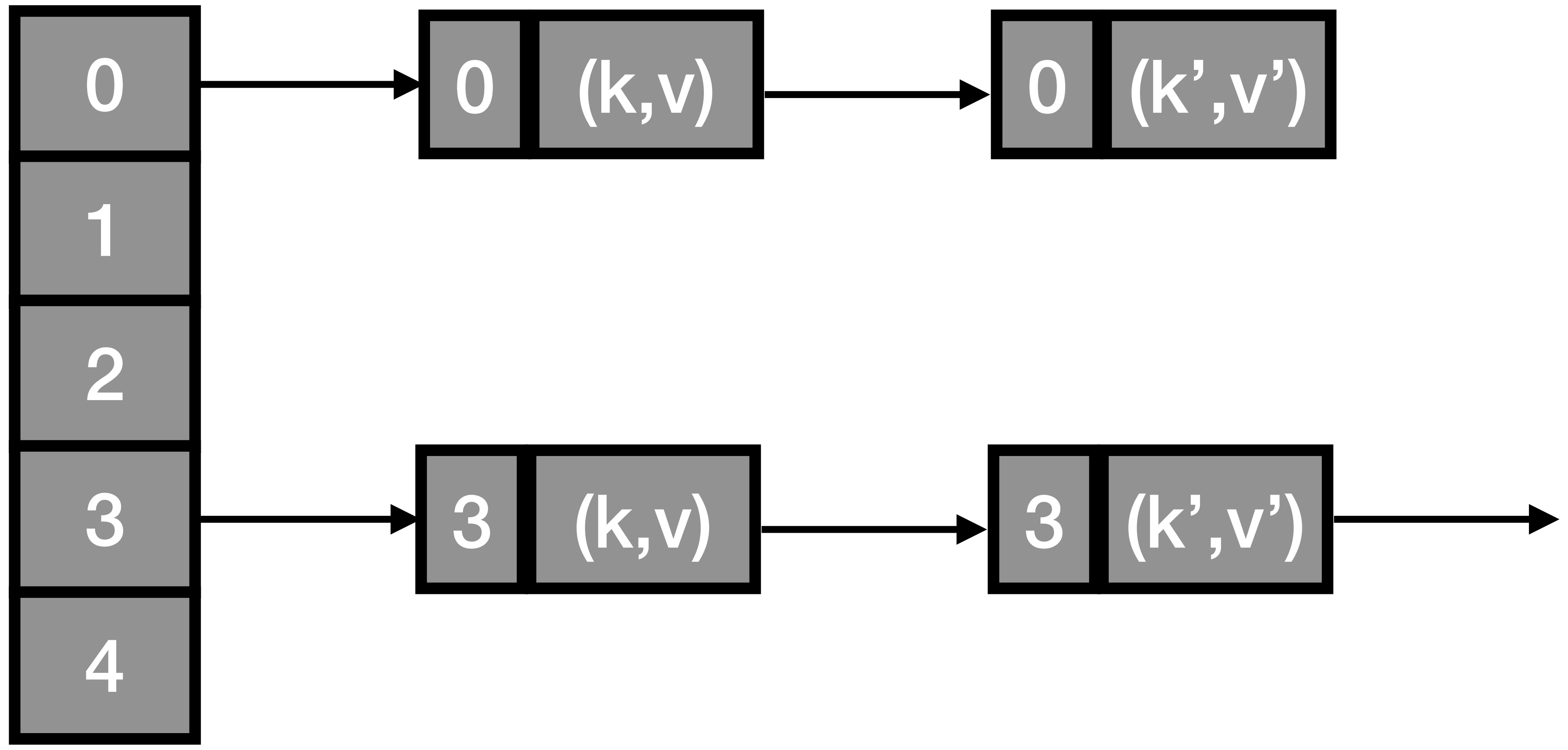
- For *find/insert/delete an element* ***e***
    - Use **H(.)** to look up the position of **e** in **HT**
    - Find/insert/delete in the linked list of the hashed slot
- **Analysis:**
    - Assume that time to compute **H(k)** is $\Theta(1)$
    - **Load factor** of an **HT:** $\alpha = n/m$ with **m** slots holding **n** elements



- Use of **Load factor**?
    - Performance Indication: α approaches 1, the probability of collision increases

Collision Resolution (1): Chaining

- For *find/insert/delete an element* ***e***
  - Use **H(.)** to look up the position of **e** in **HT**
  - Find/insert/delete in the linked list of the hashed slot
- **Analysis:**
  - Assume that time to compute **H(k)** is $\Theta(1)$
  - **Load factor** of an **HT:** $\alpha = n/m$ with **m** slots holding **n** elements



- Use of **Load factor**?
  - Performance Indication: α approaches 1, the probability of collision increases
  - Threshold for resizing: When $\alpha > 0.7$, **HT** is doubled in size

Collision Resolution (1): Chaining

Collision Resolution (1): Chaining

- **Analysis:**

Collision Resolution (1): Chaining

- **Analysis:**
  - Assume that time to compute **H(k)** is $\Theta(1)$

Collision Resolution (1): Chaining

- **Analysis:**
  - Assume that time to compute **H(k)** is $\Theta(1)$
  - **Load factor** of an **HT:** $\alpha = n/m$ with **m** slots holding **n** elements

Collision Resolution (1): Chaining

- **Analysis:**
  - Assume that time to compute **H(k)** is $\Theta(1)$
  - **Load factor** of an **HT:** $\alpha = n/m$ with **m** slots holding **n** elements
  - Expected number of elements to be examined: α

Collision Resolution (1): Chaining

- **Analysis:**
  - Assume that time to compute **H(k)** is $\Theta(1)$
  - **Load factor** of an **HT:** $\alpha = n/m$ with **m** slots holding **n** elements
  - Expected number of elements to be examined: α
  - **Search time** for unsuccessful search: $O(1 + \alpha)$

Collision Resolution (1): Chaining

- **Analysis:**
  - Assume that time to compute **H(k)** is $\Theta(1)$
  - **Load factor** of an **HT:** $\alpha = n/m$ with **m** slots holding **n** elements
  - Expected number of elements to be examined: α
  - **Search time** for unsuccessful search: $O(1+\alpha)$
  - **Search time** for successful search:

$$1 + \frac{\sum_{i=1}^{n} \frac{(i-1)}{m}}{n} \simeq \Theta(1+\alpha)$$

Collision Resolution (1): Chaining

- **Analysis:**
  - Assume that time to compute **H(k)** is $\Theta(1)$
  - **Load factor** of an **HT:** $\alpha = n/m$ with **m** slots holding **n** elements
  - Expected number of elements to be examined: α
  - **Search time** for unsuccessful search: $O(1 + \alpha)$
  - **Search time** for successful search:

$$1 + \frac{\sum_{i=1}^{n} \frac{(i-1)}{m}}{n} \simeq \Theta(1 + \alpha)$$

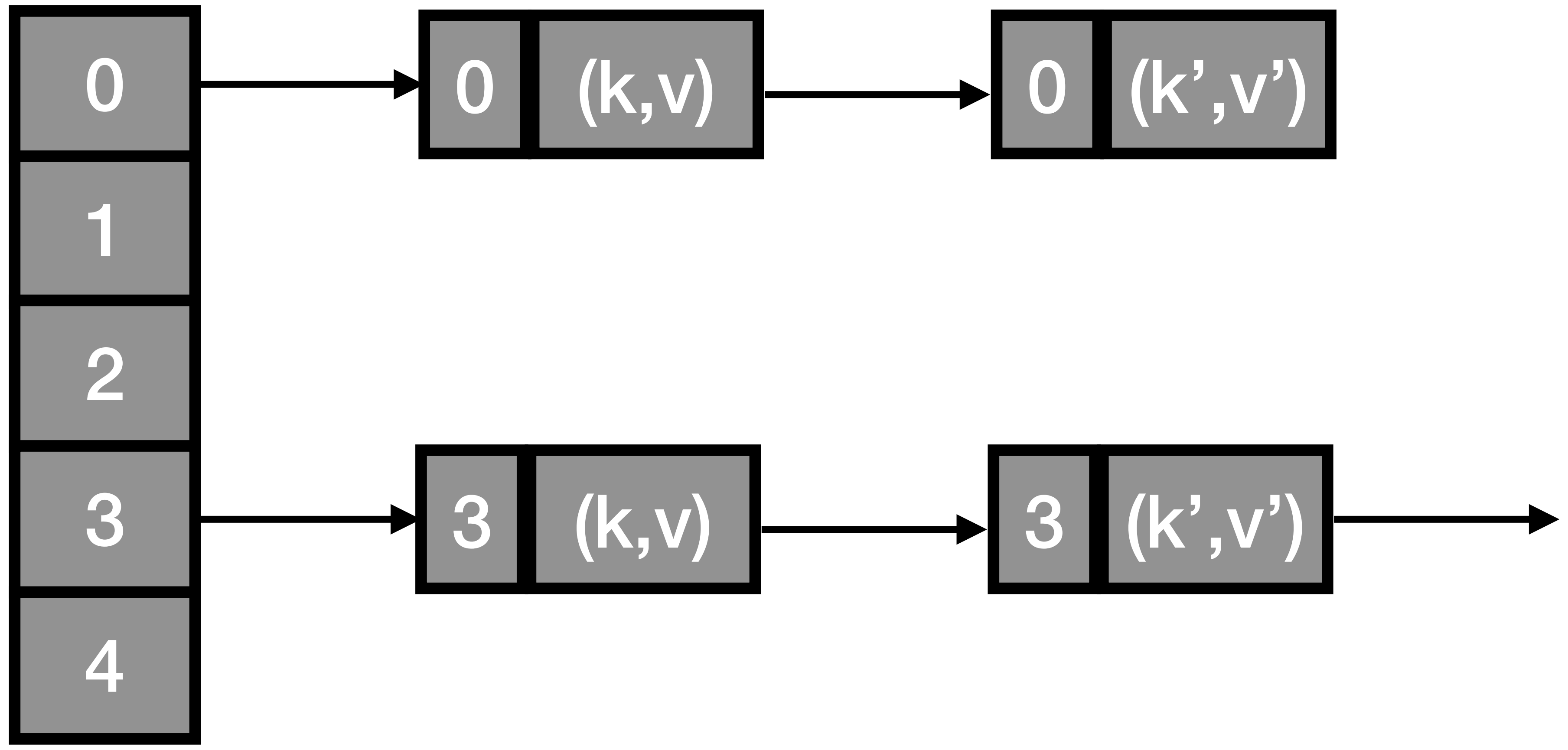
    - Upon insertion of i-th element: expected length of the list: (i-1)/m

