UCSC Silicon Valley Extension

Advanced C Programming

Hashing

Instructor: Radhika Grover

Overview

- Hashing
  - Hash functions
  - Open hashing
  - Closed hashing
  - Rehashing
  - Extendible hashing
  - Implementation and applications

Sequential search vs hashing

- Example: Find the phone number of a person given his or her name.
    - Search an array that stores the phonebook until a matching entry containing the name of the person is found.
- Sequential search
    - Worst-case performance is $O(n)$ for an array of size n.
- Hashing
    - Performance of an ideal hash table: $O(1)$

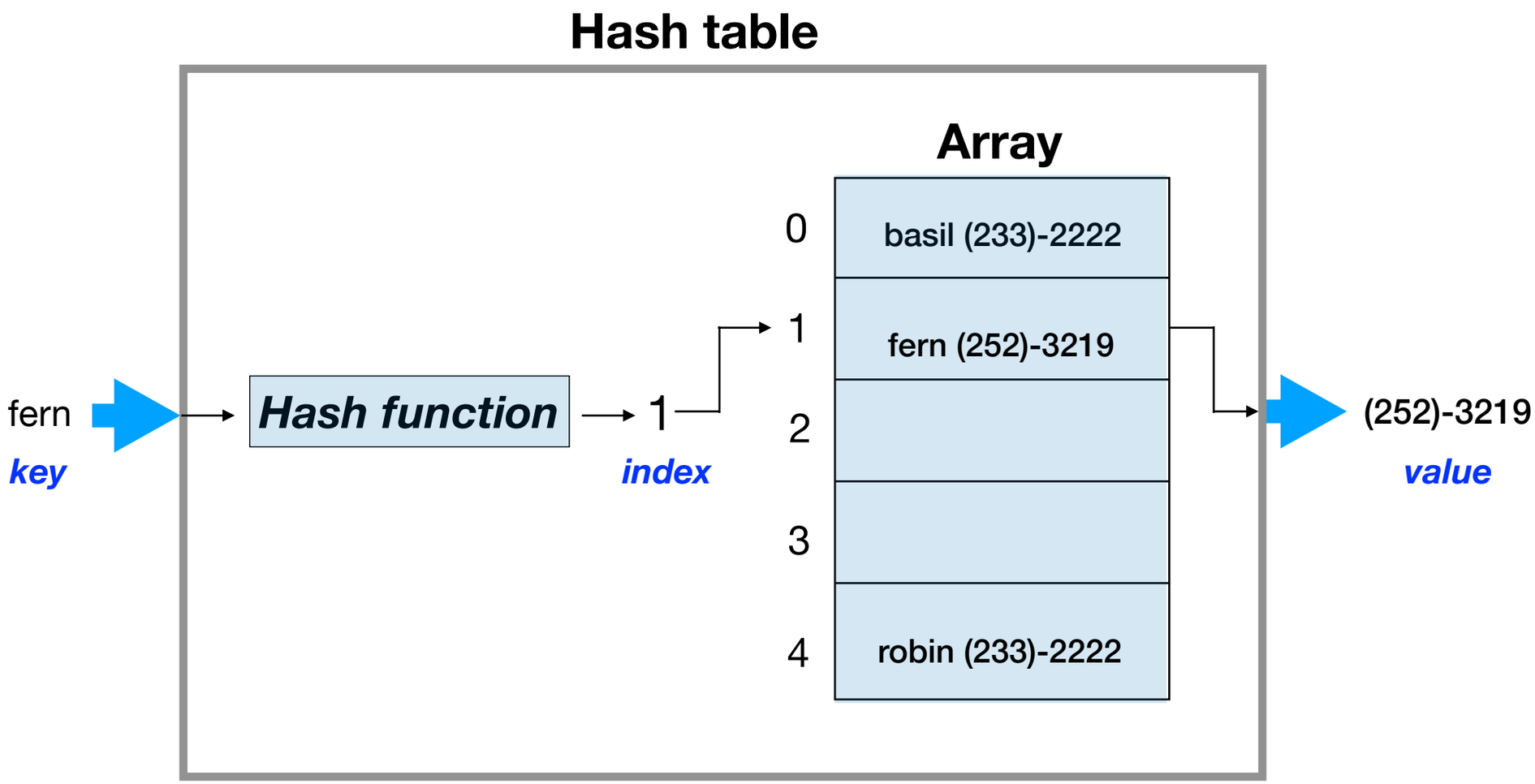
Hash table

- Contains:

    1. Array
        - Stores data as <*key*, *value*> pairs
    2. Hashing function
        - Converts the *key* to the array *index* with corresponding *value*
        - Decreases the time needed to find an item

Example: Find a phone number

- Find the phone number given the name using a hash table



Operations on hash tables

- **Insert(key, value)**: Add a <*key*, *value*> pair to table

- **Find(key)**: Return the *value* for a given *key*

- **Delete(key)**: Remove a <*key*, *value*> pair from table

Hash function properties

- Takes *key* as input and returns an *index* into the array.
- Used to insert <*key*, *value*> pairs into array
- Used to retrieve the value for a given key.
- Desirable properties:
  - Distributes keys evenly through the table avoiding collisions (multiple keys map to the same index).
  - Easy to compute

Hash function for integer keys

- Integer keys:
  - Returns key % SIZE, where SIZE is array size
  - SIZE is usually prime to reduce *collisions*

Hash function for non integer keys

- Multiplies the ASCII values of each character $key[i]$ in the key by a weight p^i, where p is a base and i is the position of the character and adds them together.

hashing function $h = (key[\text{i}] * p^i + key[\text{i}-1] * p^{i-1} + \ldots + key[0] * p^0)$ % SIZE

- For example, the ASCII values of f, e, r, and n are 102, 101, 114, and 110 respectively.
    - Using a base of 127, we get the following hashing function for the key *fern*:

$$h = (102 * 127^3 + 101 * 127^2 + 114 * 127^1 + 110 * 127^0) \text{ \% SIZE}$$

Horner's rule and modular arithmetic

The term $key[3] * p^3 + key[2] * p^2 + key[1] * p^1 + key[0] * p^0$ can be computed more efficiently (with fewer multiplications) using Horner's rule as:

$$((key[3] * p + key[2]) * p + key[1]) * p + key[0] * p^0$$

- But the resulting value may be too large to store in a long or long long - can use a property of modular arithmetic for this problem.

Modular arithmetic

- Modulo property for addition:

$$(A + B) \% N = (A \% N + B \% N) \% N$$

- Example: Using this property, the hashing function

$$h = (((key[3] * p + key[2]) * p + key[1]) * p + key[0] * p^0) \% SIZE$$

can be computed by taking modulo after each addition as:

$h = 0$

$h = (h * p + key[3]) \% SIZE$

$h = (h * p + key[2]) \% SIZE$

$h = (h * p + key[1]) \% SIZE$

$h = (h * p + key[0]) \% SIZE$ /* *Final result* */

Collision

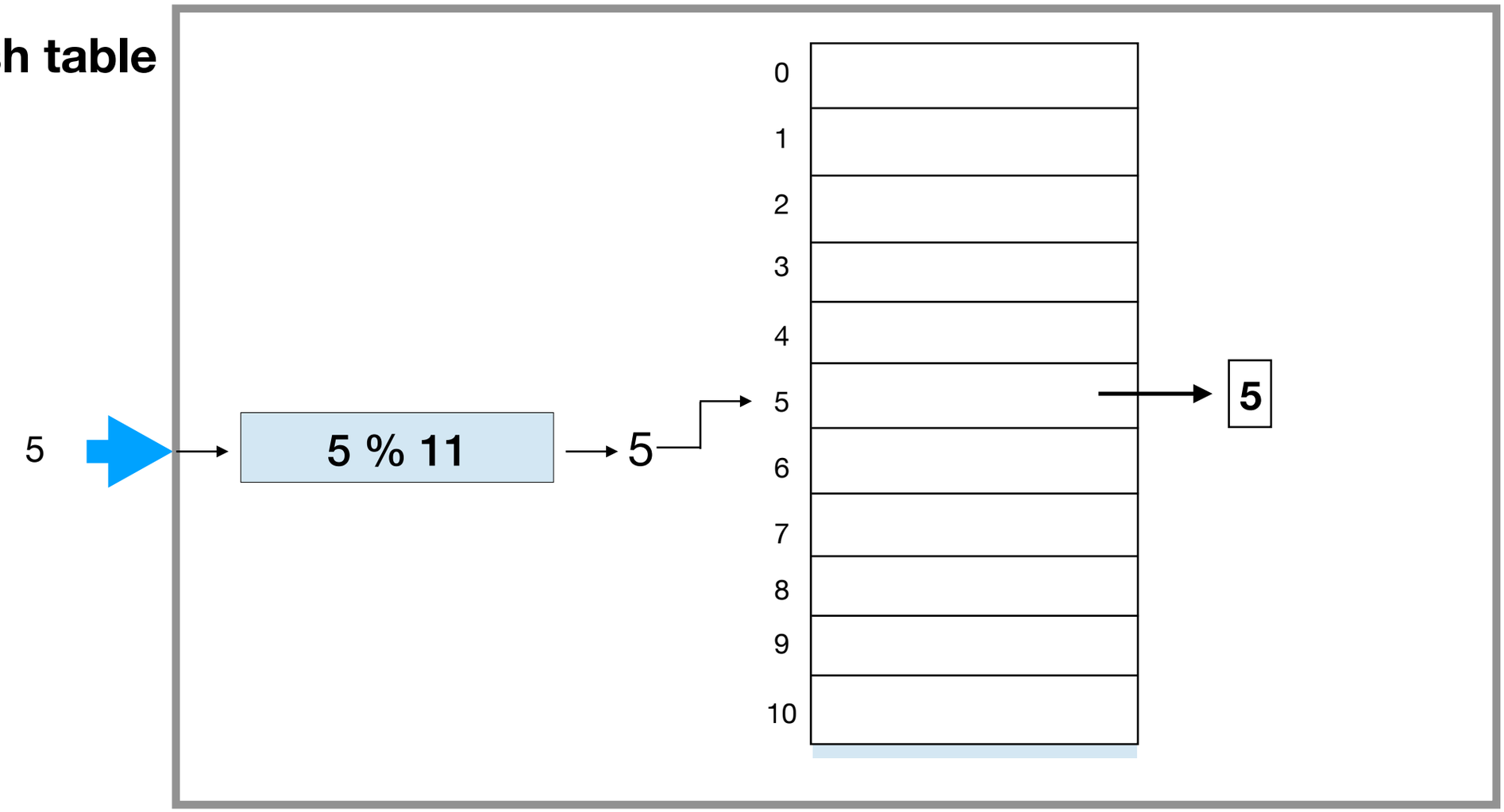
- Two different keys map to the same location (index) in the array.
- Example:
  - Insert keys 25 and 36 into a table of size 11:
    - 25 % 11 = 3 and
    - 36 % 11 = 3

Techniques to resolve collisions

- Open hashing (also known as separate chaining)
  - Keep a linked list of all elements that hash to the same index in the array
- Closed hashing
  - Try alternate indices in the array until an empty one is found.

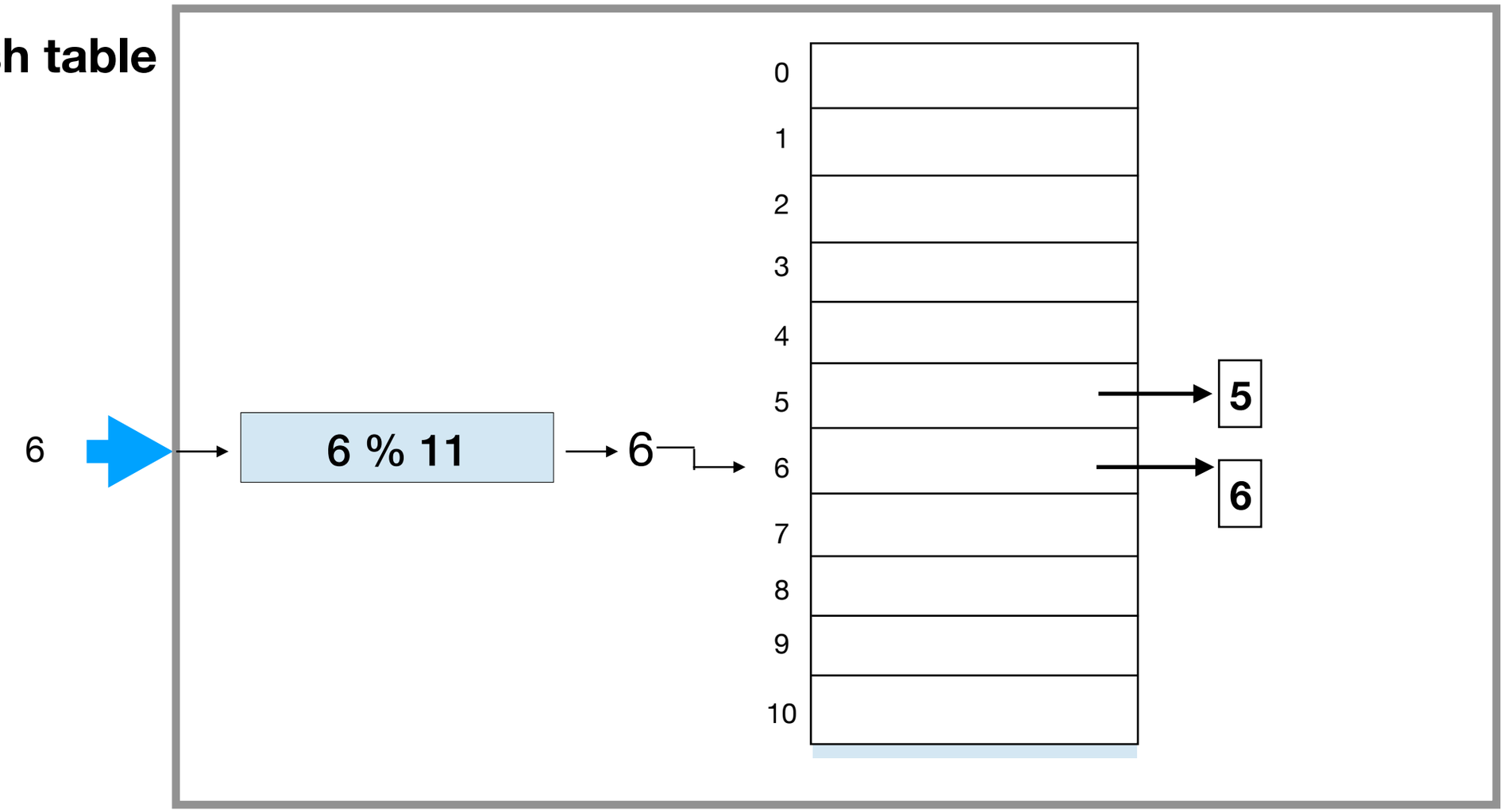
Open hashing example

- Insert the keys **5**, 6, 8, 49, 50, 55, 100 into a hash table of size 11.