Collision Resolution (1): Chaining

- **Analysis:**
  - Assume that time to compute **H(k)** is $\Theta(1)$
  - **Load factor** of an **HT:** $\alpha = n/m$ with **m** slots holding **n** elements
  - Expected number of elements to be examined: α
  - **Search time** for unsuccessful search: $O(1+\alpha)$
  - **Search time** for successful search:

$$1 + \frac{\sum_{i=1}^{n}\frac{(i-1)}{m}}{n} \simeq \Theta(1+\alpha)$$

  - Upon insertion of i-th element: expected length of the list: (i-1)/m

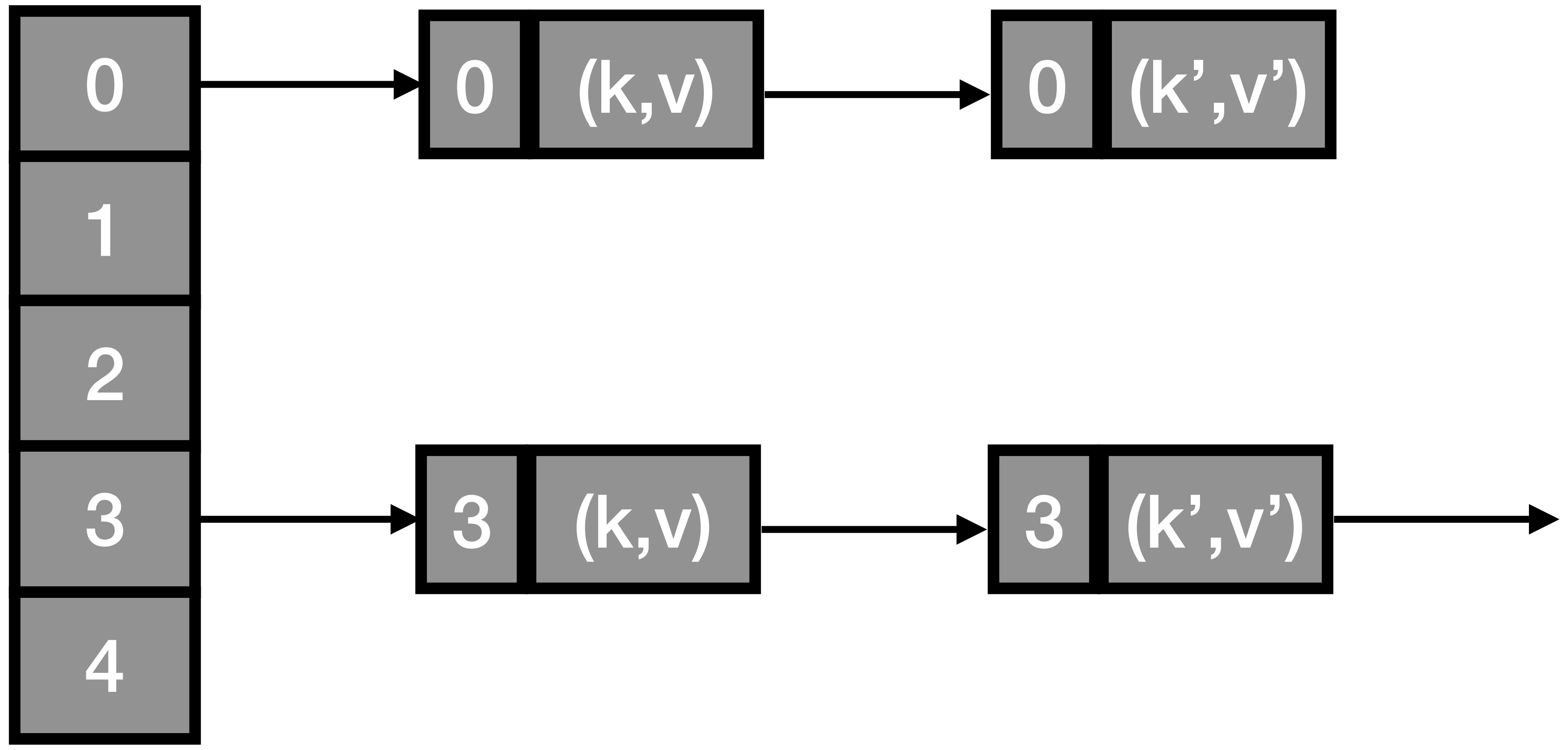


Collision Resolution (1): Chaining

- **Analysis:**
  - Assume that time to compute **H(k)** is $\Theta(1)$
  - **Load factor** of an **HT:** $\alpha = n/m$ with **m** slots holding **n** elements
  - Expected number of elements to be examined: α
  - **Search time** for unsuccessful search: $O(1+\alpha)$
  - **Search time** for successful search:

$$1 + \frac{\sum_{i=1}^{n} \frac{(i-1)}{m}}{n} \simeq \Theta(1+\alpha)$$

    - Upon insertion of i-th element: expected length of the list: (i-1)/m



- Assumption: We use a **simple uniform** hash function

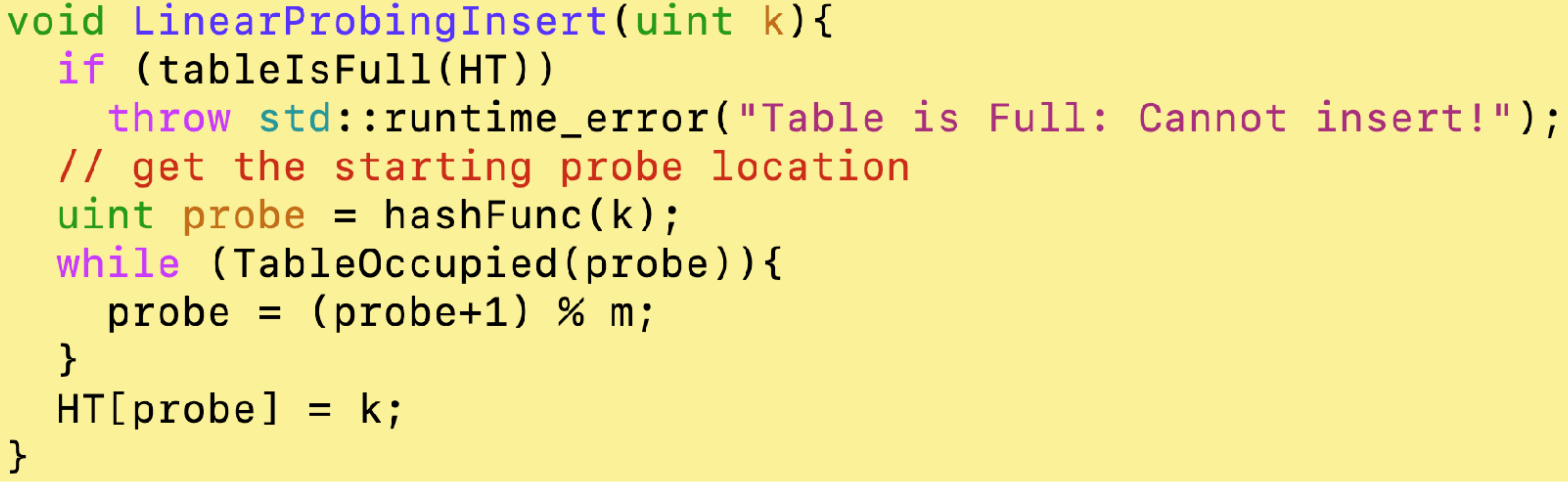
Collision Resolution (2): Linear Probing

Collision Resolution (2): Linear Probing

- Each slot can accommodate one element (i.e. $n \leq m$)
- It uses **less** memory than Chaining, but slower
- **Algorithm:** Insert

Collision Resolution (2): Linear Probing

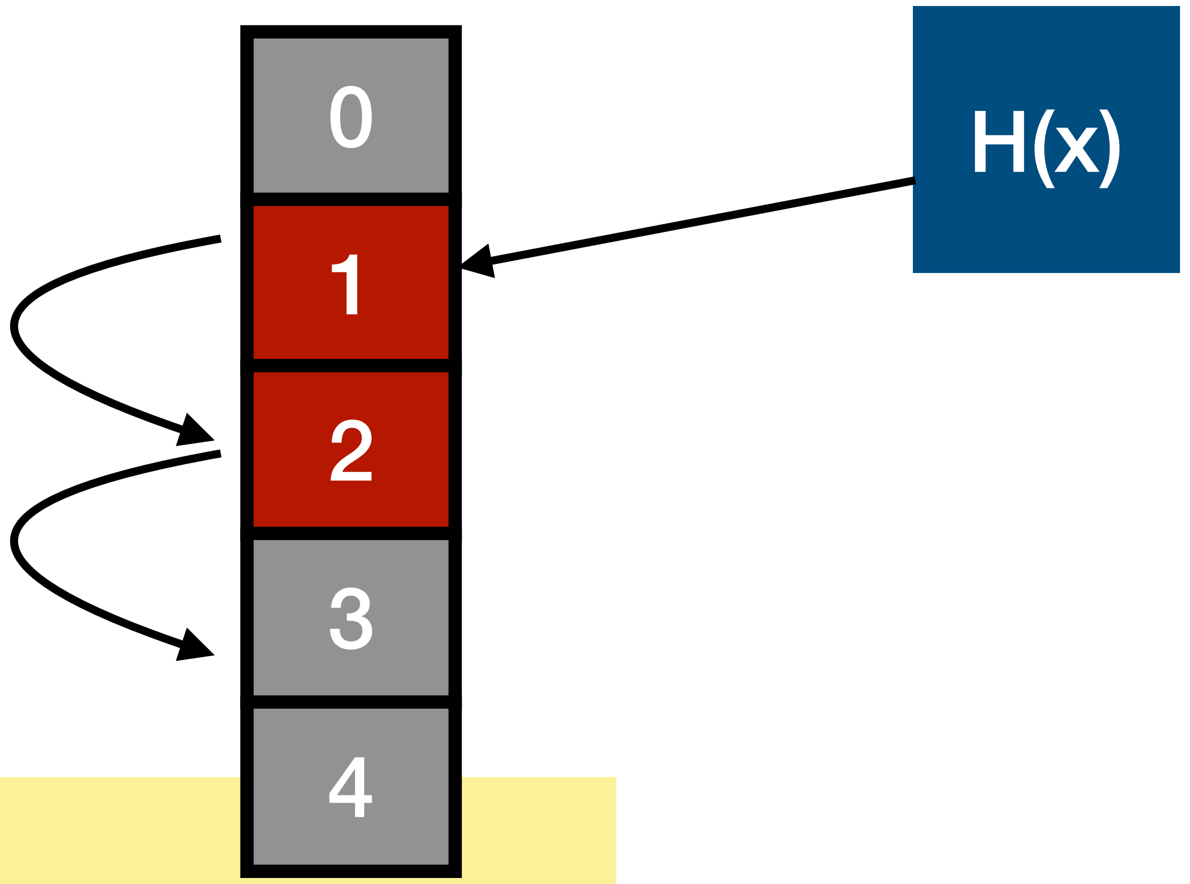
- Each slot can accommodate one element (i.e. $n \leq m$)
- It uses **less** memory than Chaining, but slower
- **Algorithm:** Insert



```
void LinearProbingInsert(uint k){
  if (tableIsFull(HT))
    throw std::runtime_error("Table is Full: Cannot insert!");
  // get the starting probe location
  uint probe = hashFunc(k);
  while (TableOccupied(probe)){
    probe = (probe+1) % m;
  }
  HT[probe] = k;
}
```

Collision Resolution (2): Linear Probing

- Each slot can accommodate one element (i.e. $n \leq m$)
- It uses **less** memory than Chaining, but slower
- **Algorithm:** Insert



```
void LinearProbingInsert(uint k){
  if (tableIsFull(HT))
    throw std::runtime_error("Table is Full: Cannot insert!");
  // get the starting probe location
  uint probe = hashFunc(k);
  while (TableOccupied(probe)){
    probe = (probe+1) % m;
  }
  HT[probe] = k;
}
```

Collision Resolution (2): Linear Probing

Collision Resolution (2): Linear Probing

- Example: $H(k) = k \mod 13$
- Insert keys: 18, 41, 22, 44, 59, 32, 31
- **Successful search**: Go to the hash value and continue until the location
- **Unsuccessful search**: Go the hash value and continue until the end of the array OR an empty location

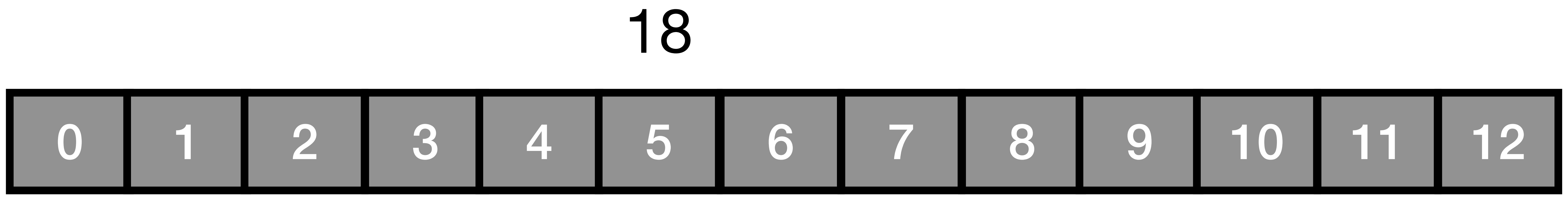
Collision Resolution (2): Linear Probing

- Example: $H(k) = k \mod 13$
- Insert keys: 18, 41, 22, 44, 59, 32, 31
- **Successful search**: Go to the hash value and continue until the location
- **Unsuccessful search**: Go the hash value and continue until the end of the array OR an empty location

0	1	2	3	4	5	6	7	8	9	10	11	12

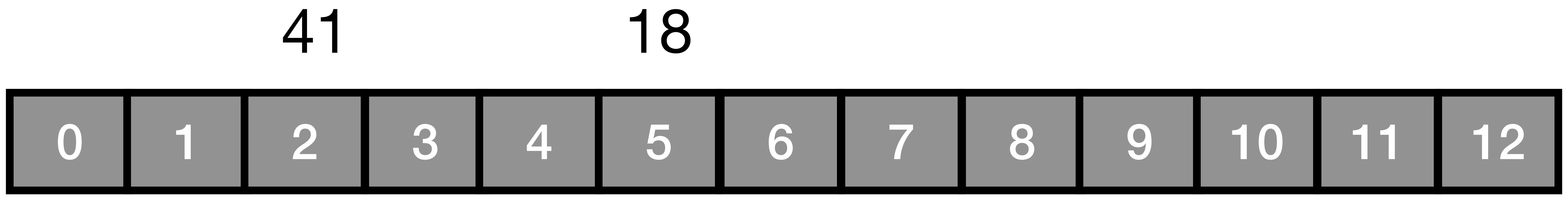
Collision Resolution (2): Linear Probing

- Example: $H(k) = k \mod 13$
- Insert keys: 18, 41, 22, 44, 59, 32, 31
- **Successful search**: Go to the hash value and continue until the location
- **Unsuccessful search**: Go the hash value and continue until the end of the array OR an empty location



Collision Resolution (2): Linear Probing

- Example: $H(k) = k \mod 13$
- Insert keys: 18, 41, 22, 44, 59, 32, 31
- **Successful search**: Go to the hash value and continue until the location
- **Unsuccessful search**: Go the hash value and continue until the end of the array OR an empty location



		41			18							
0	1	2	3	4	5	6	7	8	9	10	11	12

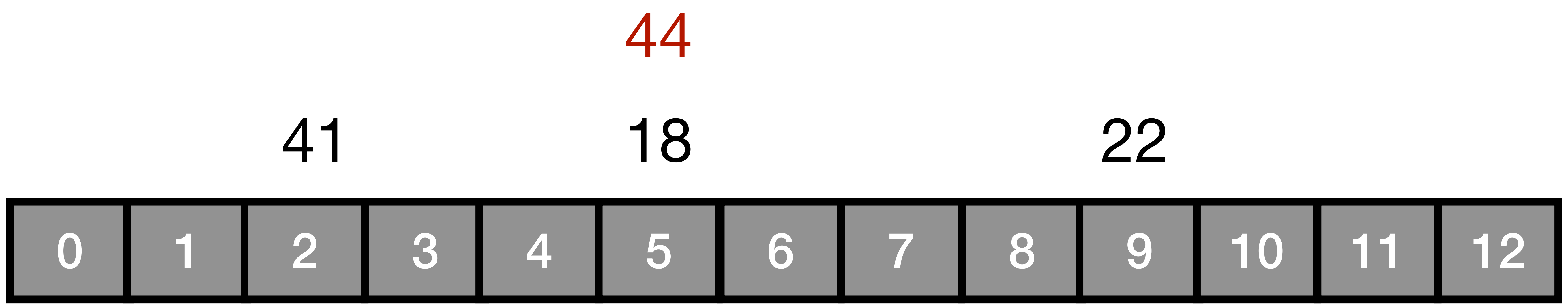
Collision Resolution (2): Linear Probing

- Example: $H(k) = k \mod 13$
- Insert keys: 18, 41, 22, 44, 59, 32, 31
- **Successful search**: Go to the hash value and continue until the location
- **Unsuccessful search**: Go the hash value and continue until the end of the array OR an empty location

		41			18				22			
0	1	2	3	4	5	6	7	8	9	10	11	12

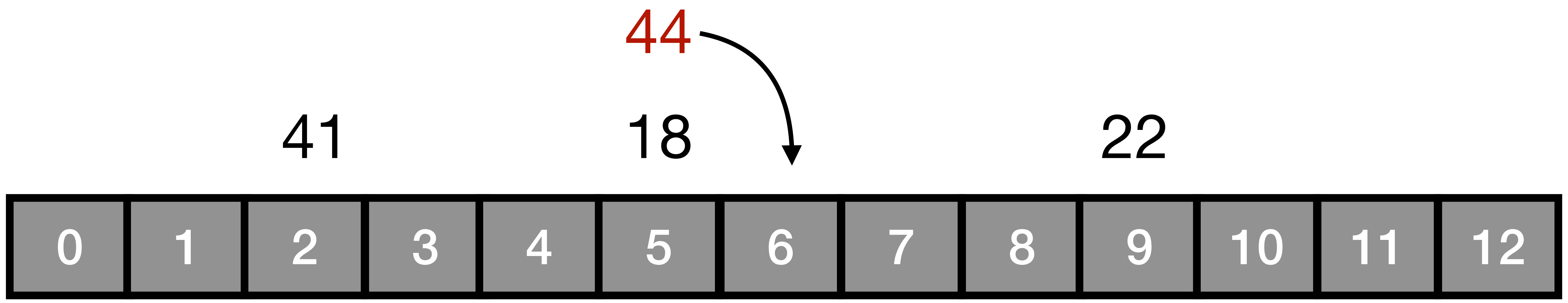
Collision Resolution (2): Linear Probing

- Example: $H(k) = k \mod 13$
- Insert keys: 18, 41, 22, 44, 59, 32, 31
- **Successful search**: Go to the hash value and continue until the location
- **Unsuccessful search**: Go the hash value and continue until the end of the array OR an empty location



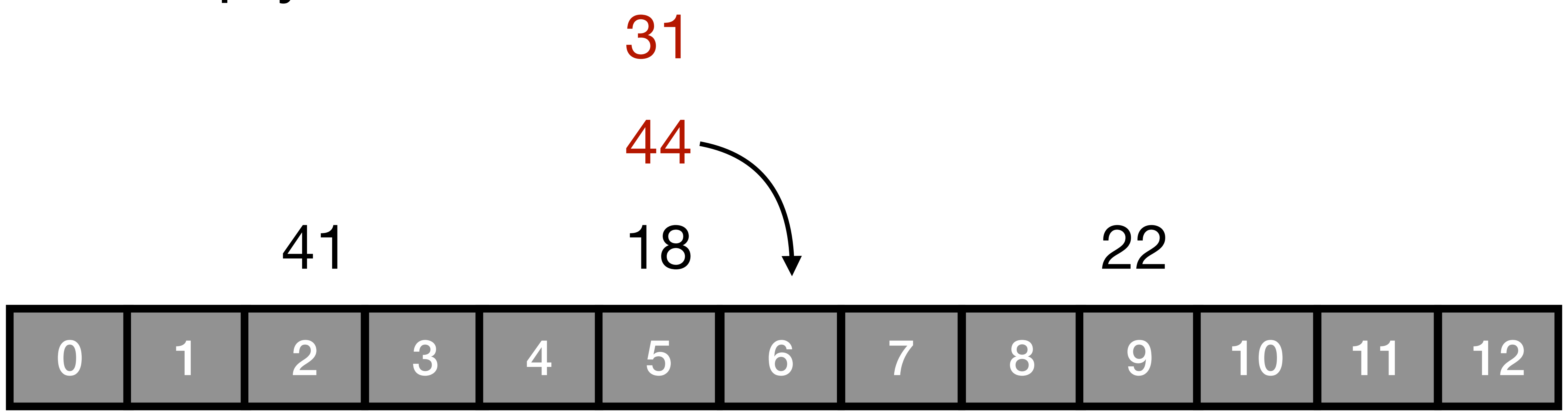
Collision Resolution (2): Linear Probing

- Example: $H(k) = k \mod 13$
- Insert keys: 18, 41, 22, 44, 59, 32, 31
- **Successful search**: Go to the hash value and continue until the location
- **Unsuccessful search**: Go the hash value and continue until the end of the array OR an empty location



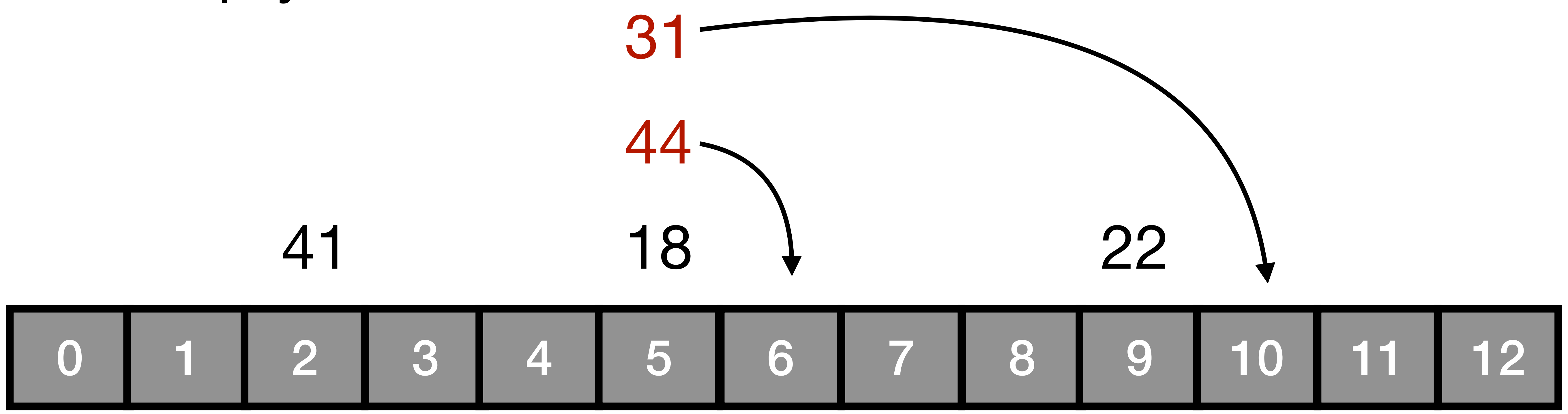
Collision Resolution (2): Linear Probing

- Example: $H(k) = k \mod 13$
- Insert keys: 18, 41, 22, 44, 59, 32, 31
- **Successful search**: Go to the hash value and continue until the location
- **Unsuccessful search**: Go the hash value and continue until the end of the array OR an empty location



Collision Resolution (2): Linear Probing

- Example: $H(k) = k \mod 13$
- Insert keys: 18, 41, 22, 44, 59, 32, 31
- **Successful search**: Go to the hash value and continue until the location
- **Unsuccessful search**: Go the hash value and continue until the end of the array OR an empty location



Collision Resolution (2): Linear Probing

Collision Resolution (2): Linear Probing

- Example: $H(k) = k \mod 13$

Collision Resolution (2): Linear Probing

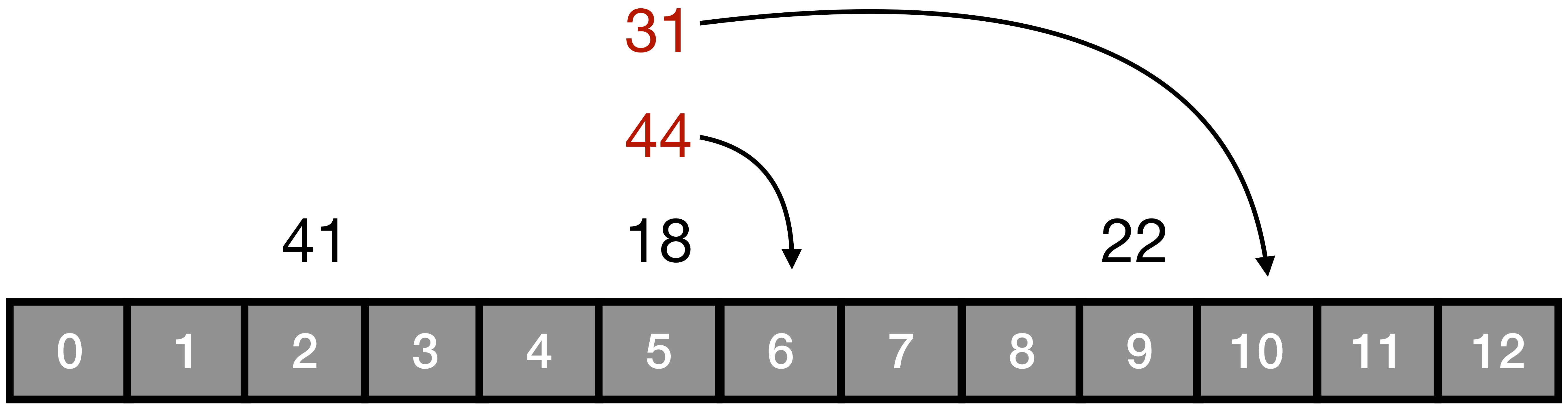
- Example: $H(k) = k \mod 13$
- Keys: 18, 41, 22, 44, 59, 32, 31