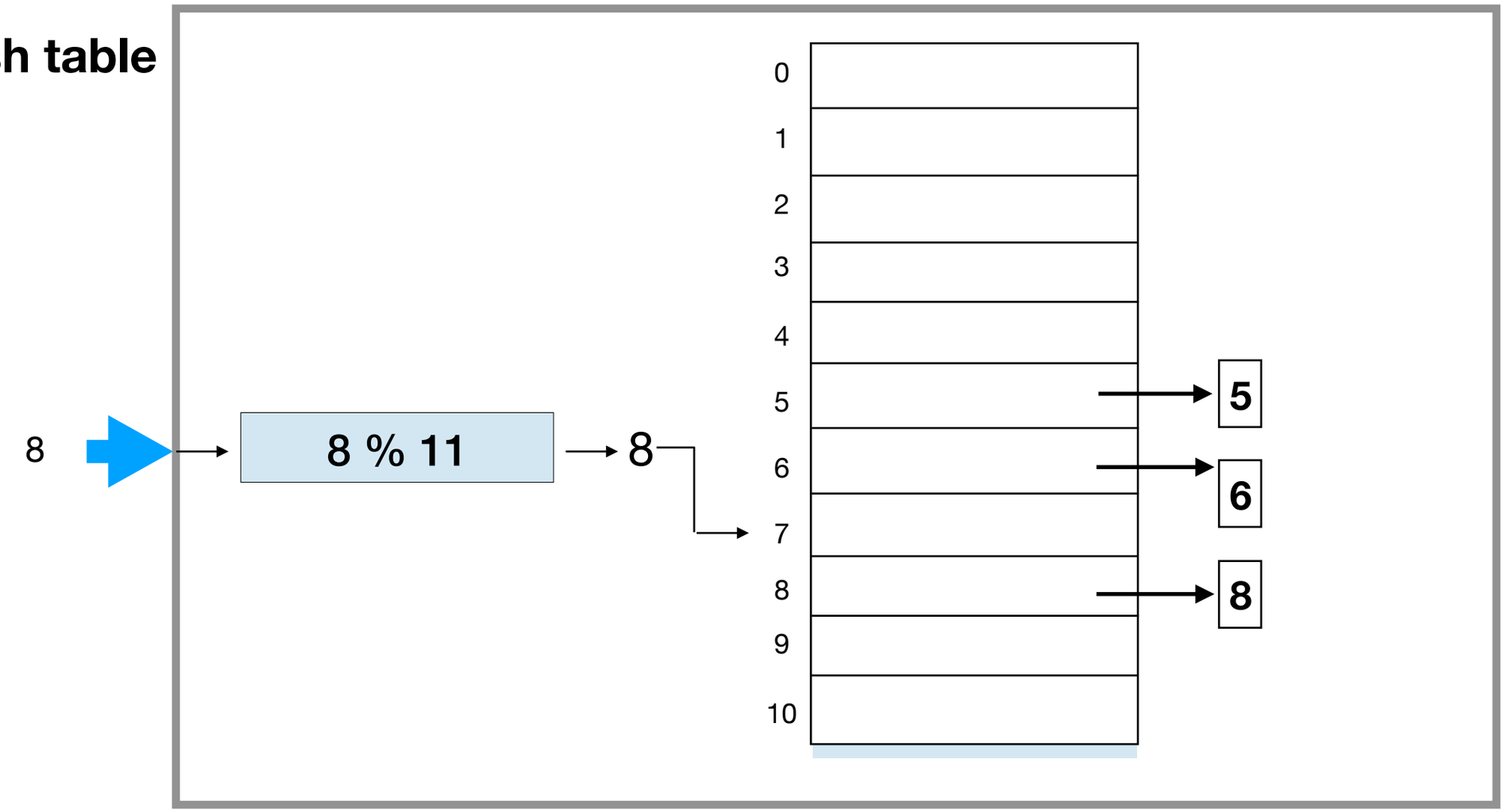
Open hashing example continued

- Insert the keys 5, **6**, 8, 49, 50, 55, 100 into a hash table of size 11.



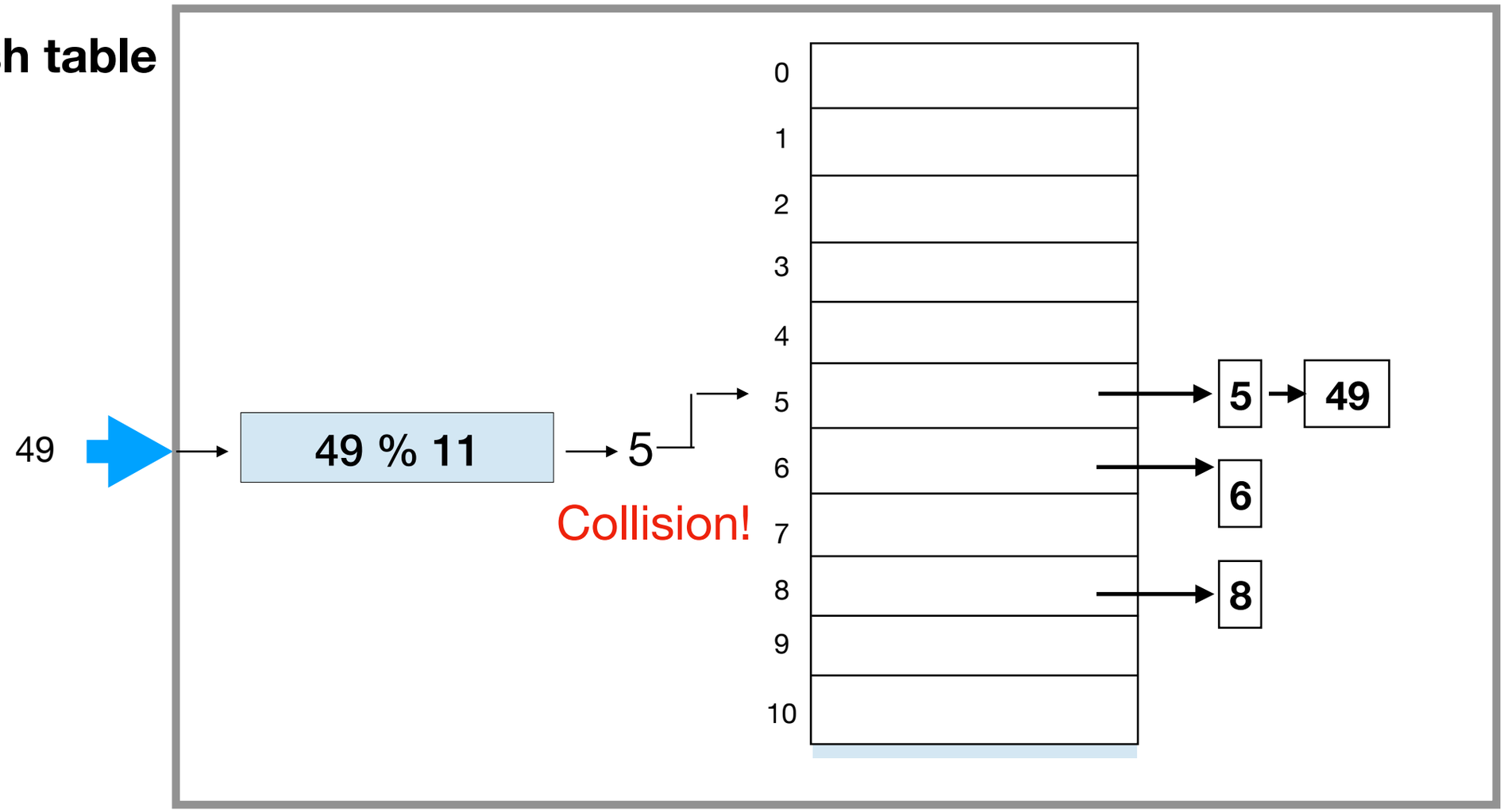
Open hashing example continued

- Insert the keys 5, 6, **8**, 49, 50, 55, 100 into a hash table of size 11.



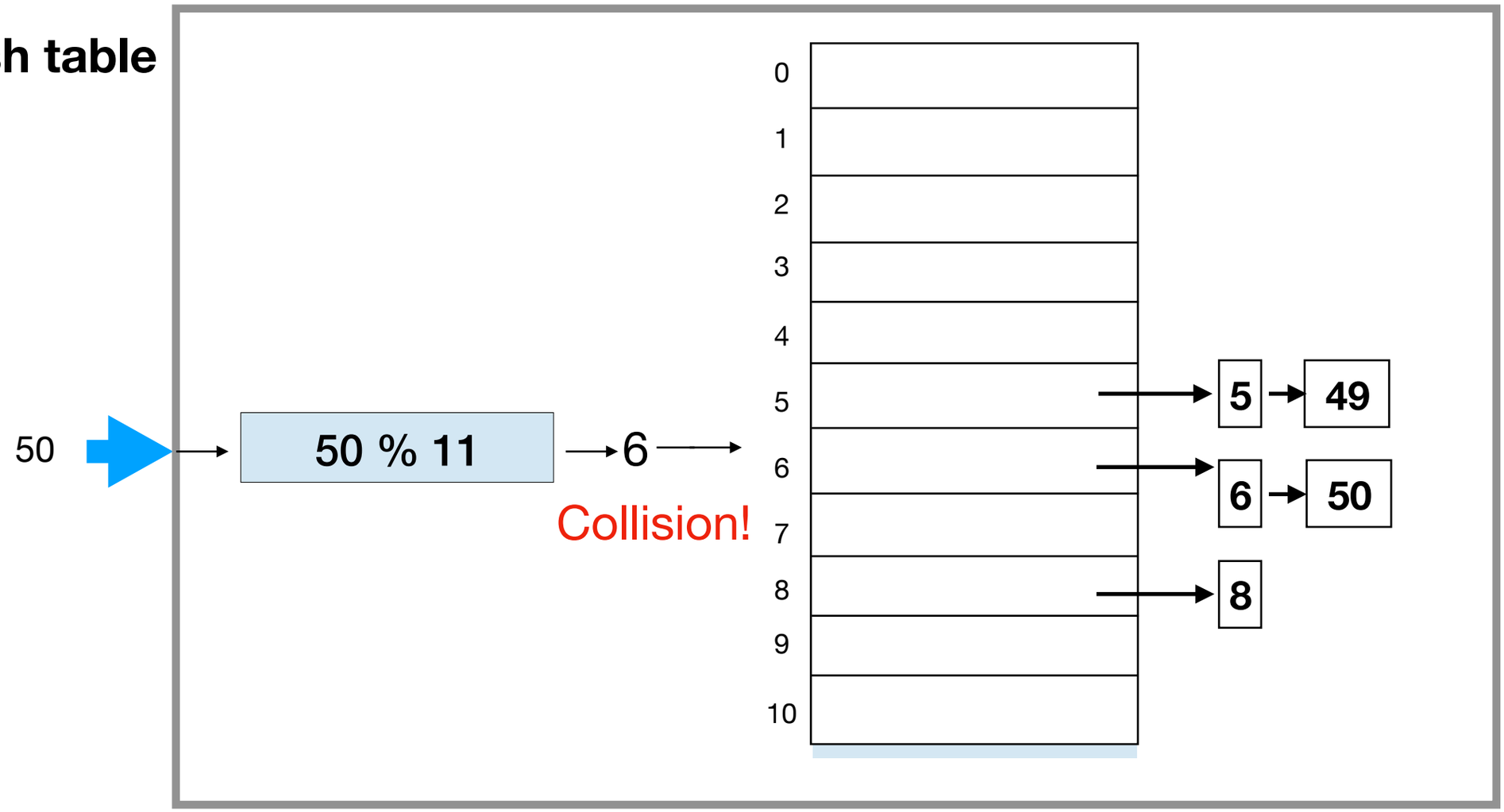
Open hashing example continued

- Insert the keys 5, 6, 8, **49**, 50, 55, 100 into a hash table of size 11.



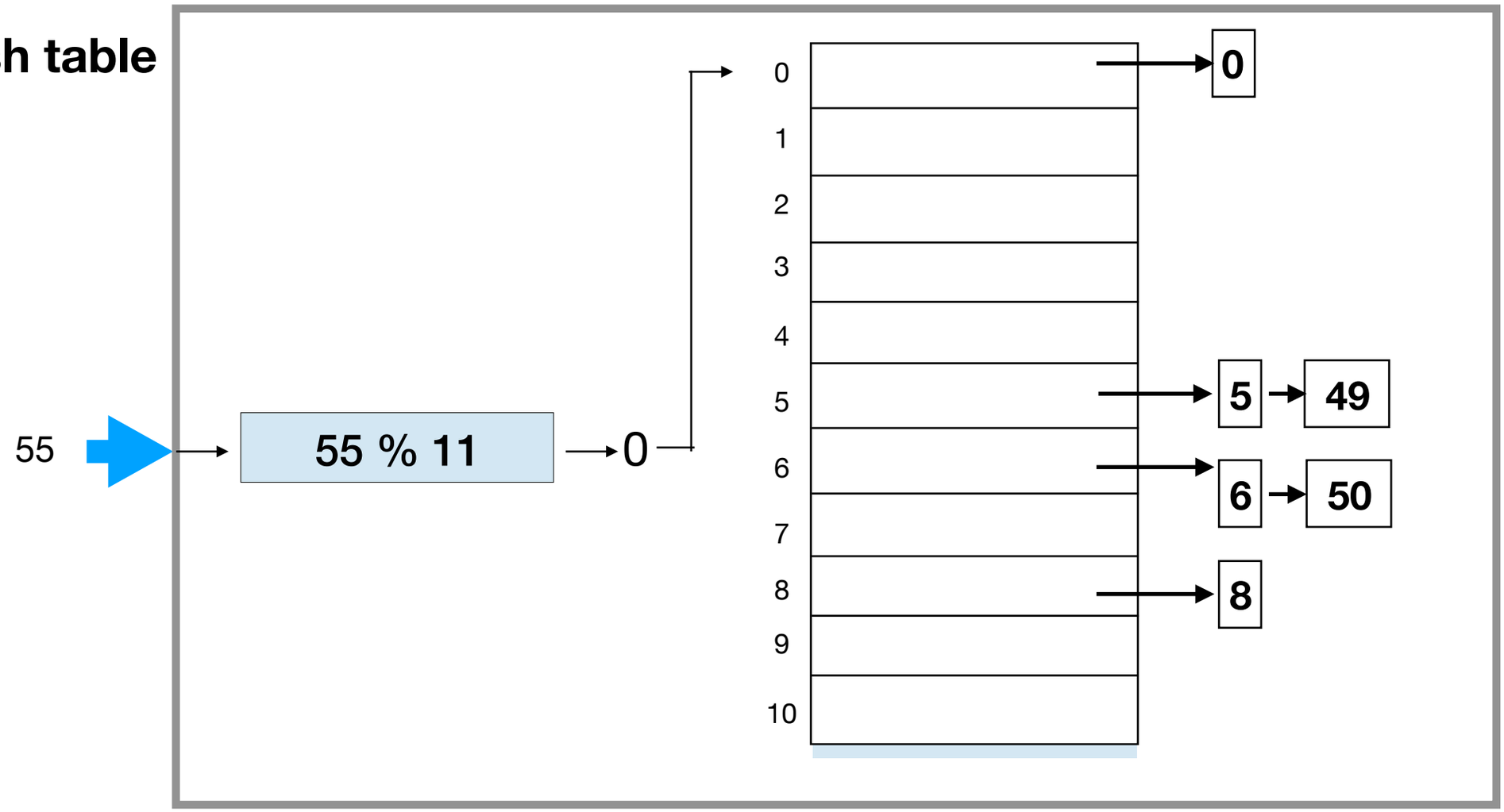
Open hashing example continued

- Insert the keys 5, 6, 8, 49, **50**, 55, 100 into a hash table of size 11.



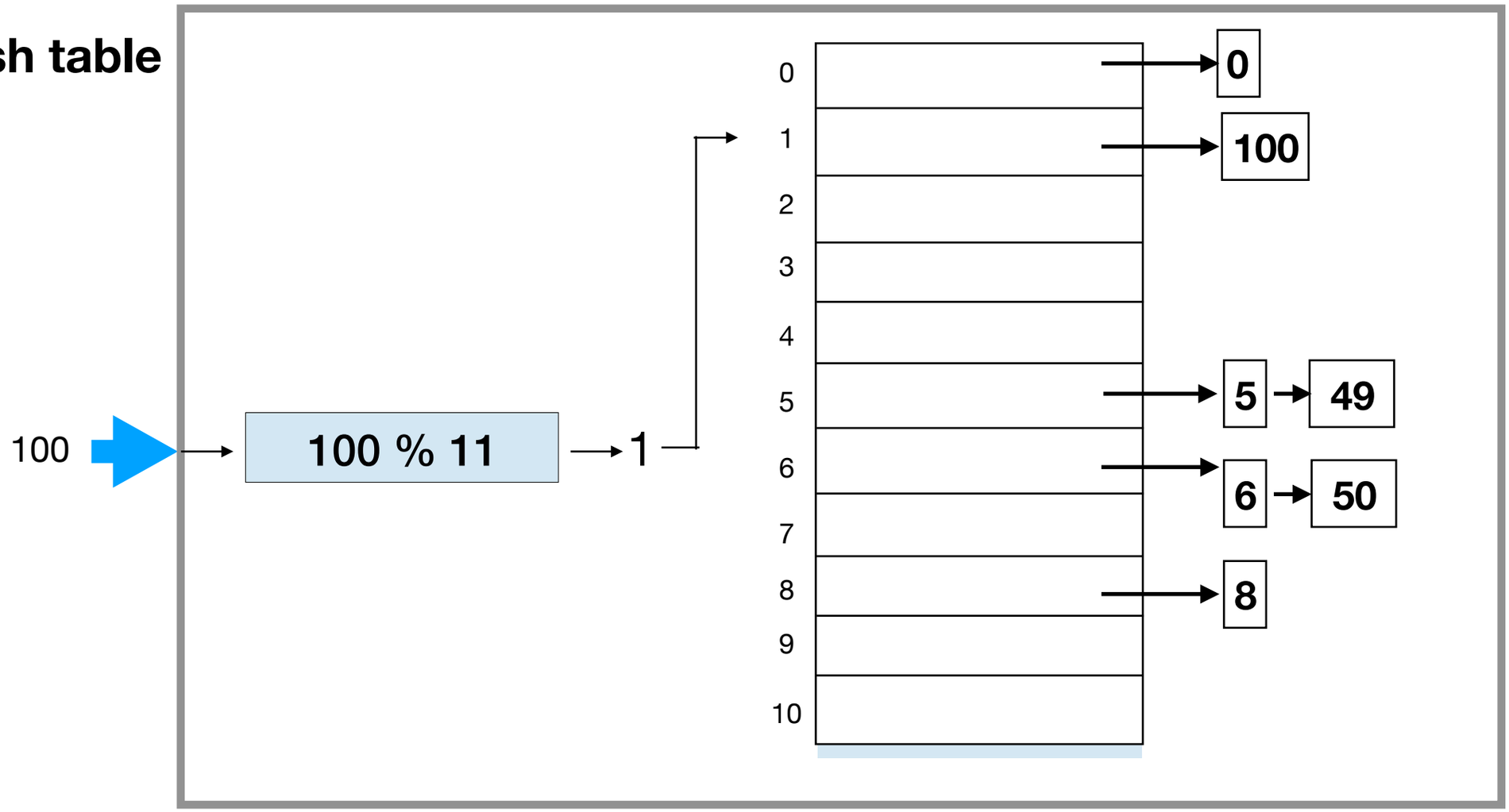
Open hashing example continued

- Insert the keys 5, 6, 8, 49, 50, **55**, 100 into a hash table of size 11.



Open hashing example continued

- Insert the keys 5, 6, 8, 49, 50, 55, **100** into a hash table of size 11.



Closed hashing

- Open hashing uses linked lists - slows access to data.
- Closed hashing uses the same array for resolving the collision
  - Tries alternate locations in the array until it finds a free one

    New hashing function $h' = (h + f(i)) \ \% \ \text{SIZE}$

  - Two types: Linear probing and Quadratic probing

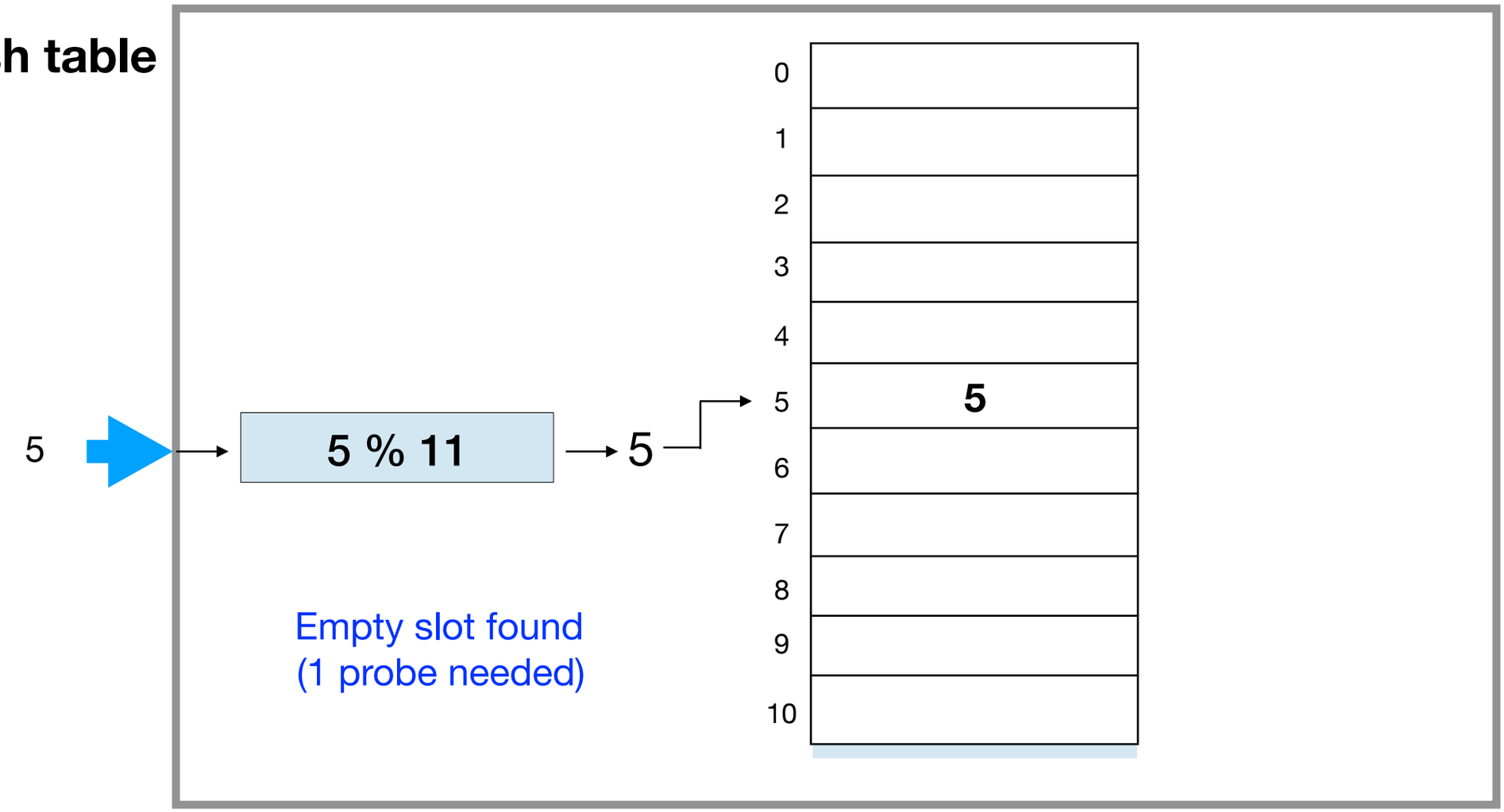
Linear probing

Hashing function $h' = (h + f(i)\ )\ \%\ \text{SIZE}$

- **In linear probing,** $f(i)$ is linear
  - Example: $f(i) = i$ means that successive locations 1, 2, 3,.., following the index *h* should be probed
  - May lead to groups of data blocks, which increase time to find given key (called primary clustering)

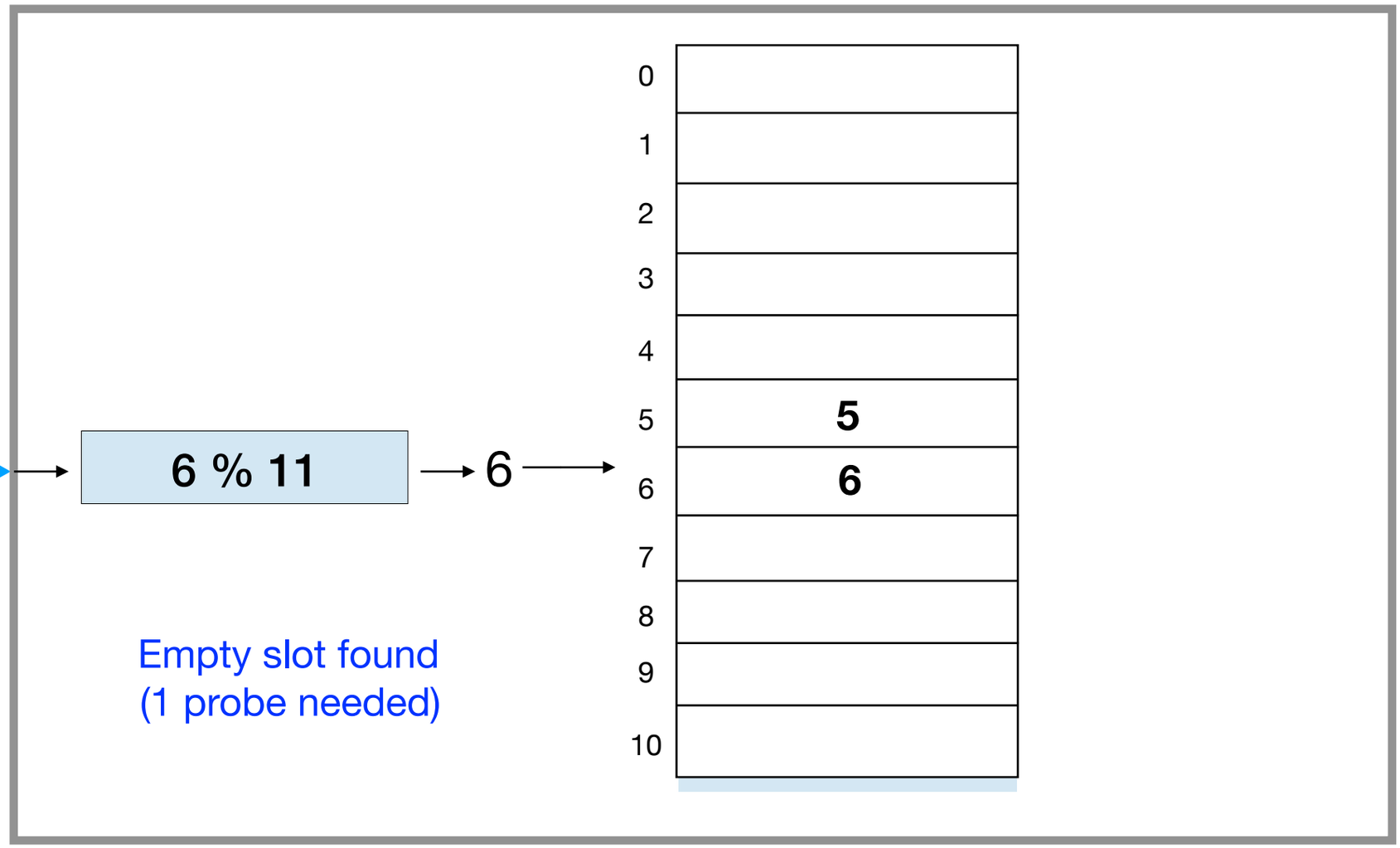
Linear probing insertion with $f(i) = i$

- Insert the keys **5**, 6, 8, 49, 50, 55, 100 into a hash table of size 11.