Collision Resolution (2): Linear Probing

- Example: $H(k) = k \mod 13$
- Keys: 18, 41, 22, 44, 59, 32, 31
- **Deletion Algorithm**: Go to the hash value and mark it (X)

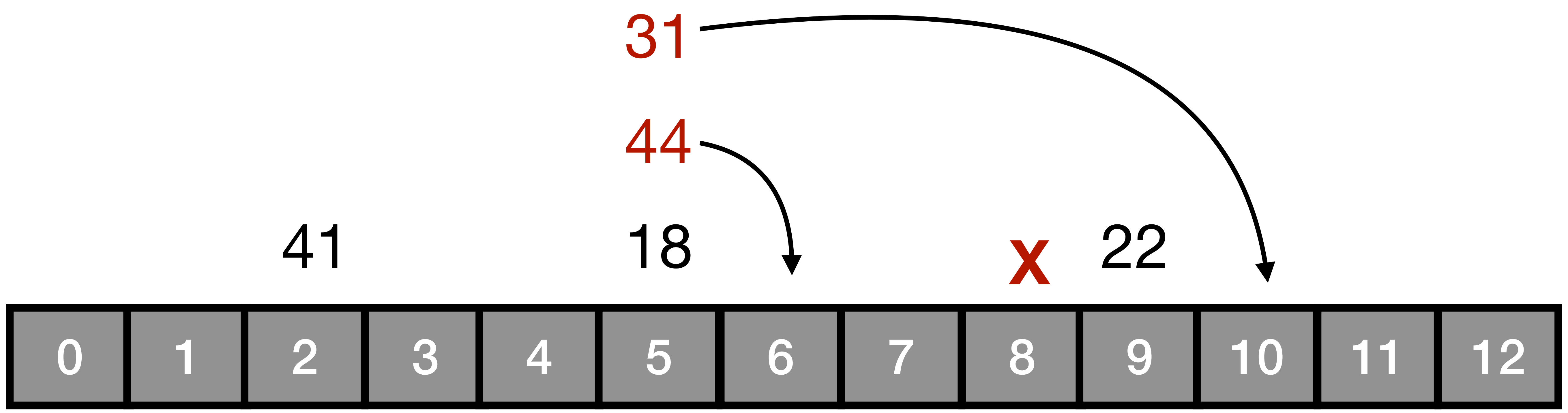
Collision Resolution (2): Linear Probing

- Example: $H(k) = k \mod 13$
- Keys: 18, 41, 22, 44, 59, 32, 31
- **Deletion Algorithm**: Go to the hash value and mark it (**X**)



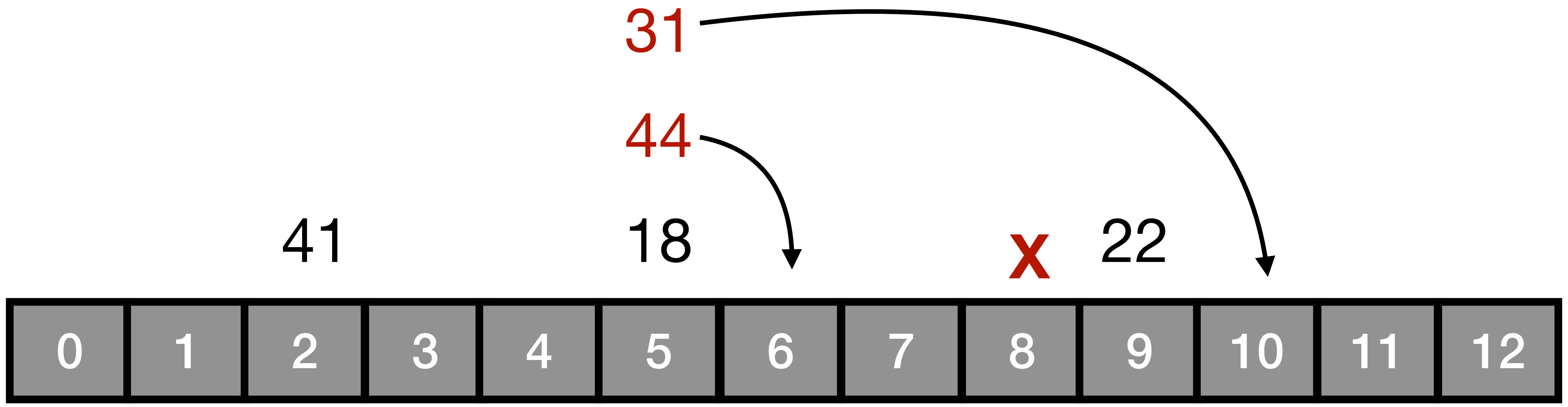
Collision Resolution (2): Linear Probing

- Example: $H(k) = k \mod 13$
- Keys: 18, 41, 22, 44, 59, 32, 31
- **Deletion Algorithm**: Go to the hash value and mark it (**X**)



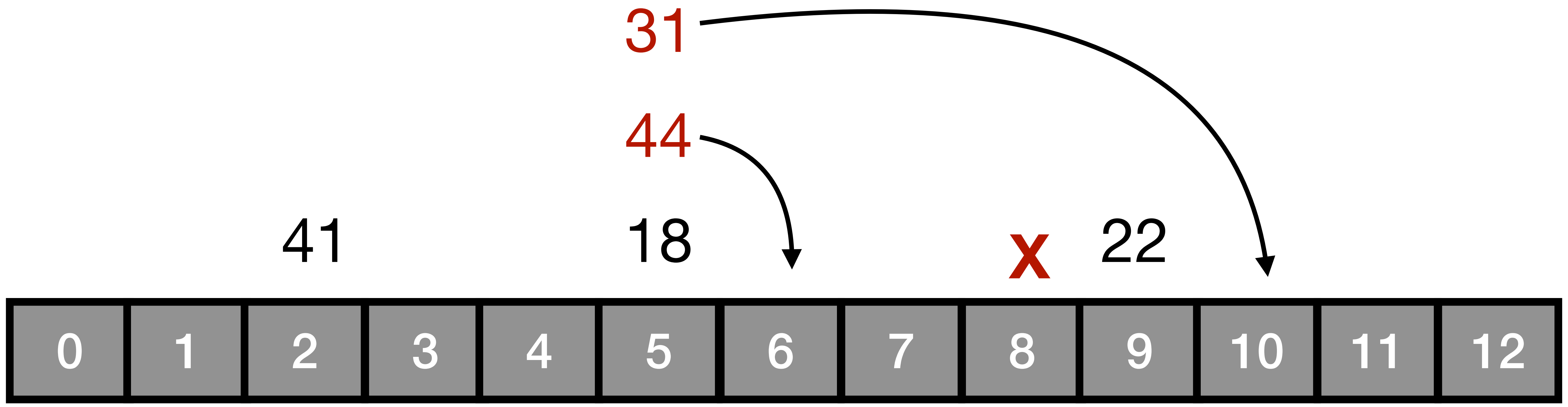
Collision Resolution (2): Linear Probing

- Example: $H(k) = k \mod 13$
- Keys: 18, 41, 22, 44, 59, 32, 31
- **Deletion Algorithm**: Go to the hash value and mark it (**X**)
- Lookup ignores the marked cells; Insert uses the marked cells to place the value



Collision Resolution (2): Linear Probing

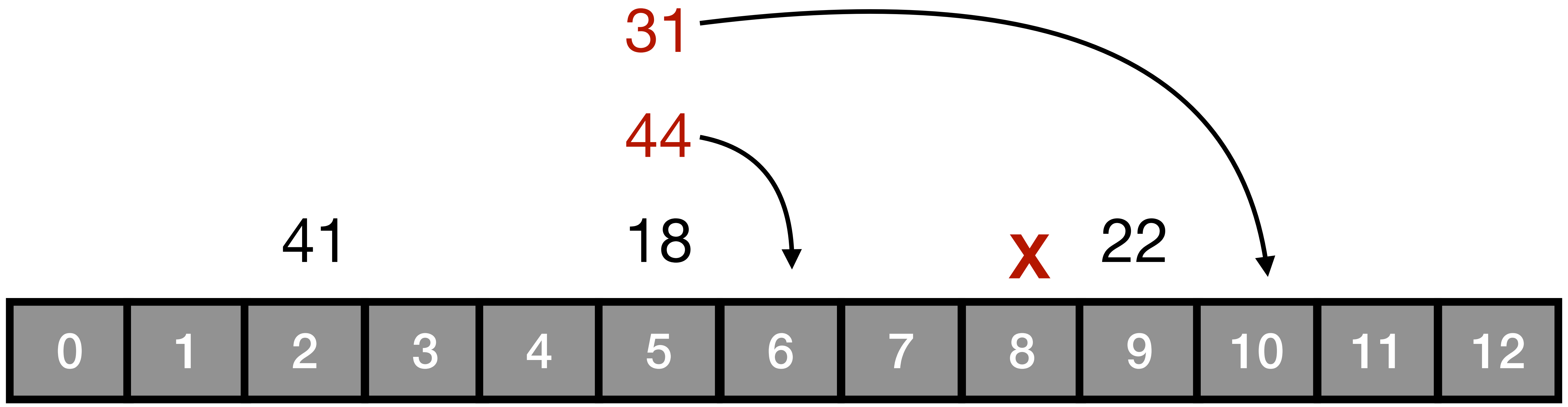
- Example: $H(k) = k \mod 13$
- Keys: 18, 41, 22, 44, 59, 32, 31
- **Deletion Algorithm**: Go to the hash value and mark it (**X**)
- Lookup ignores the marked cells; Insert uses the marked cells to place the value



- **Q: What happens to the efficiency of lookup if there are too many marks?**

Collision Resolution (2): Linear Probing

- Example: $H(k) = k \mod 13$
- Keys: 18, 41, 22, 44, 59, 32, 31
- **Deletion Algorithm**: Go to the hash value and mark it (**X**)
- Lookup ignores the marked cells; Insert uses the marked cells to place the value



- **Q: What happens to the efficiency of lookup if there are too many marks?**
- **Q: What is the solution to the above problem?**

Collision Resolution (2): Linear Probing

Collision Resolution (2): Linear Probing

- Example: $H(k) = k \mod 16$

Collision Resolution (2): Linear Probing

- Example: $H(k) = k \mod 16$
- **HT** has 16 slots

Collision Resolution (2): Linear Probing

- Example: $H(k) = k \mod 16$
- **HT** has 16 slots
- Assume the values 0 to 15 are represented as Bit Vectors (ie. 0000 to 1111)

Collision Resolution (2): Linear Probing

- Example: $H(k) = k \mod 16$
- **HT** has 16 slots
- Assume the values 0 to 15 are represented as Bit Vectors (ie. 0000 to 1111)
- **What problem do you foresee?**

Collision Resolution (2): Linear Probing

- Example: $H(k) = k \mod 16$
- **HT** has 16 slots
- Assume the values 0 to 15 are represented as Bit Vectors (ie. 0000 to 1111)
- **What problem do you foresee?**
- The value returned by **H** is solely dependent on the least significant 4 bits of the key

Collision Resolution (2): Linear Probing

- Example: $H(k) = k \mod 16$
- **HT** has 16 slots
- Assume the values 0 to 15 are represented as Bit Vectors (ie. 0000 to 1111)
- **What problem do you foresee?**
- The value returned by **H** is solely dependent on the least significant 4 bits of the key
    - Therefore the results will be poorly distributed!