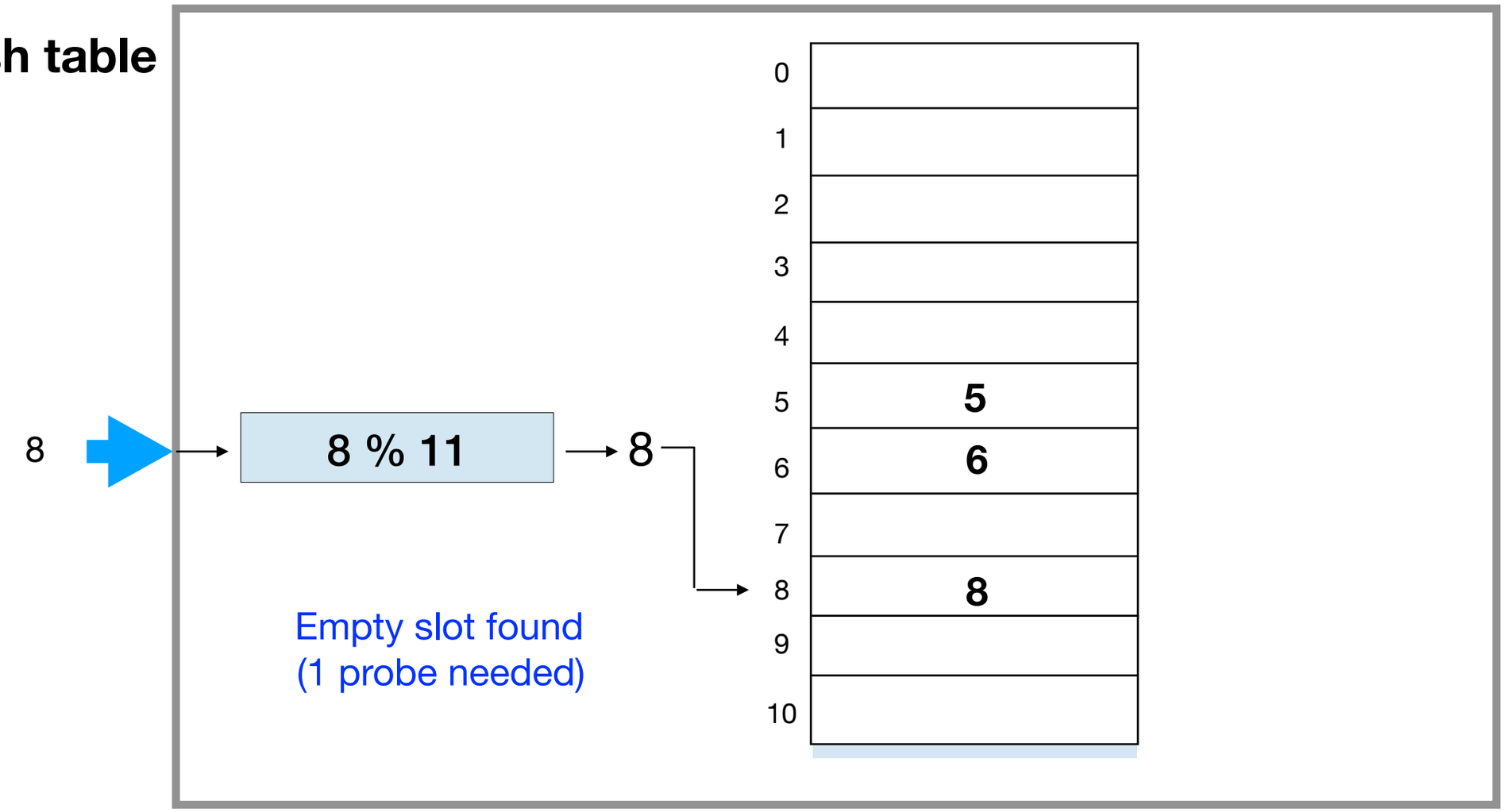
Linear probing insertion with $f(i) = i$ continued

- Insert the keys 5, **6**, 8, 49, 50, 55, 100 into a hash table of size 11.



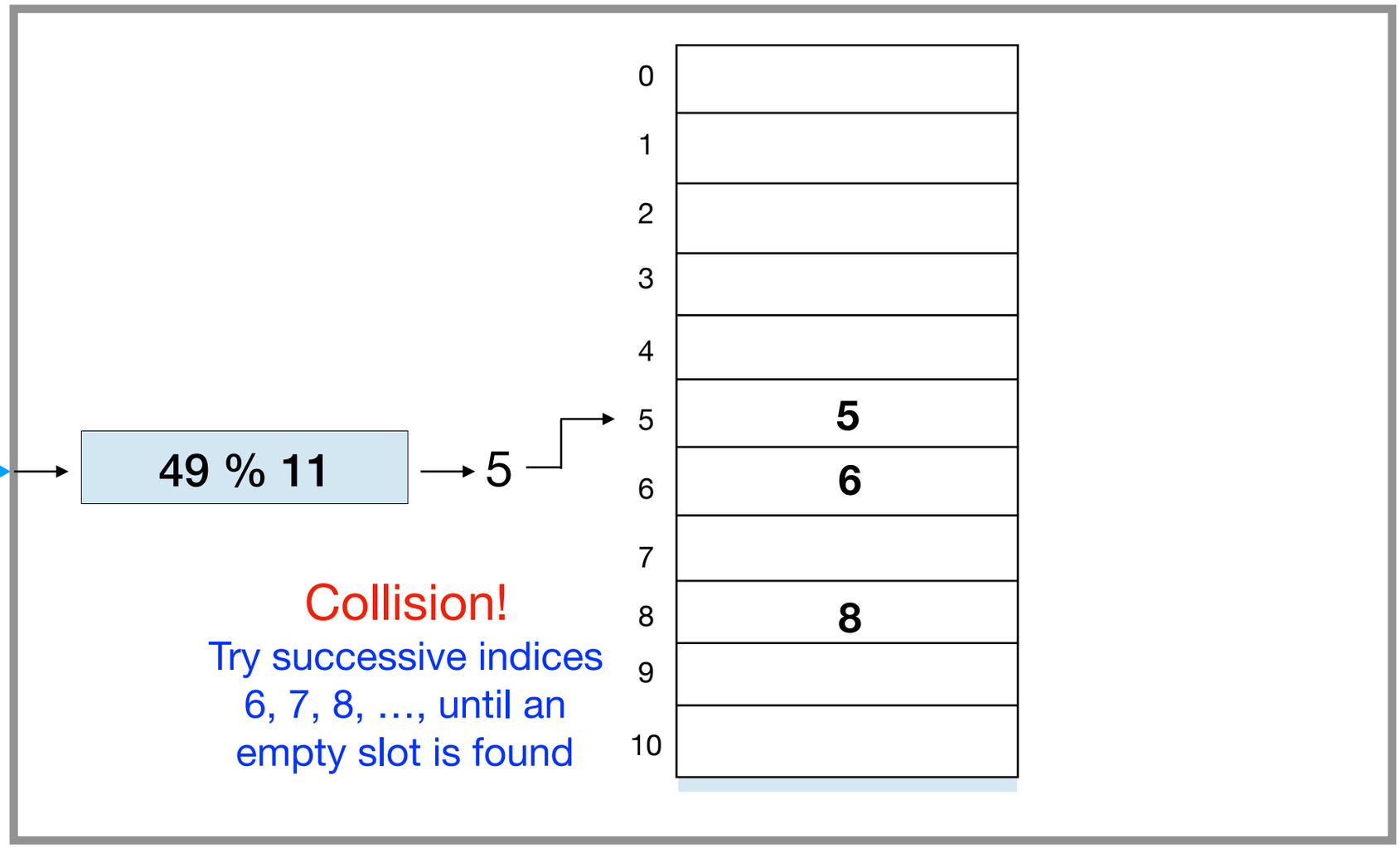
Linear probing insertion with $f(i) = i$ continued

- Insert the keys 5, 6, **8**, 49, 50, 55, 100 into a hash table of size 11.



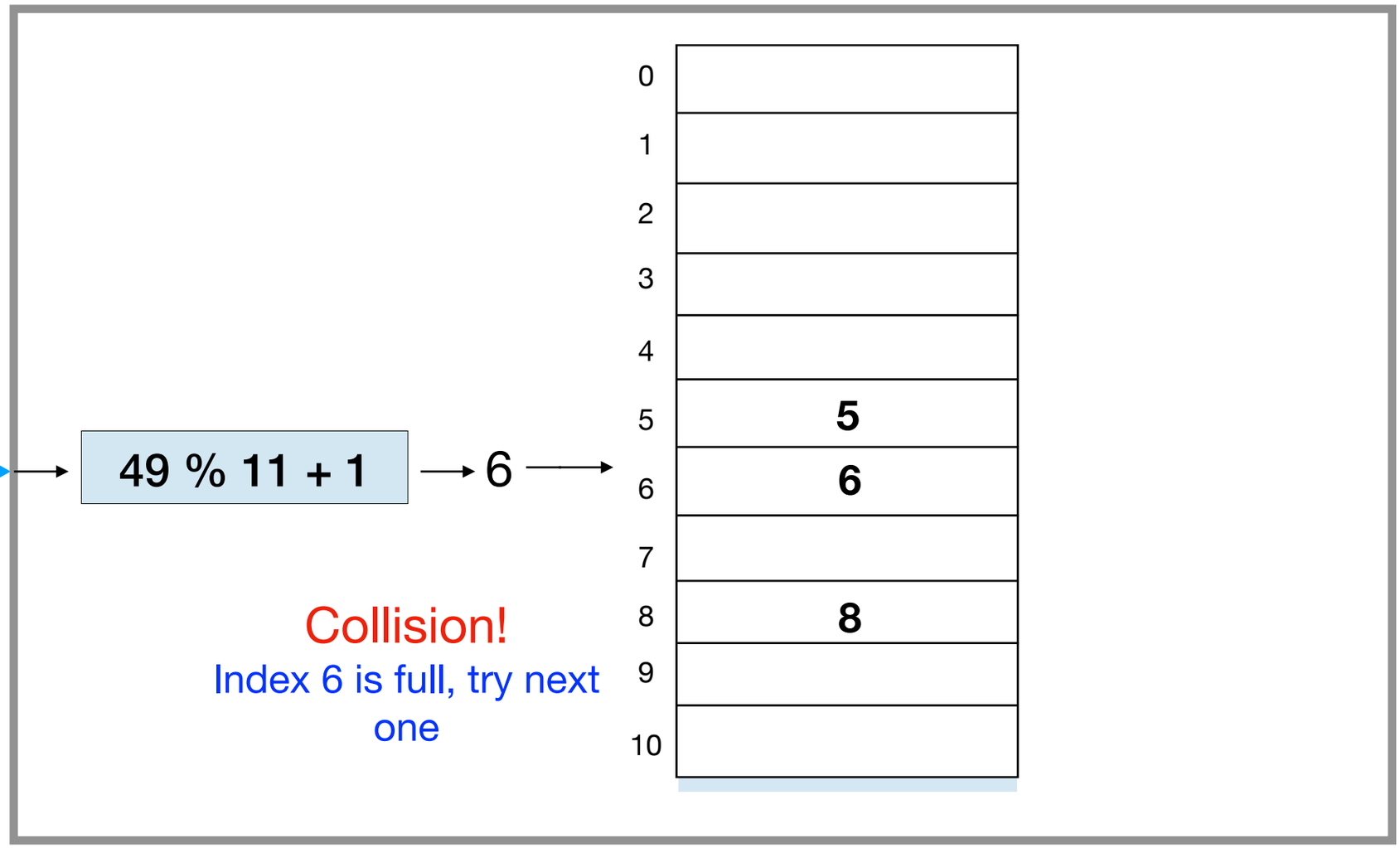
Linear probing insertion with $f(i) = i$ continued

- Insert the keys 5, 6, 8, **49**, 50, 55, 100 into a hash table of size 11.