Collision Resolution (2): Linear Probing

- Example: $H(k) = k \mod 16$
- **HT** has 16 slots
- Assume the values 0 to 15 are represented as Bit Vectors (ie. 0000 to 1111)
- **What problem do you foresee?**
- The value returned by **H** is solely dependent on the least significant 4 bits of the key
    - Therefore the results will be poorly distributed!
- **EXERCISE: Try with different table sizes, different levels of loading (the number of keys inserted), and different input distributions.**

Collision Resolution (3): Double Hashing

Collision Resolution (3): Double Hashing

- Use two hash functions : H_1, H_2

Collision Resolution (3): Double Hashing

- Use two hash functions : H_1, H_2
- $H_1(k)$: Is the position in the table where we first check for key **k**

Collision Resolution (3): Double Hashing

- Use two hash functions : H_1, H_2
- $H_1(k)$: Is the position in the table where we first check for key **k**
- $H_2(k)$: Determines the offset when we search for **k**

Collision Resolution (3): Double Hashing

- Use two hash functions : H_1, H_2
- $H_1(k)$: Is the position in the table where we first check for key **k**
- $H_2(k)$: Determines the offset when we search for **k**
- In Linear Probing $H_2(k)$ is 1

Collision Resolution (3): Double Hashing

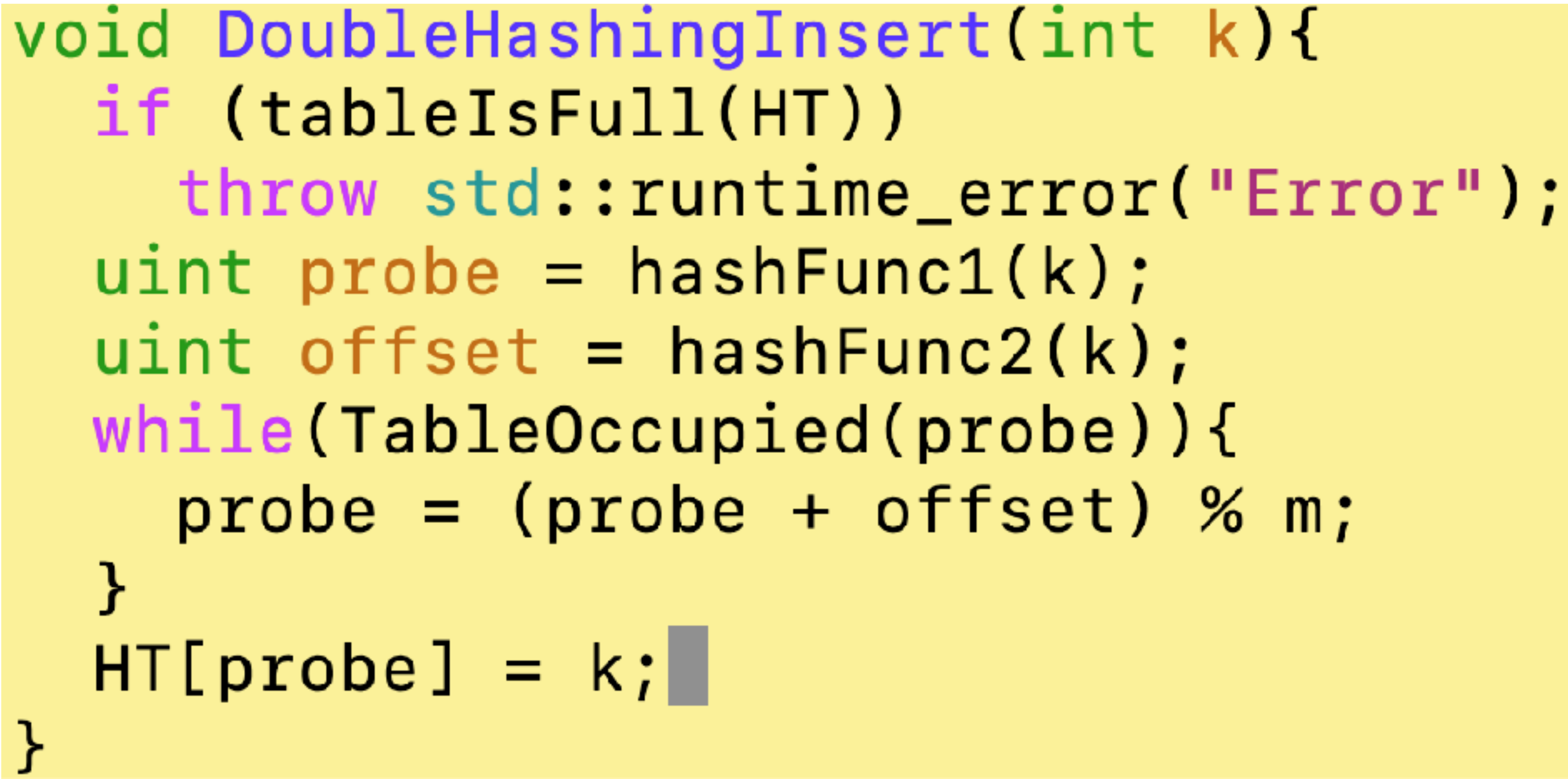
- Use two hash functions : H_1, H_2
- $H_1(k)$: Is the position in the table where we first check for key **k**
- $H_2(k)$: Determines the offset when we search for **k**
- In Linear Probing $H_2(k)$ is 1
- **Insert Algorithm:**

Collision Resolution (3): Double Hashing

- Use two hash functions : H_1, H_2
- $H_1(k)$: Is the position in the table where we first check for key **k**
- $H_2(k)$: Determines the offset when we search for **k**
- In Linear Probing $H_2(k)$ is 1
- **Insert Algorithm:**
- Distributes keys more uniformly

Collision Resolution (3): Double Hashing

- Use two hash functions : H_1, H_2
- $H_1(k)$: Is the position in the table where we first check for key **k**
- $H_2(k)$: Determines the offset when we search for **k**
- In Linear Probing $H_2(k)$ is 1
- **Insert Algorithm:**
- Distributes keys more uniformly



```
void DoubleHashingInsert(int k){
  if (tableIsFull(HT))
    throw std::runtime_error("Error");
  uint probe = hashFunc1(k);
  uint offset = hashFunc2(k);
  while(TableOccupied(probe)){
    probe = (probe + offset) % m;
  }
  HT[probe] = k;
}
```

Collision Resolution (3): Double Hashing

Collision Resolution (3): Double Hashing

- $H_1(k)$: k mod 13
- $H_2(k)$: 8 - (k mod 8)
- Insert keys: 18, 41, 22, 44, 59, 32, 31

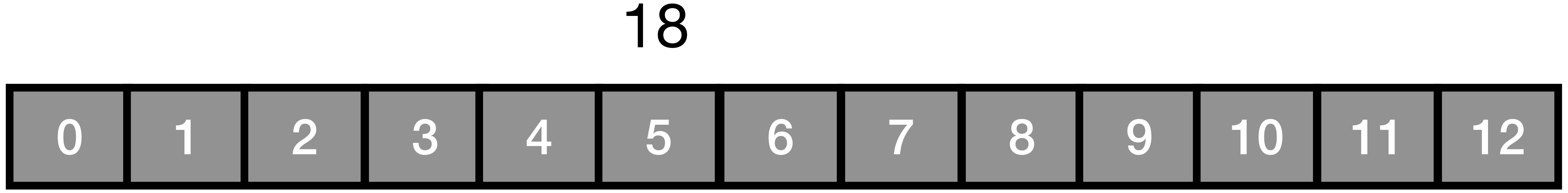
Collision Resolution (3): Double Hashing

- $H_1(k)$: k mod 13
- $H_2(k)$: 8 - (k mod 8)
- Insert keys: 18, 41, 22, 44, 59, 32, 31

0	1	2	3	4	5	6	7	8	9	10	11	12

Collision Resolution (3): Double Hashing

- $H_1(k)$: k mod 13
- $H_2(k)$: 8 - (k mod 8)
- Insert keys: 18, 41, 22, 44, 59, 32, 31



Collision Resolution (3): Double Hashing

- $H_1(k)$: k mod 13
- $H_2(k)$: 8 - (k mod 8)
- Insert keys: 18, 41, 22, 44, 59, 32, 31

		41			18							
0	1	2	3	4	5	6	7	8	9	10	11	12

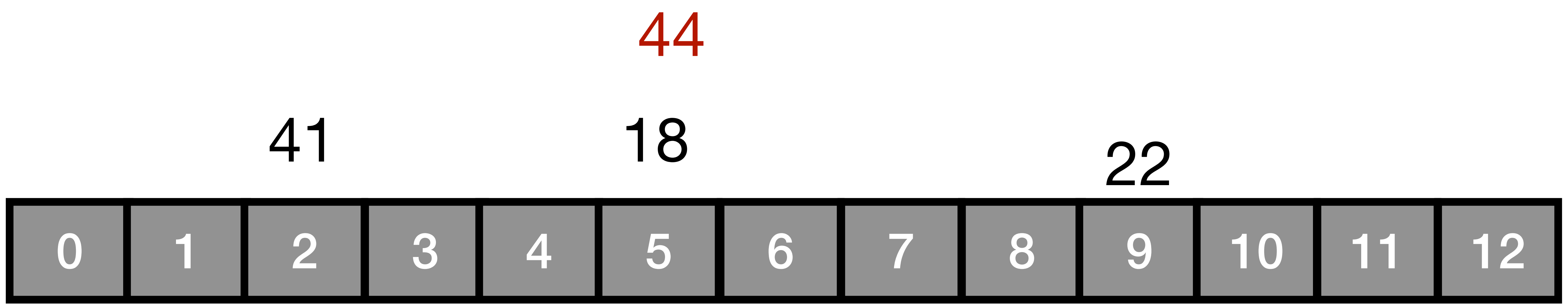
Collision Resolution (3): Double Hashing

- $H_1(k)$: k mod 13
- $H_2(k)$: 8 - (k mod 8)
- Insert keys: 18, 41, 22, 44, 59, 32, 31

		41			18				22			
0	1	2	3	4	5	6	7	8	9	10	11	12

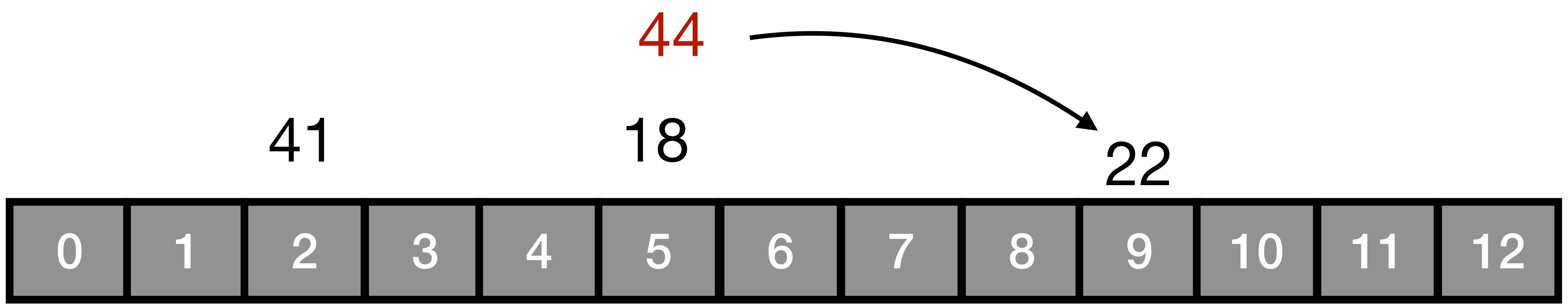
Collision Resolution (3): Double Hashing

- $H_1(k)$: k mod 13
- $H_2(k)$: 8 - (k mod 8)
- Insert keys: 18, 41, 22, 44, 59, 32, 31



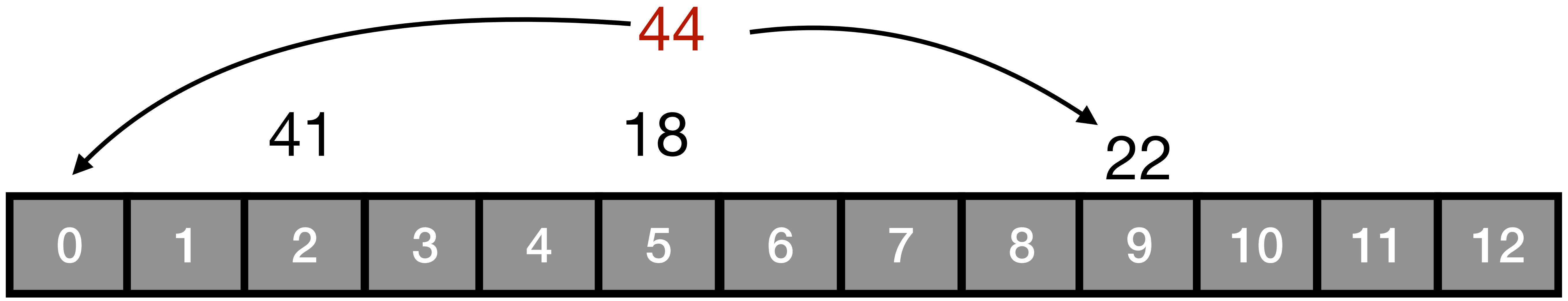
Collision Resolution (3): Double Hashing

- $H_1(k)$: k mod 13
- $H_2(k)$: 8 - (k mod 8)
- Insert keys: 18, 41, 22, 44, 59, 32, 31



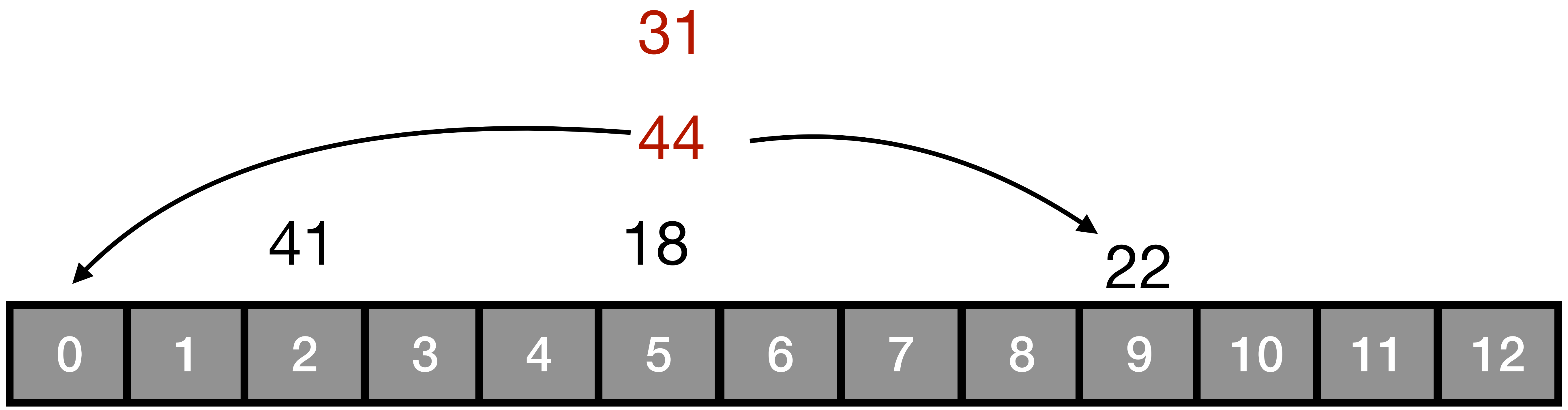
Collision Resolution (3): Double Hashing

- $H_1(k)$: k mod 13
- $H_2(k)$: 8 - (k mod 8)
- Insert keys: 18, 41, 22, 44, 59, 32, 31

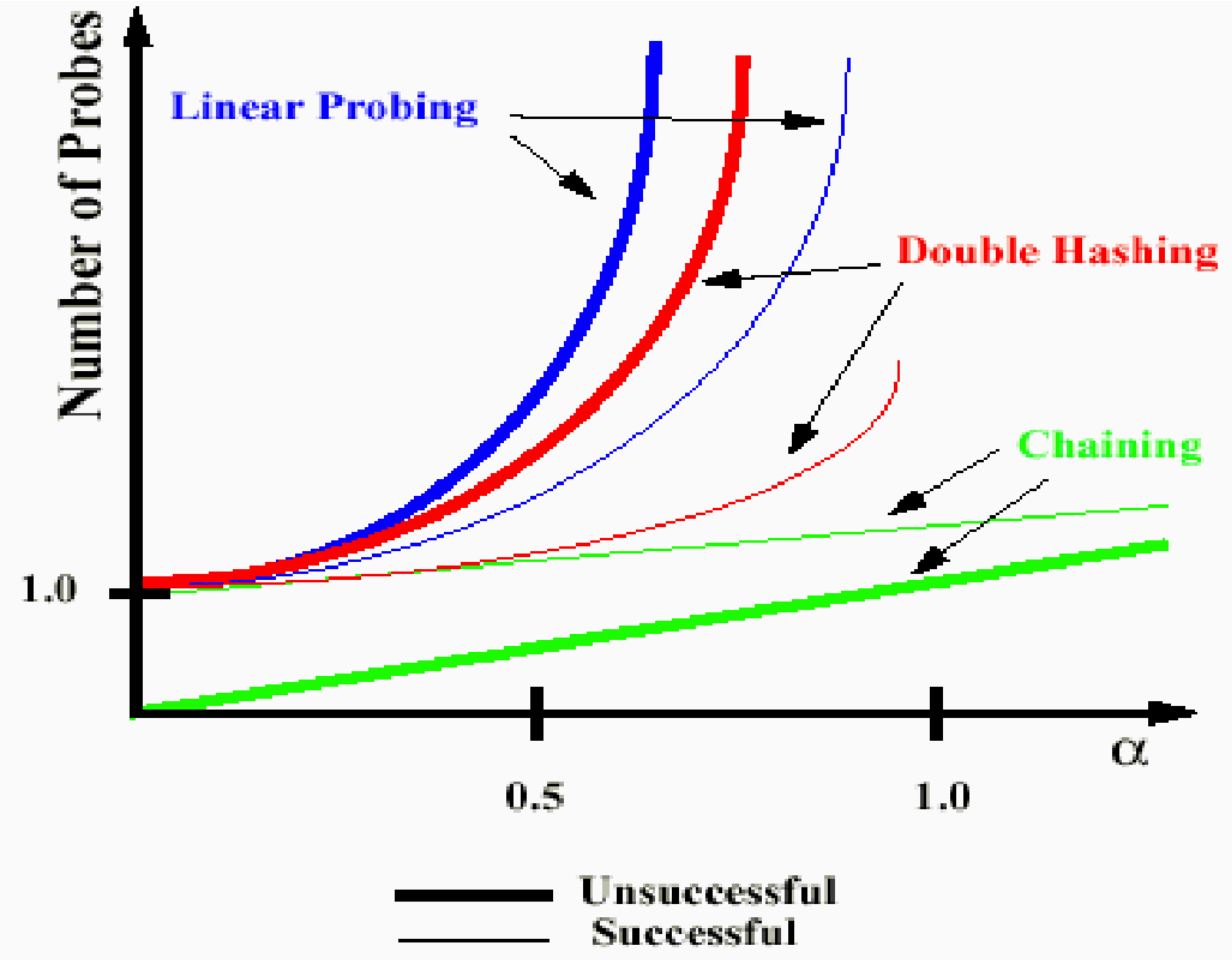


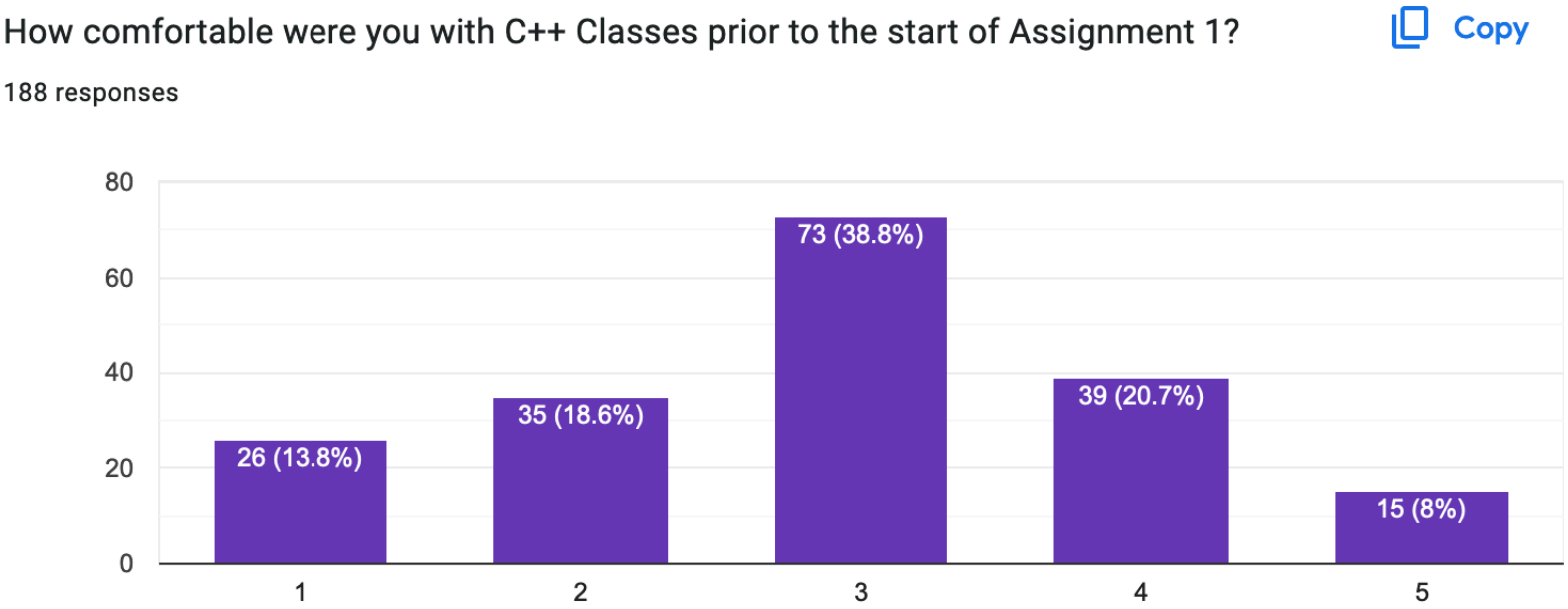
Collision Resolution (3): Double Hashing

- $H_1(k)$: k mod 13
- $H_2(k)$: 8 - (k mod 8)
- Insert keys: 18, 41, 22, 44, 59, 32, 31