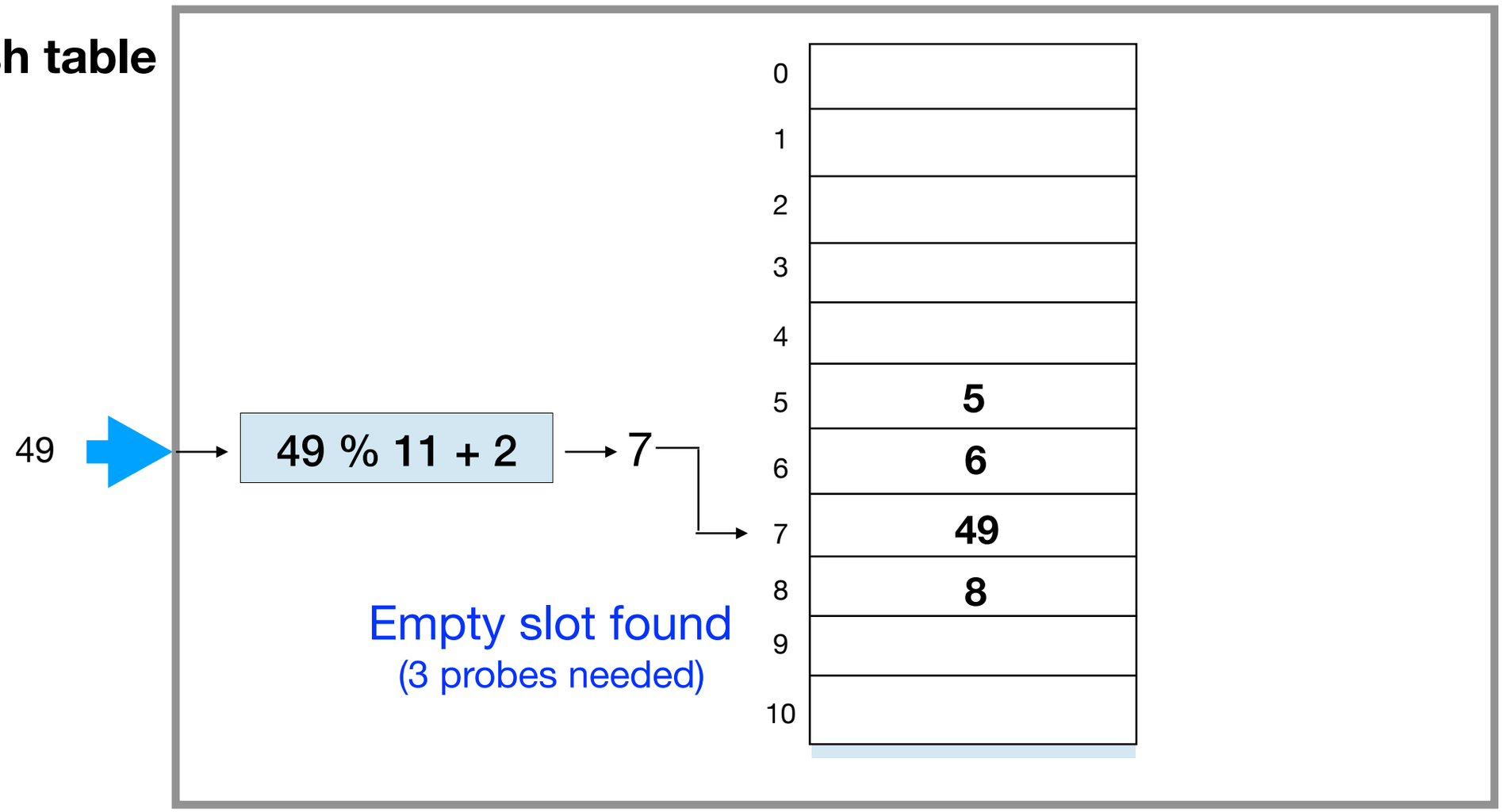
Linear probing insertion with $f(i) = i$ continued

- Insert the keys 5, 6, 8, **49**, 50, 55, 100 into a hash table of size 11.



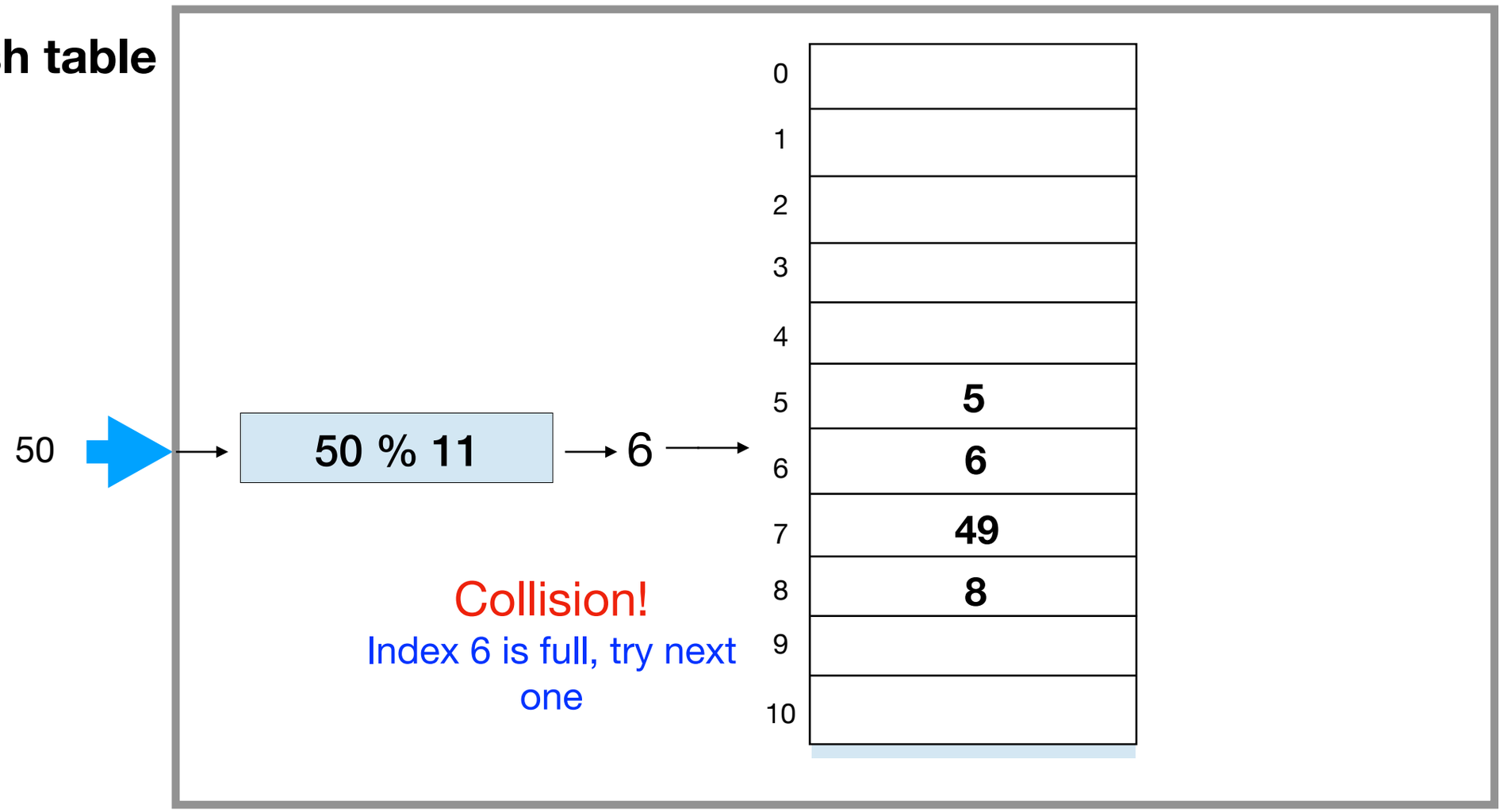
Linear probing insertion with $f(i) = i$ continued

- Insert the keys 5, 6, 8, **49**, 50, 55, 100 into a hash table of size 11.



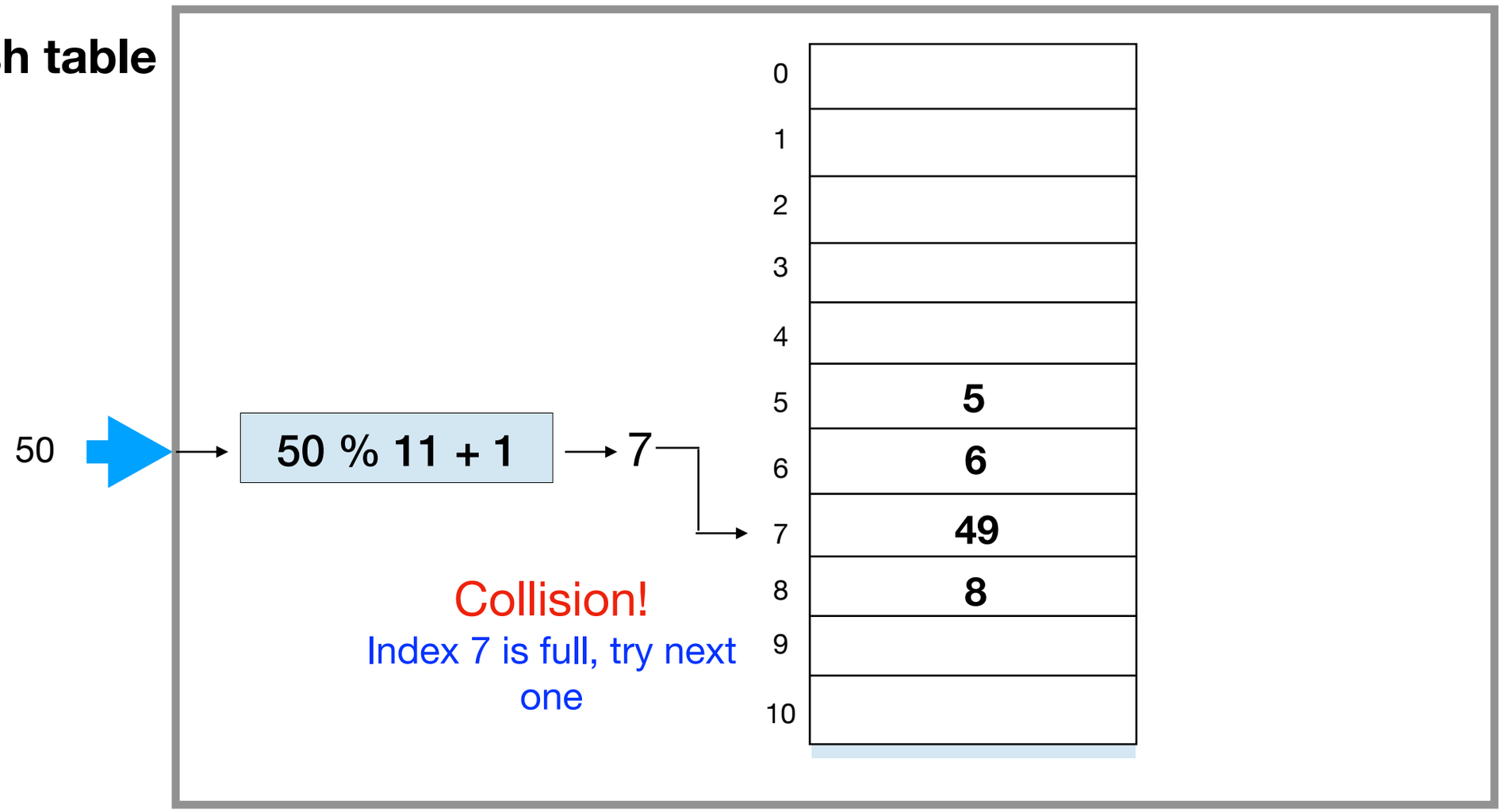
Linear probing insertion with $f(i) = i$ continued

- Insert the keys 5, 6, 8, 49, **50**, 55, 100 into a hash table of size 11.



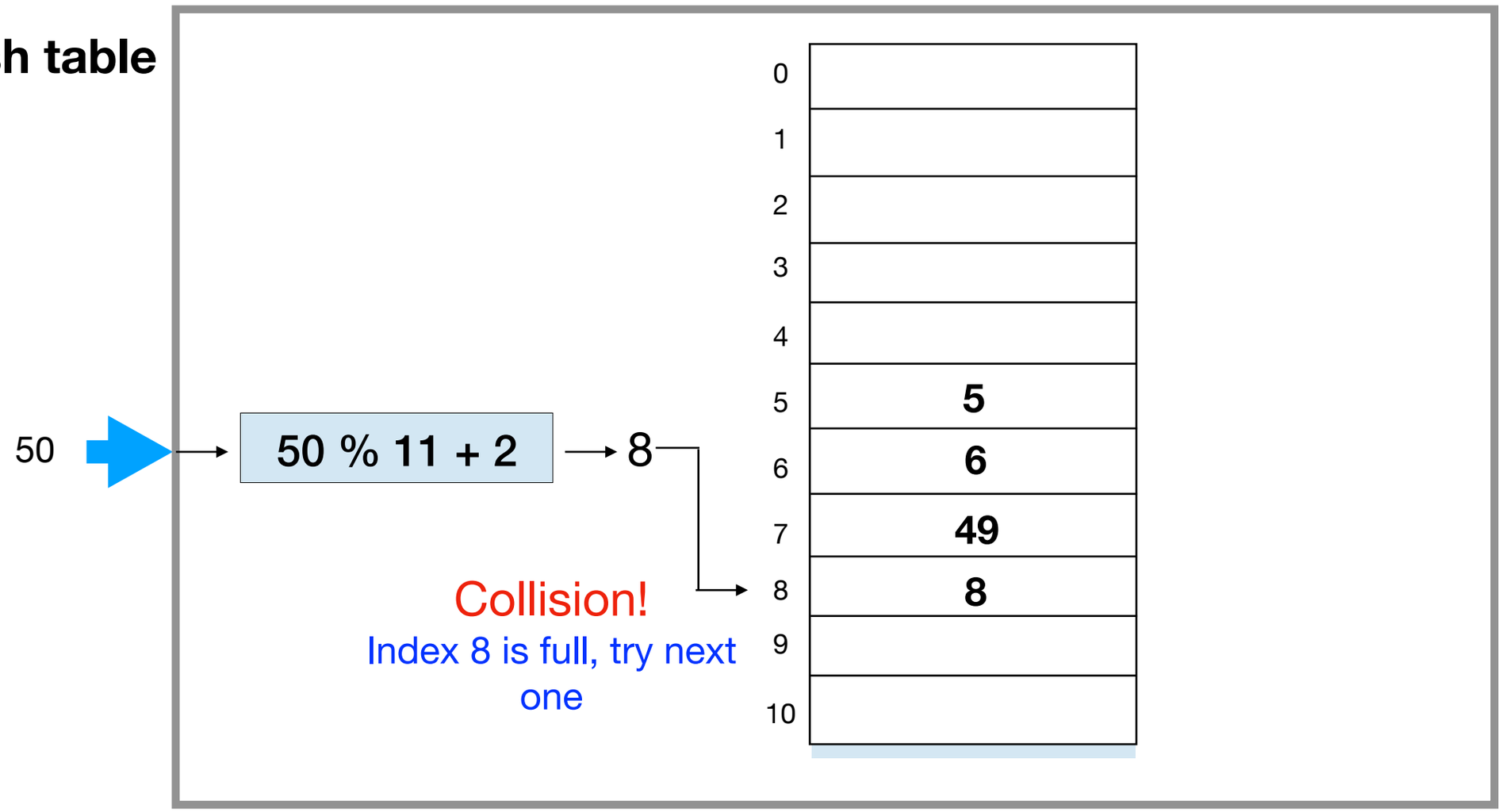
Linear probing insertion with $f(i) = i$ continued

- Insert the keys 5, 6, 8, 49, **50**, 55, 100 into a hash table of size 11.