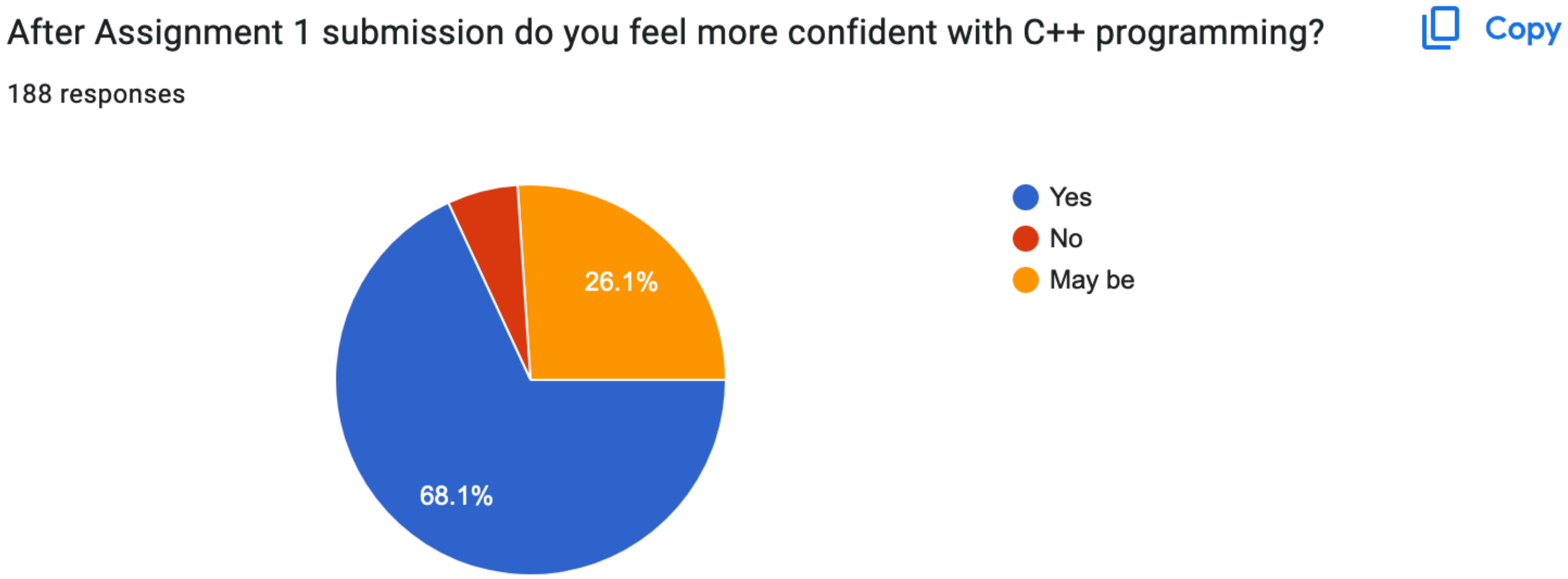
Expected Number of Probes





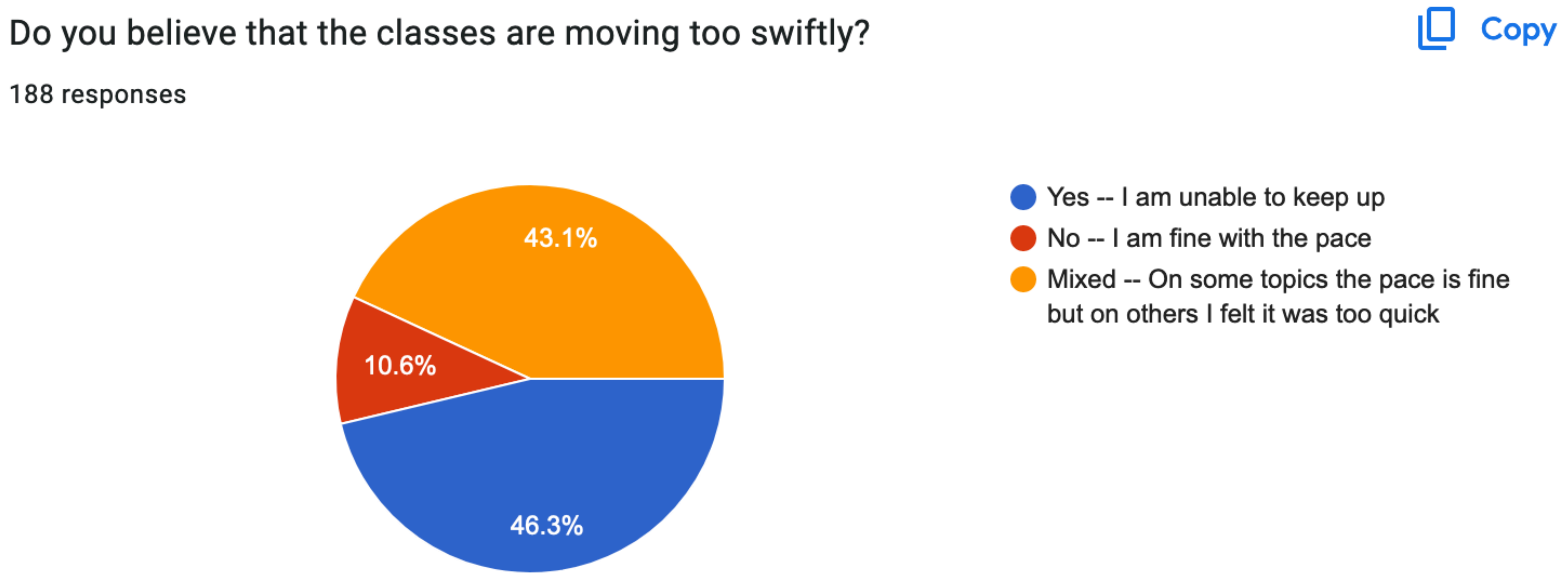
How comfortable were you with C++ Classes prior to the start of Assignment 1?

188 responses



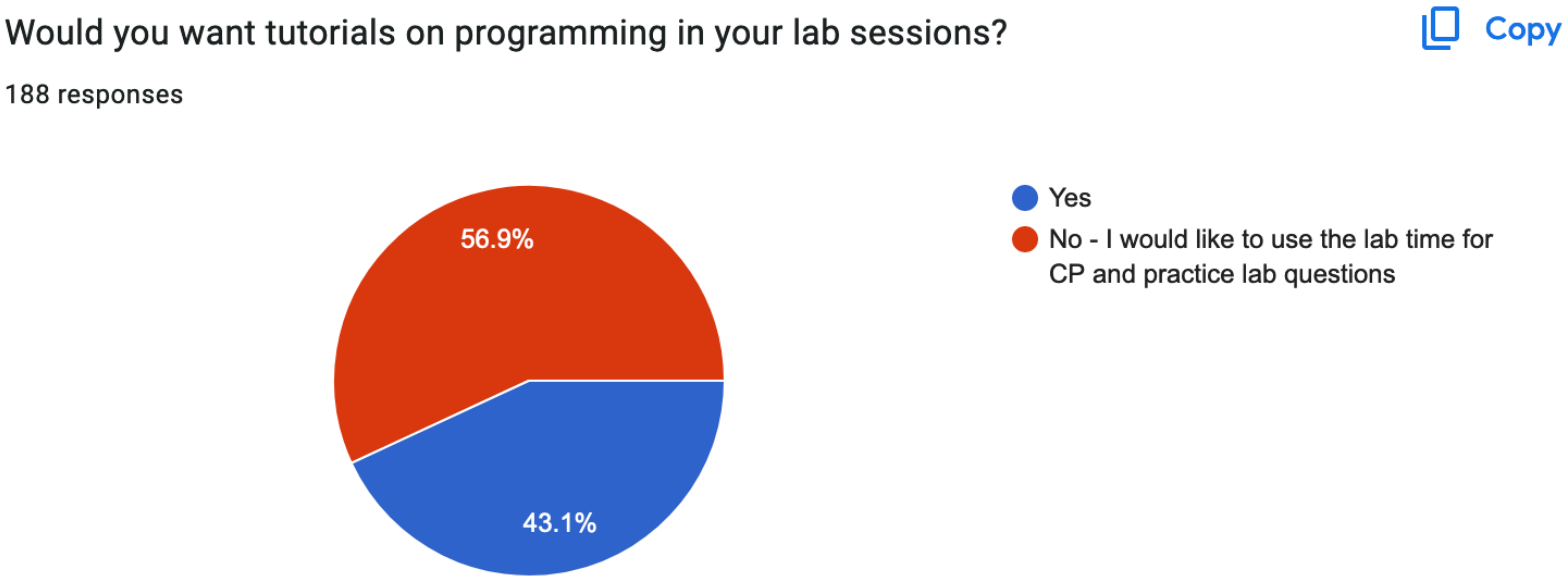
After Assignment 1 submission do you feel more confident with C++ programming?

188 responses



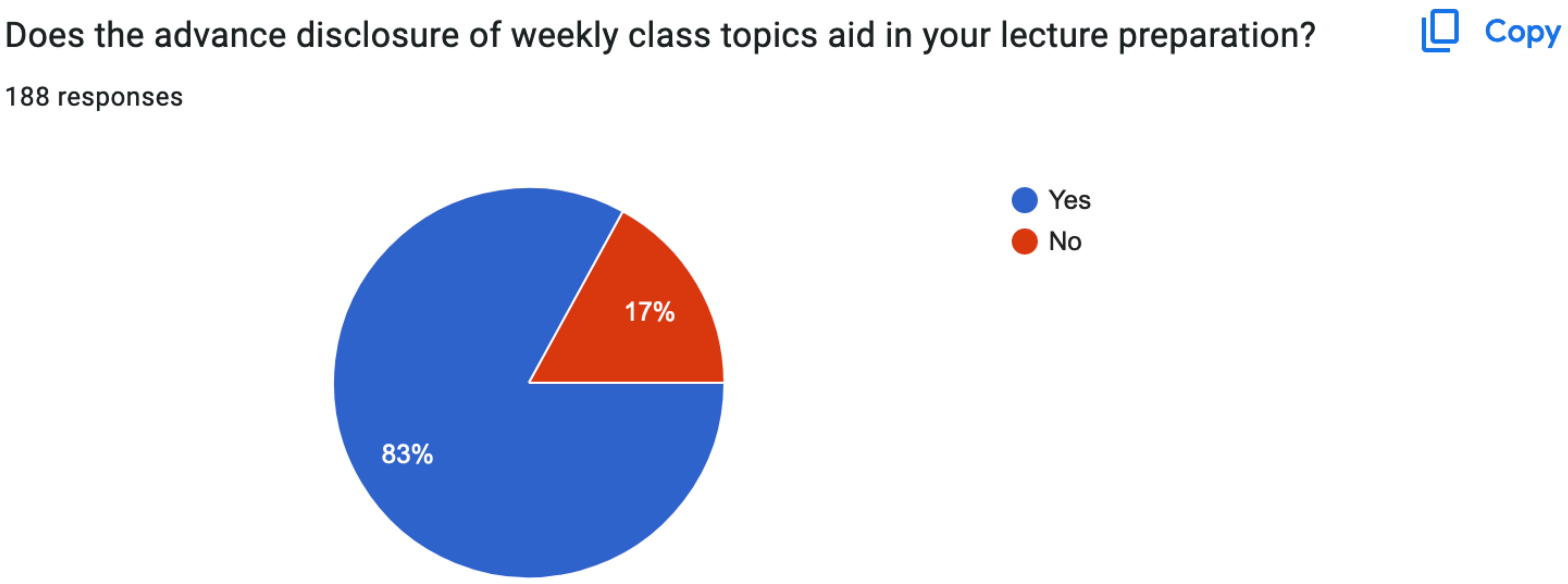
Do you believe that the classes are moving too swiftly?

188 responses



Would you want tutorials on programming in your lab sessions?

188 responses



Does the advance disclosure of weekly class topics aid in your lecture preparation?

188 responses

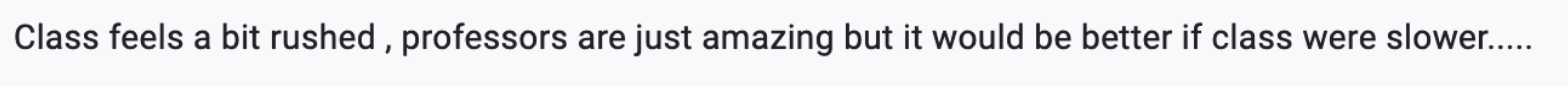
RECOMMENDATIONS

RECOMMENDATIONS

Please upload slide simultaneously l

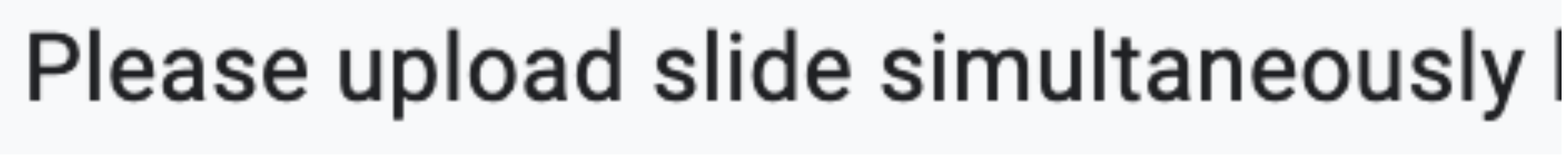
RECOMMENDATIONS

Please upload slide simultaneously

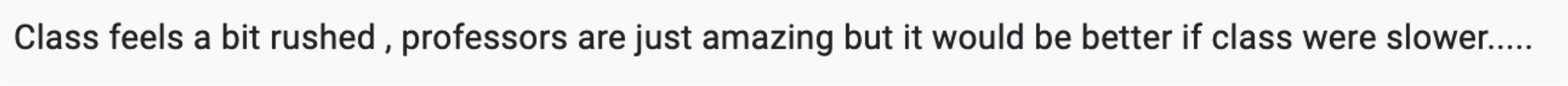


Class feels a bit rushed , professors are just amazing but it would be better if class were slower.....