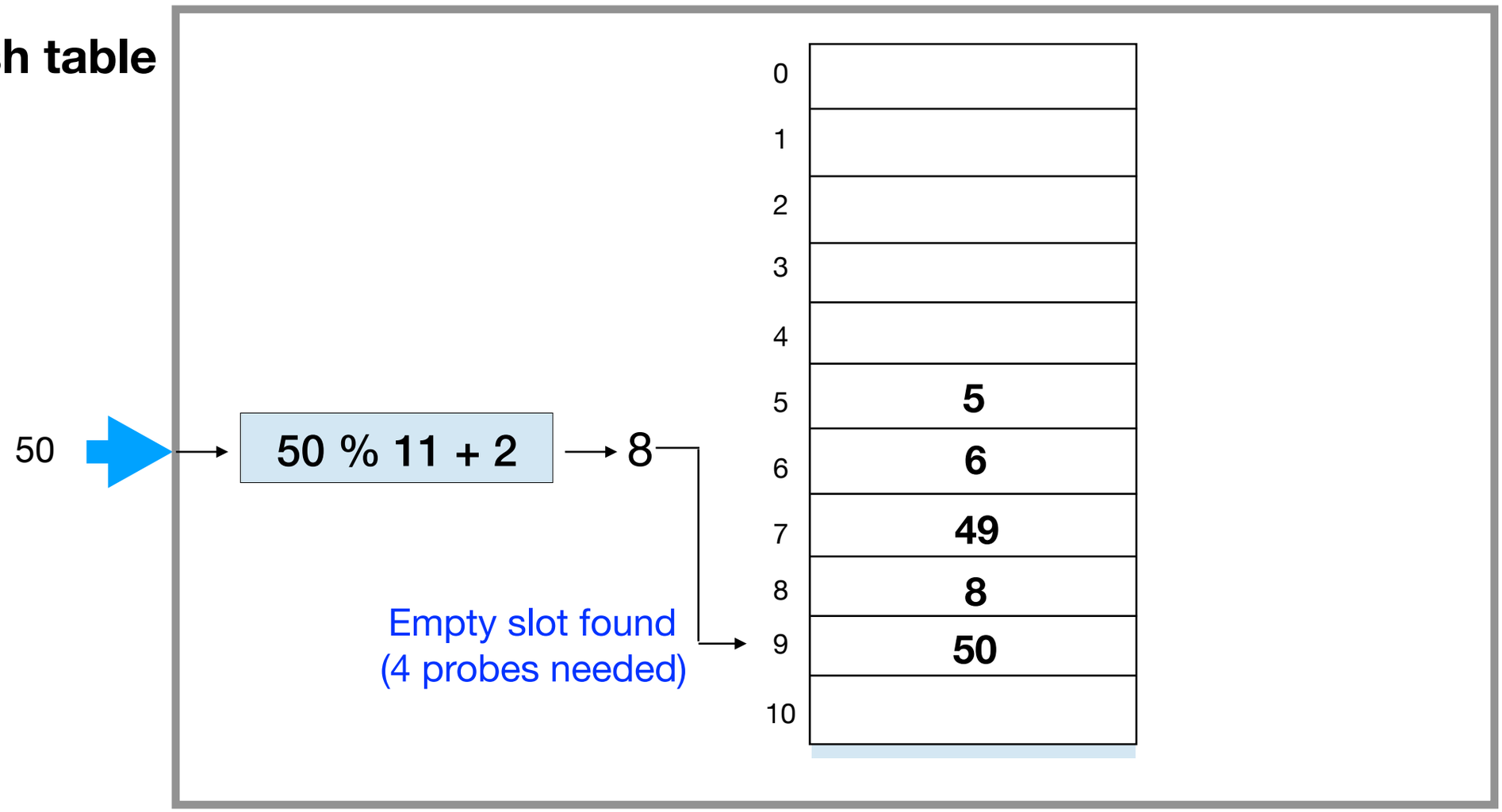
Linear probing insertion with $f(i) = i$ continued

- Insert the keys 5, 6, 8, 49, **50**, 55, 100 into a hash table of size 11.



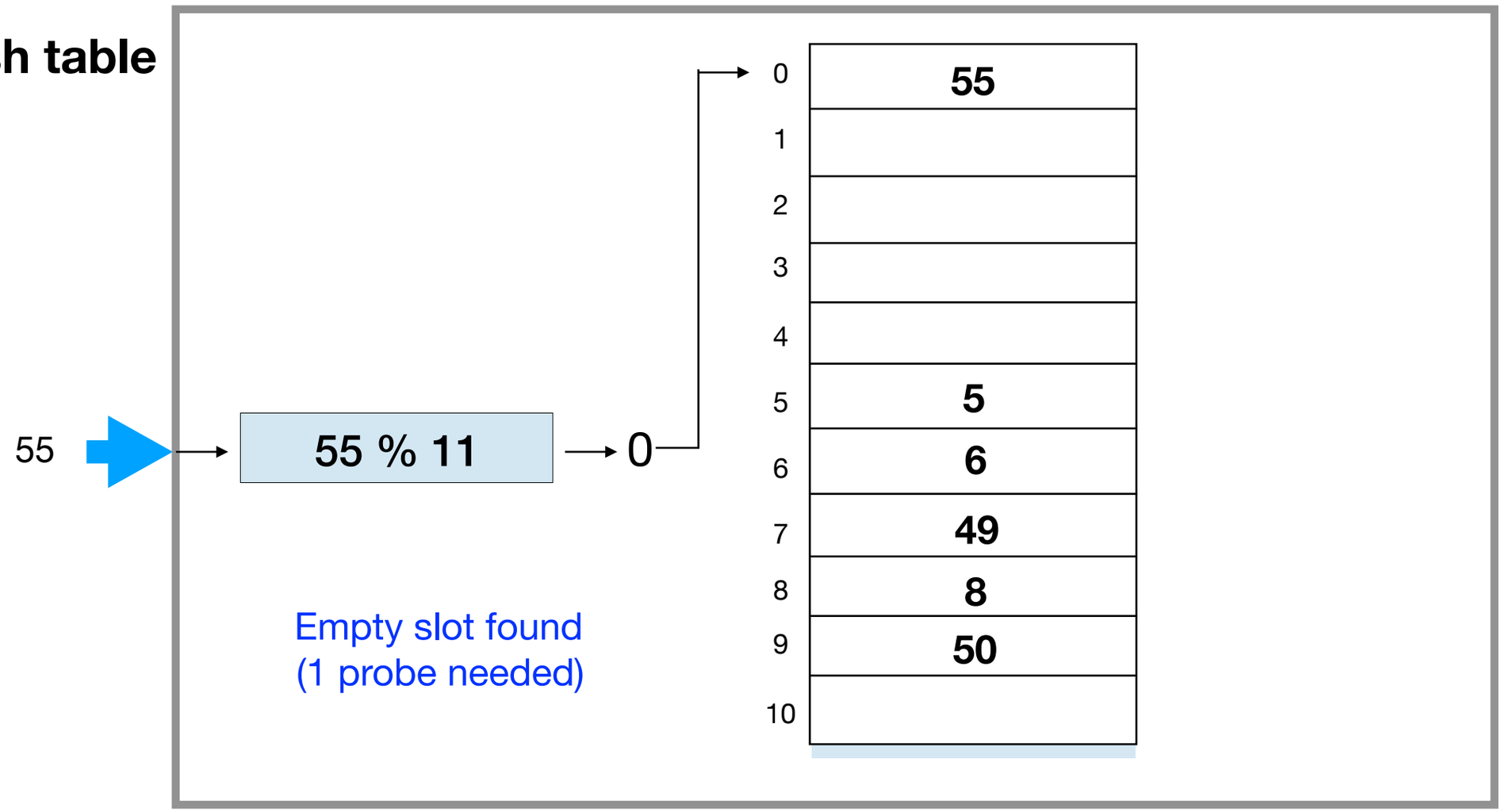
Linear probing insertion with $f(i) = i$ continued

- Insert the keys 5, 6, 8, 49, **50**, 55, 100 into a hash table of size 11.



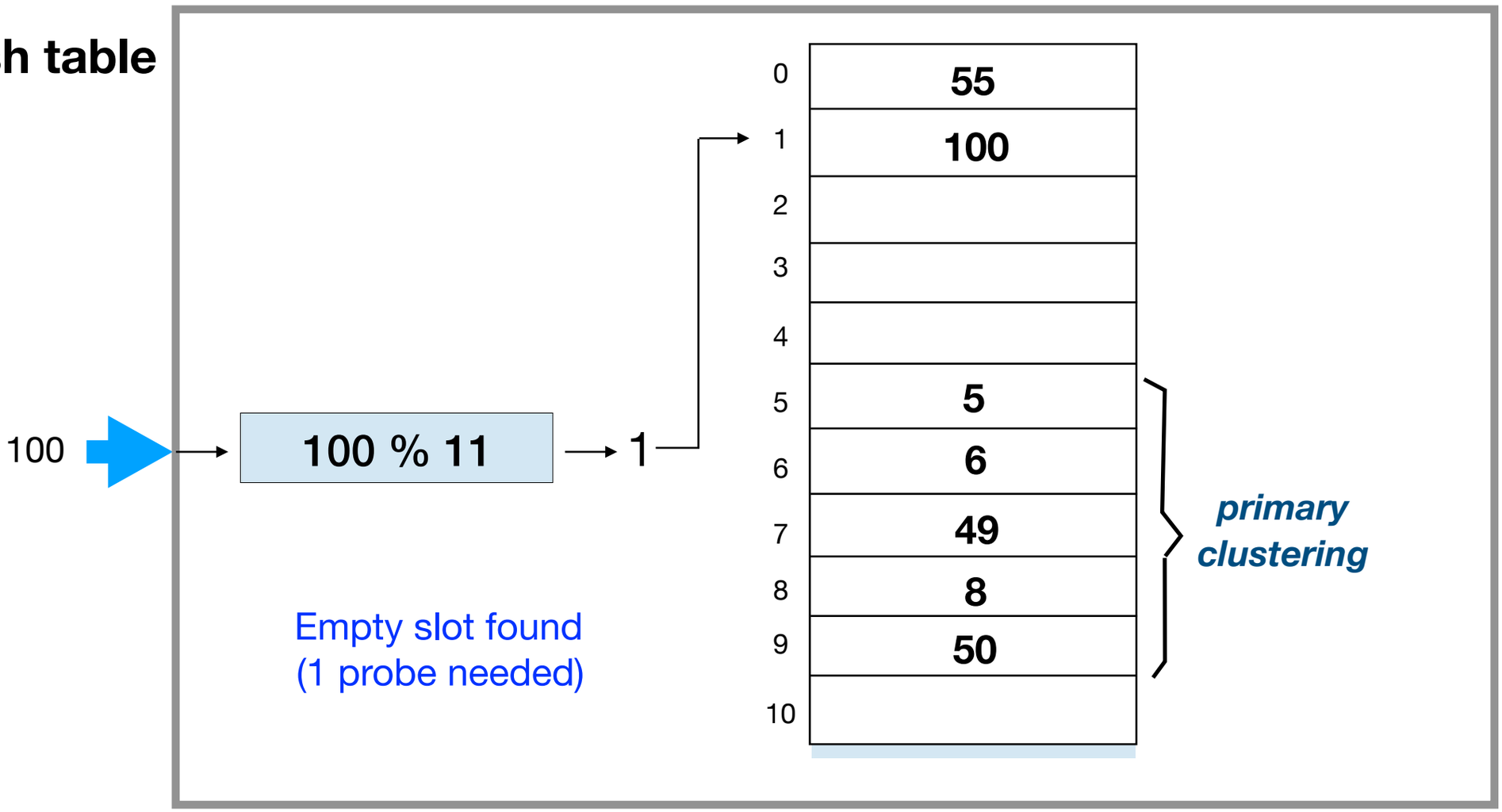
Linear probing insertion with $f(i) = i$ continued

- Insert the keys 5, 6, 8, 49, 50, **55**, 100 into a hash table of size 11.



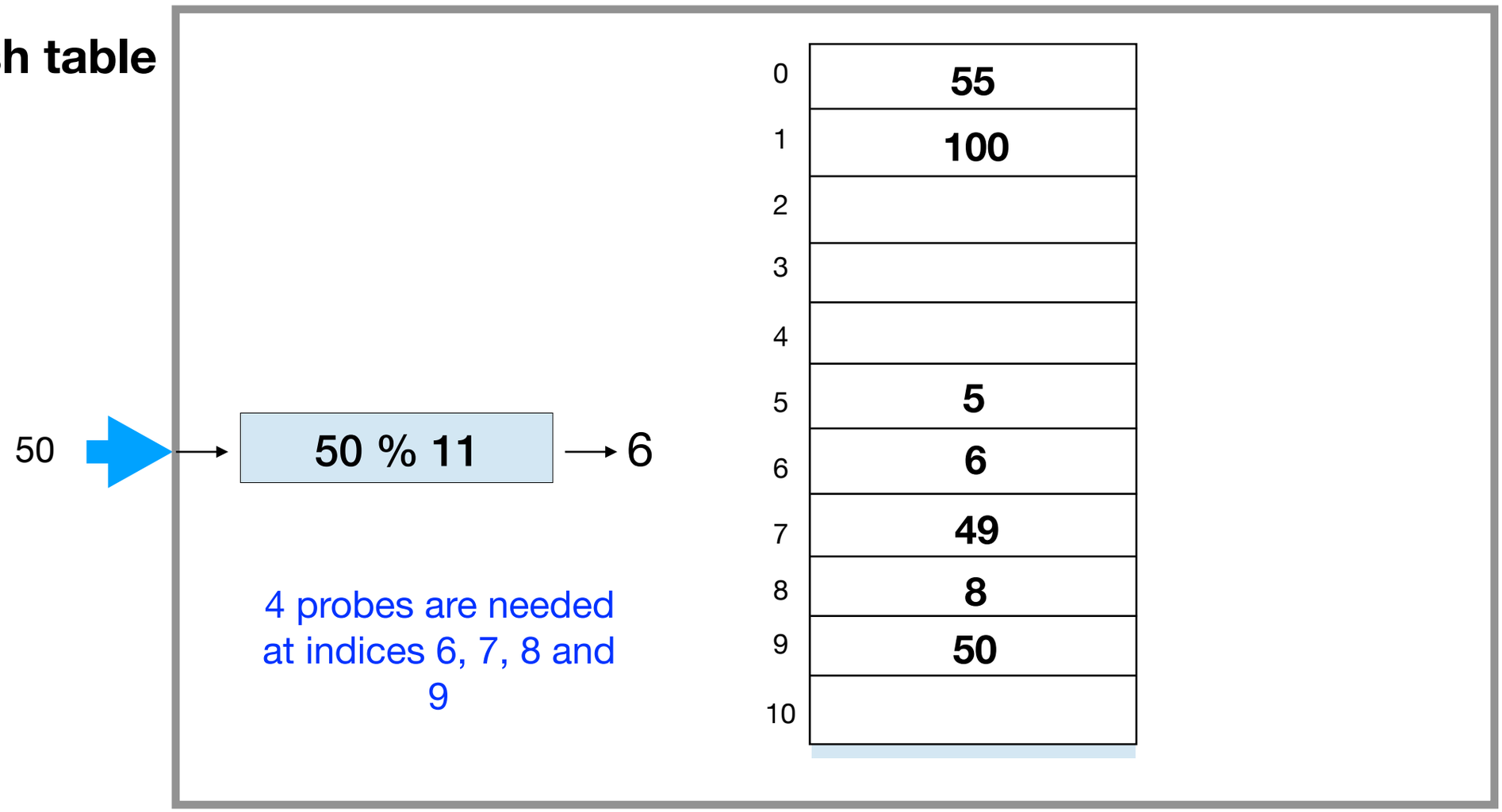
Linear probing insertion with $f(i) = i$ continued

- Insert the keys 5, 6, 8, 49, 50, 55, **100** into a hash table of size 11.



Linear probing insertion with $f(i) = i$ continued

- Find the key 50 in the hash table - how many probes are needed?



Quadratic probing

Hashing function $h' = (h + f(i)\ )\ \%\ \text{SIZE}$

- In quadratic probing, $f(i)$ is quadratic
  - Example: $f(i) = i + i^2$ means that

$i = 1, f(1) = 1 + 1 = 2$

$i = 2, f(2) = 2 + 2*2 = 6$

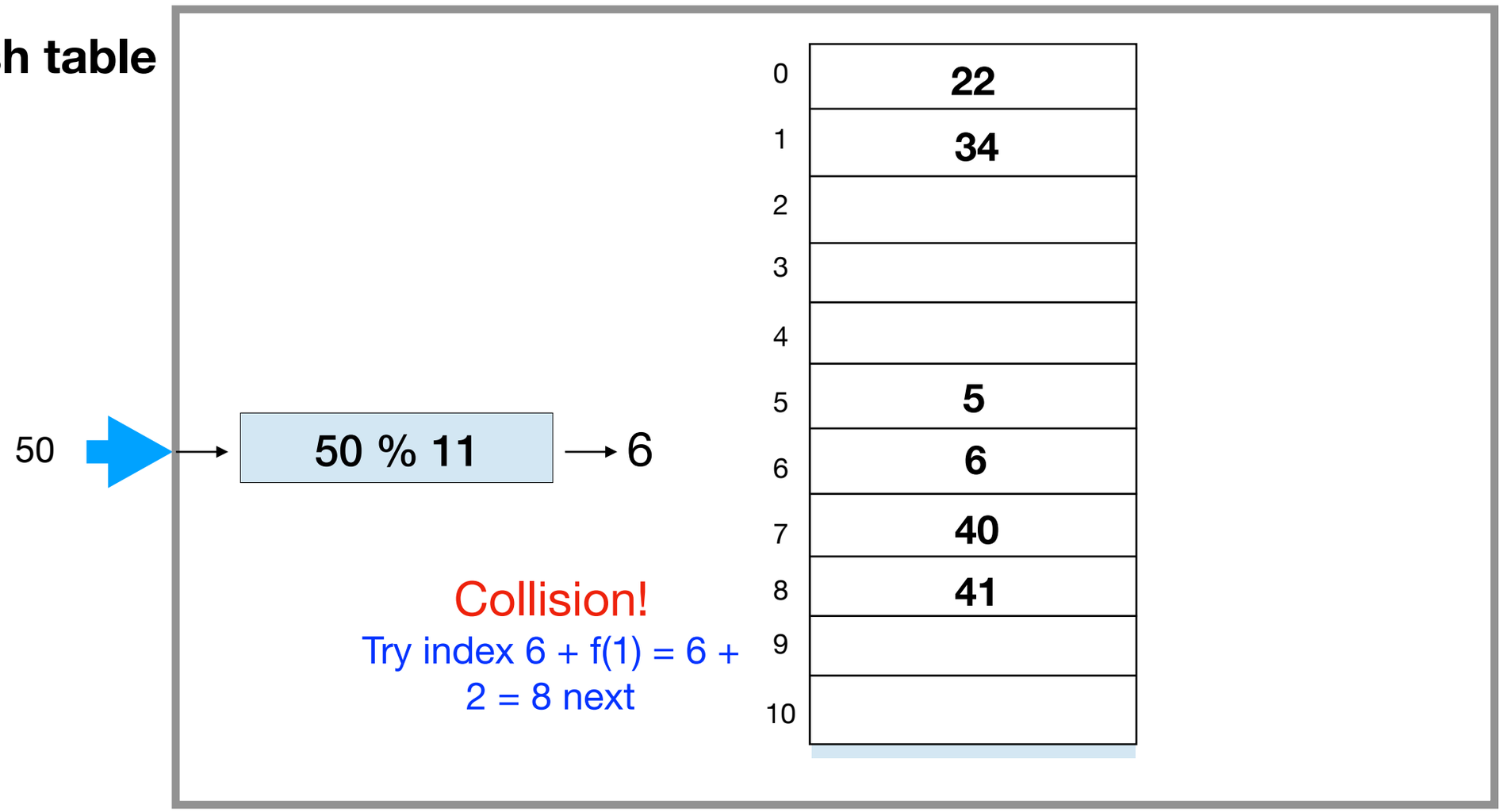
$i = 3, f(3) = 3 + 3*3 = 12$

So locations 2, 6, 12,.., (for *i* = 1, 2, 3, …) following index *h* should be probed

Quadratic probing insertion with

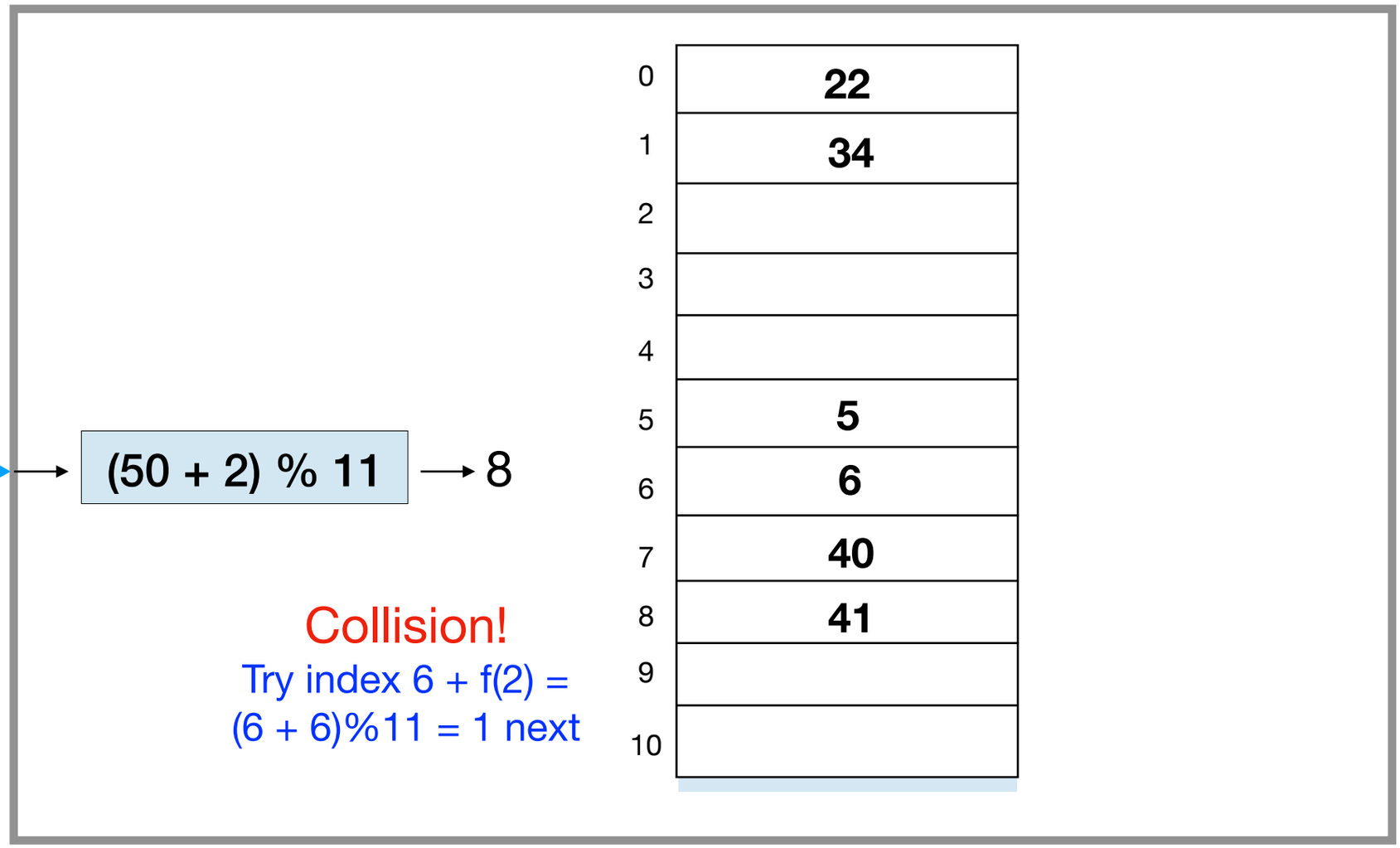
$$f(i) = i + i^2$$

- Insert key 50 in the following hash table - how many probes are needed?



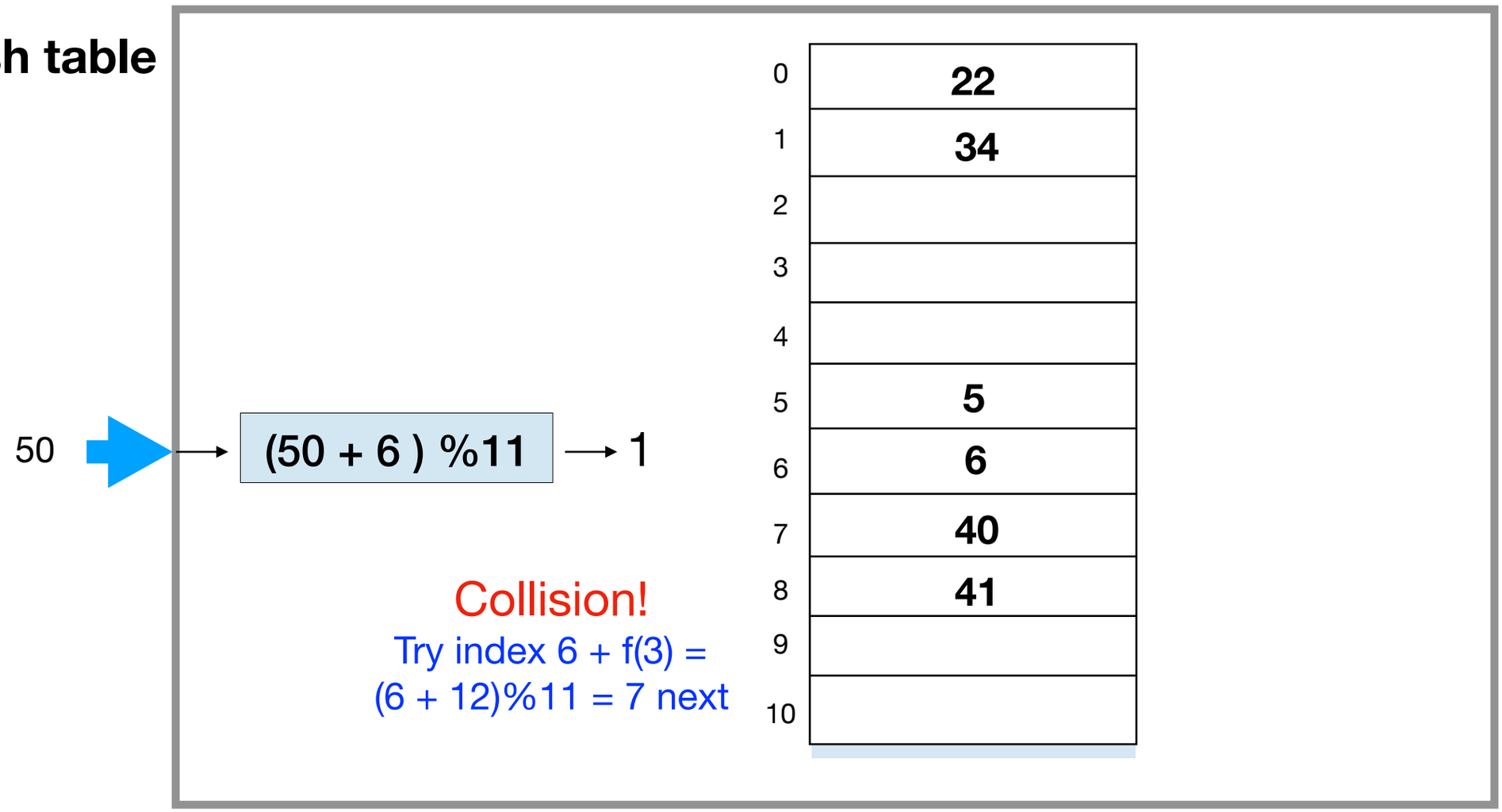
Quadratic probing insertion with $f(i) = i+i^2$ continued

- Insert key 50 in the following hash table - how many probes are needed?