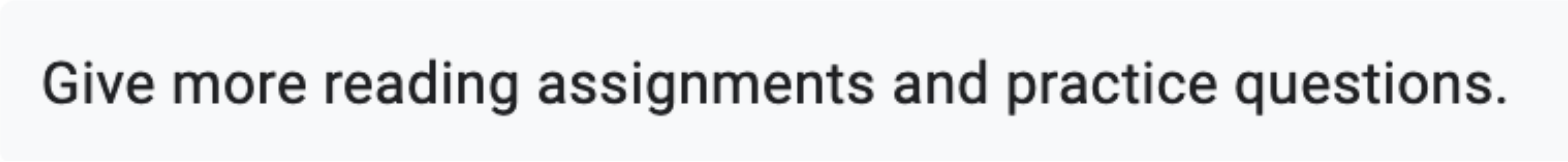
RECOMMENDATIONS



Please upload slide simultaneously

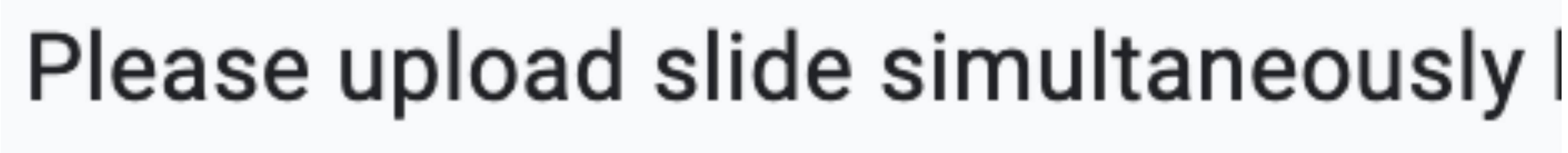


Class feels a bit rushed , professors are just amazing but it would be better if class were slower.....

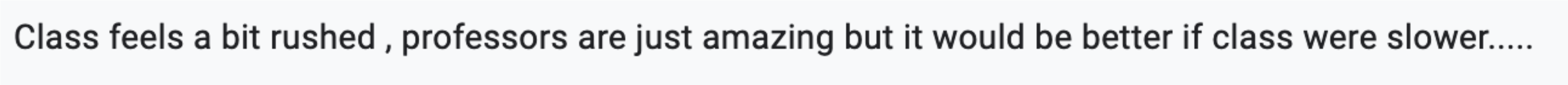


Give more reading assignments and practice questions.

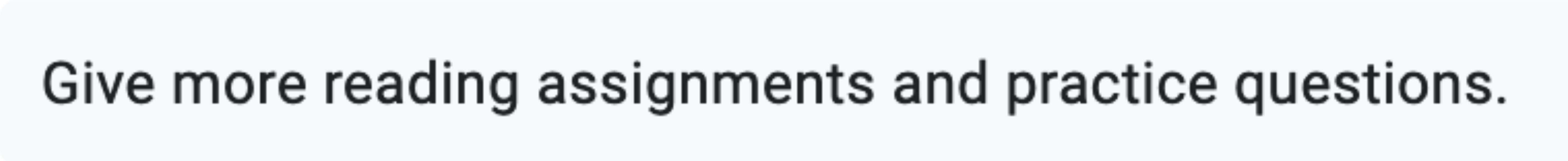
RECOMMENDATIONS



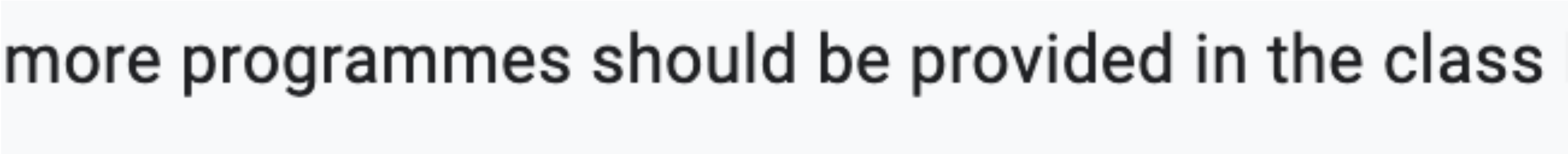
Please upload slide simultaneously



Class feels a bit rushed , professors are just amazing but it would be better if class were slower.....



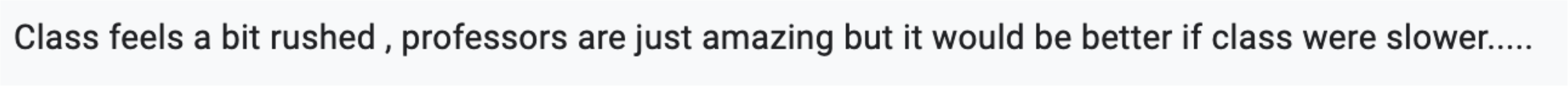
Give more reading assignments and practice questions.



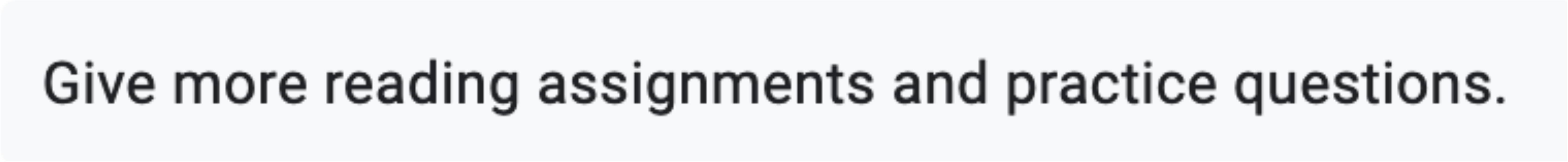
more programmes should be provided in the class

RECOMMENDATIONS

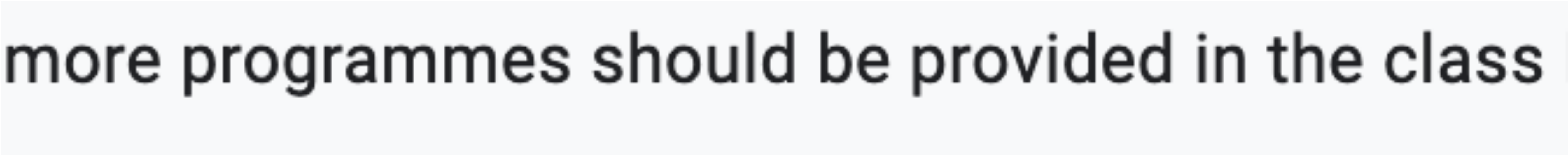
Please upload slide simultaneously



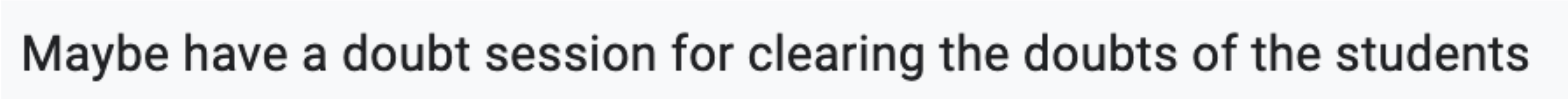
Class feels a bit rushed , professors are just amazing but it would be better if class were slower.....



Give more reading assignments and practice questions.

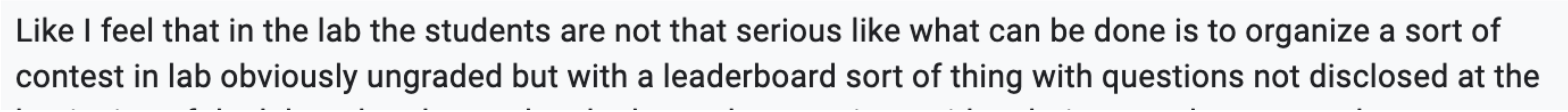


more programmes should be provided in the class

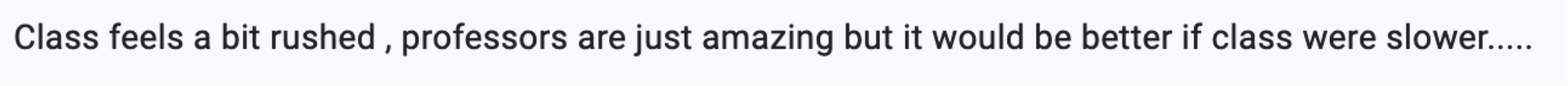


Maybe have a doubt session for clearing the doubts of the students

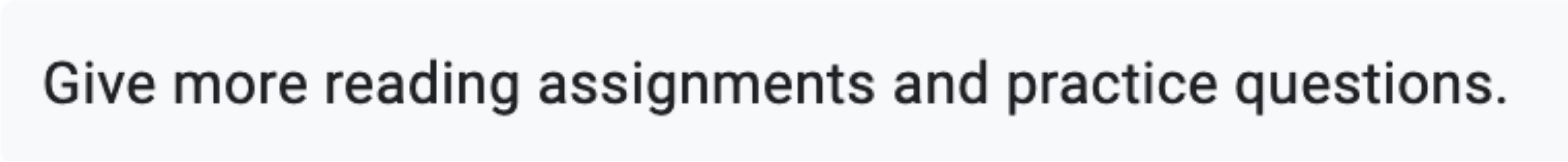
RECOMMENDATIONS



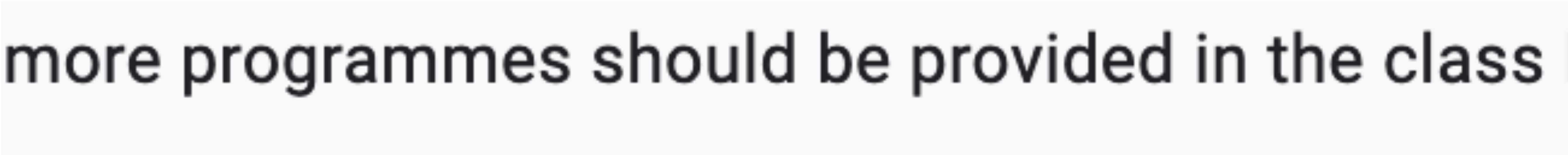
Like I feel that in the lab the students are not that serious like what can be done is to organize a sort of contest in lab obviously ungraded but with a leaderboard sort of thing with questions not disclosed at the



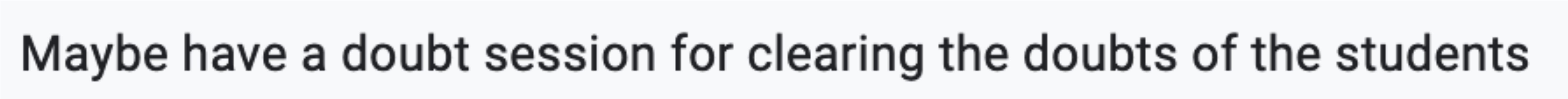
Class feels a bit rushed , professors are just amazing but it would be better if class were slower.....



Give more reading assignments and practice questions.



more programmes should be provided in the class



Maybe have a doubt session for clearing the doubts of the students

RECOMMENDATIONS

Certainly, it seems that grasping data structures and algorithms, or indeed any subject, hinges largely on one's personal commitment. Assuming that attending a mere three-hour lecture per week would confer a profound understanding of their intricacies is rather unrealistic. In my view, a more systematic approach to teaching data structures and algorithms would require around nine hours of weekly instruction, delving into sufficient detail to comprehensively elucidate all underlying concepts while effectively fostering motivation.

However, given the constraints imposed by institutional regulations, realizing such an intensive teaching schedule is unfeasible. Thus, in my estimation, a more pragmatic approach involves proactively acquainting oneself with the subject matter through self-guided study from recommended resources prior to class sessions. These interactive sessions could then be optimized for addressing queries and igniting enthusiasm through the exploration of novel topics.

Having shifted my perspective away from harboring overly ambitious expectations from class hours and embracing the aforementioned strategy, I have found myself considerably content with the pace and depth of my learning journey.