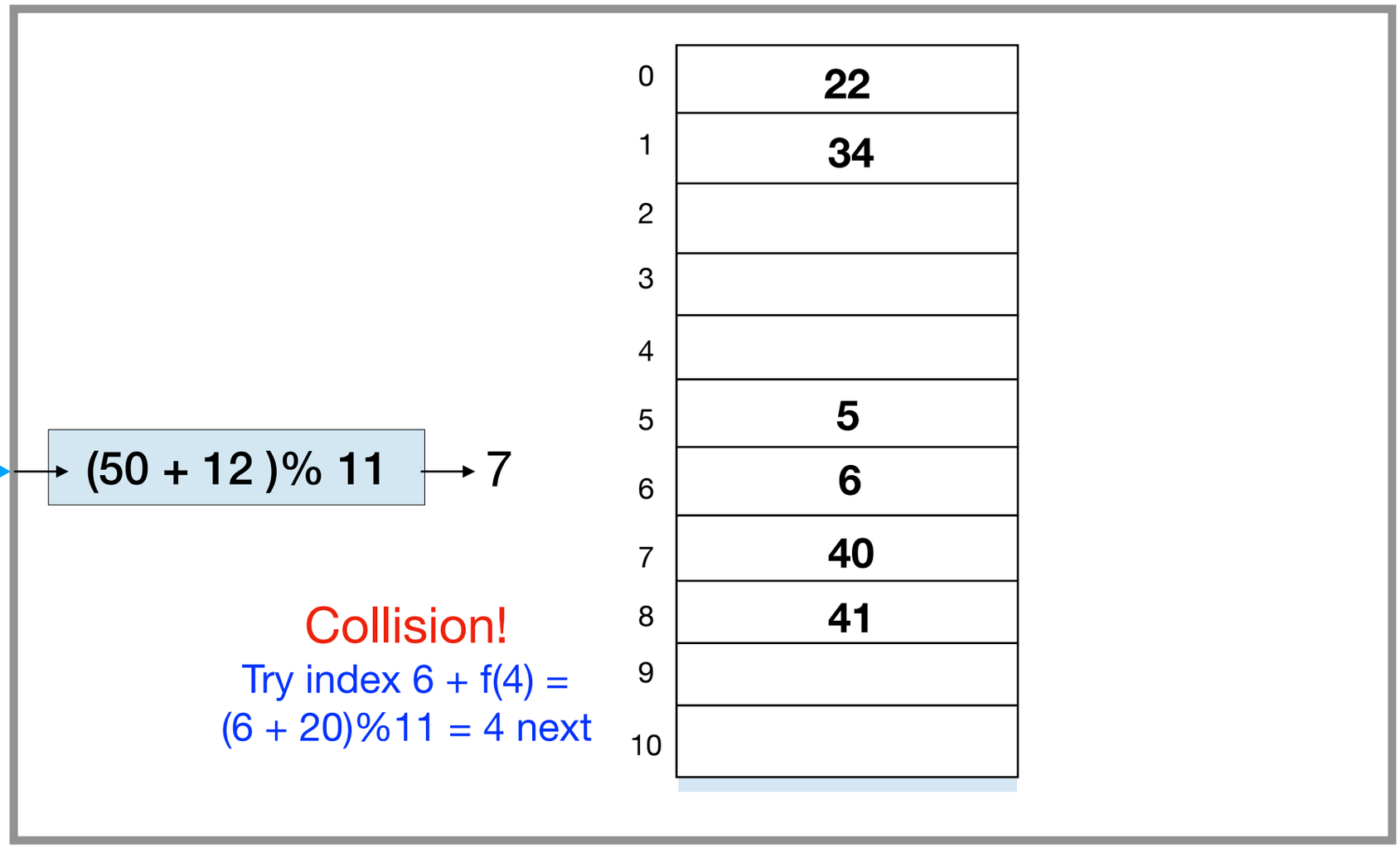
Quadratic probing insertion with $f(i) = i+i^2$ continued

- Insert key 50 in the following hash table - how many probes are needed?



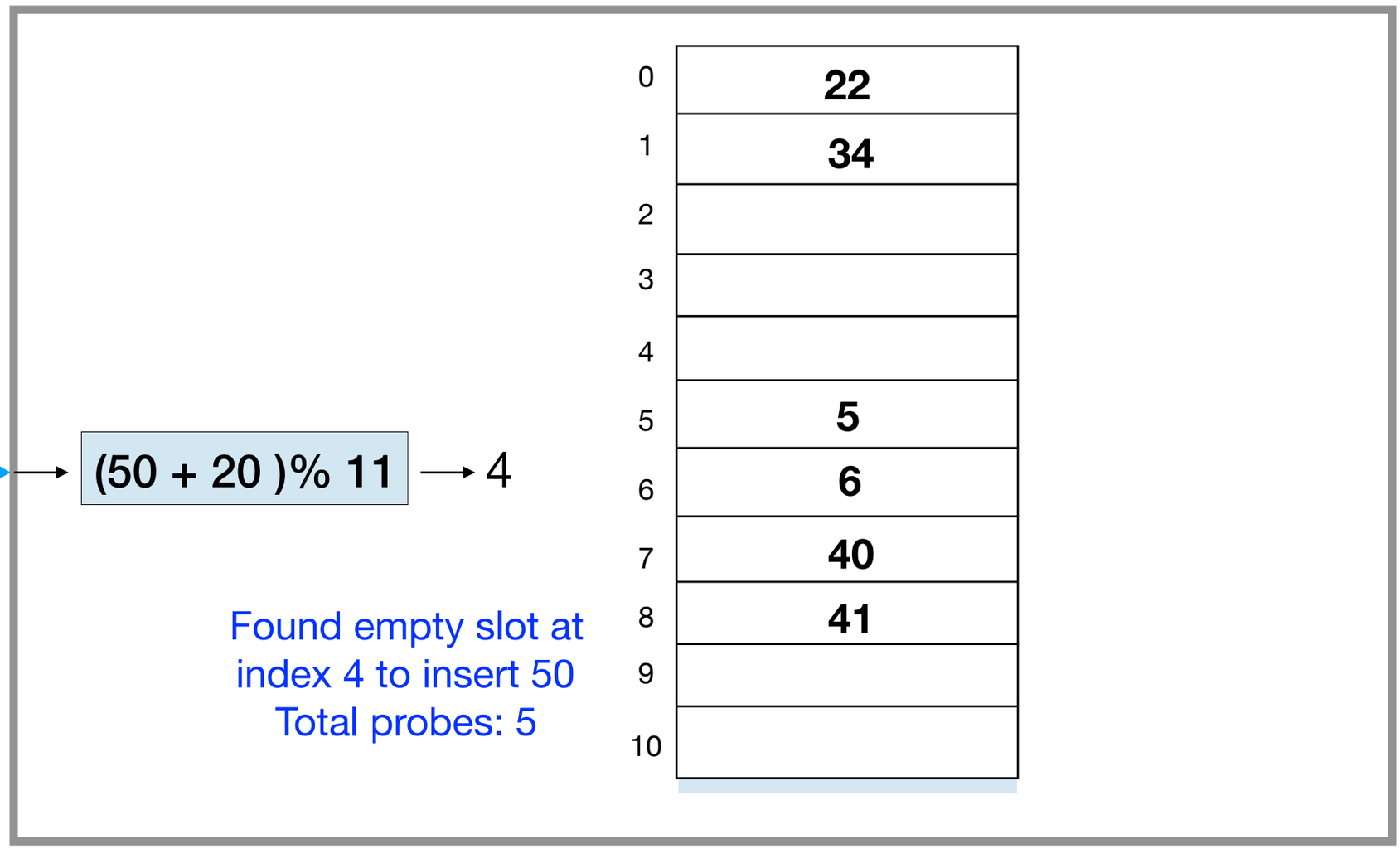
Quadratic probing insertion with $f(i) = i+i^2$ continued

- Insert key 50 in the following hash table - how many probes are needed?



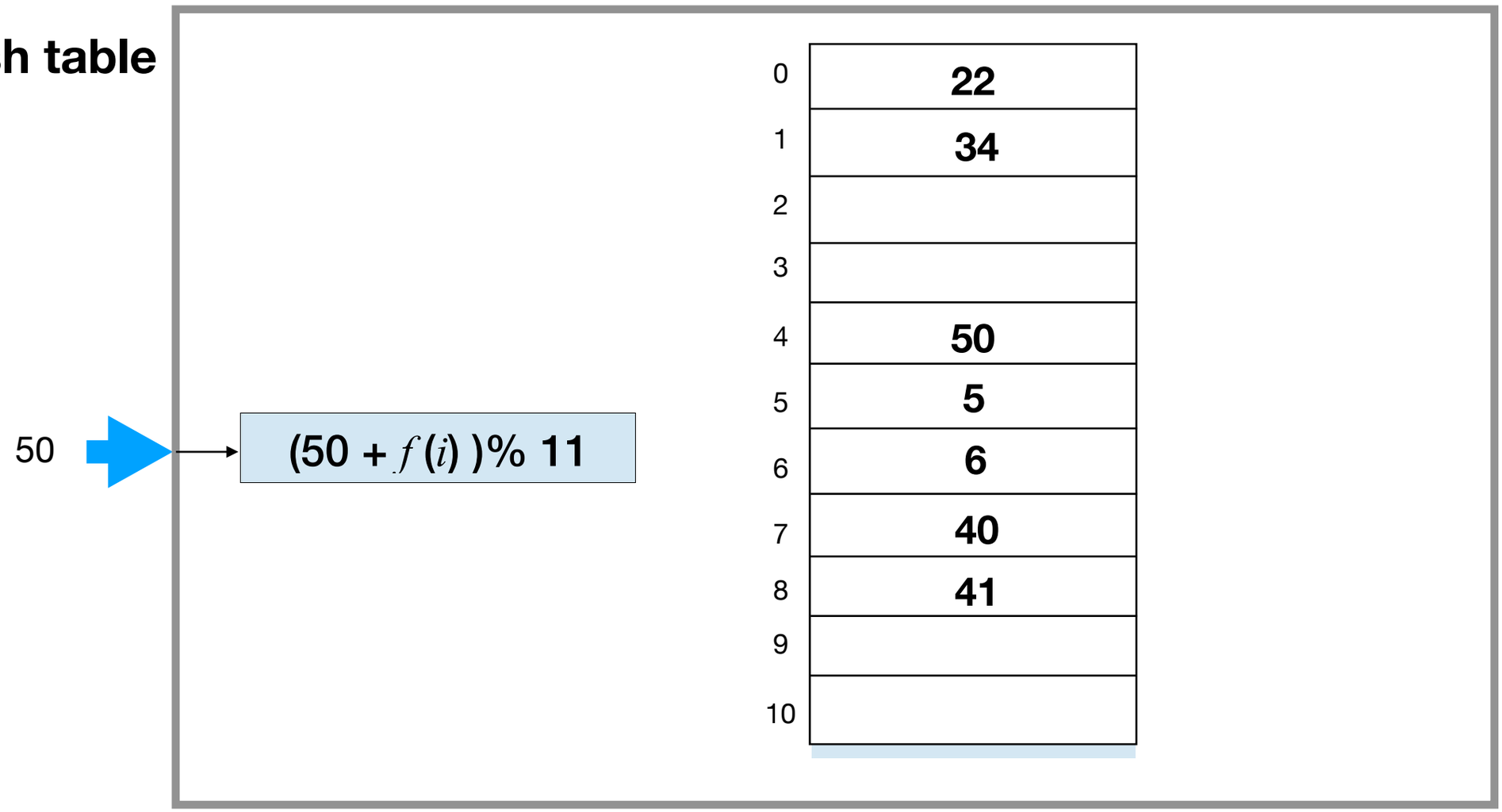
Quadratic probing insertion with $f(i) = i+i^2$ continued

- Insert key 50 in the following hash table - how many probes are needed?



Exercise: Quadratic probing search with $f(i) = i+i^2$

- Find key 50 in the following hash table - how many probes are needed?



Answer: 5 probes

Quadratic probing

- Helps avoid primary clustering (group of filled consecutive slots), but suffers from *secondary* clustering.

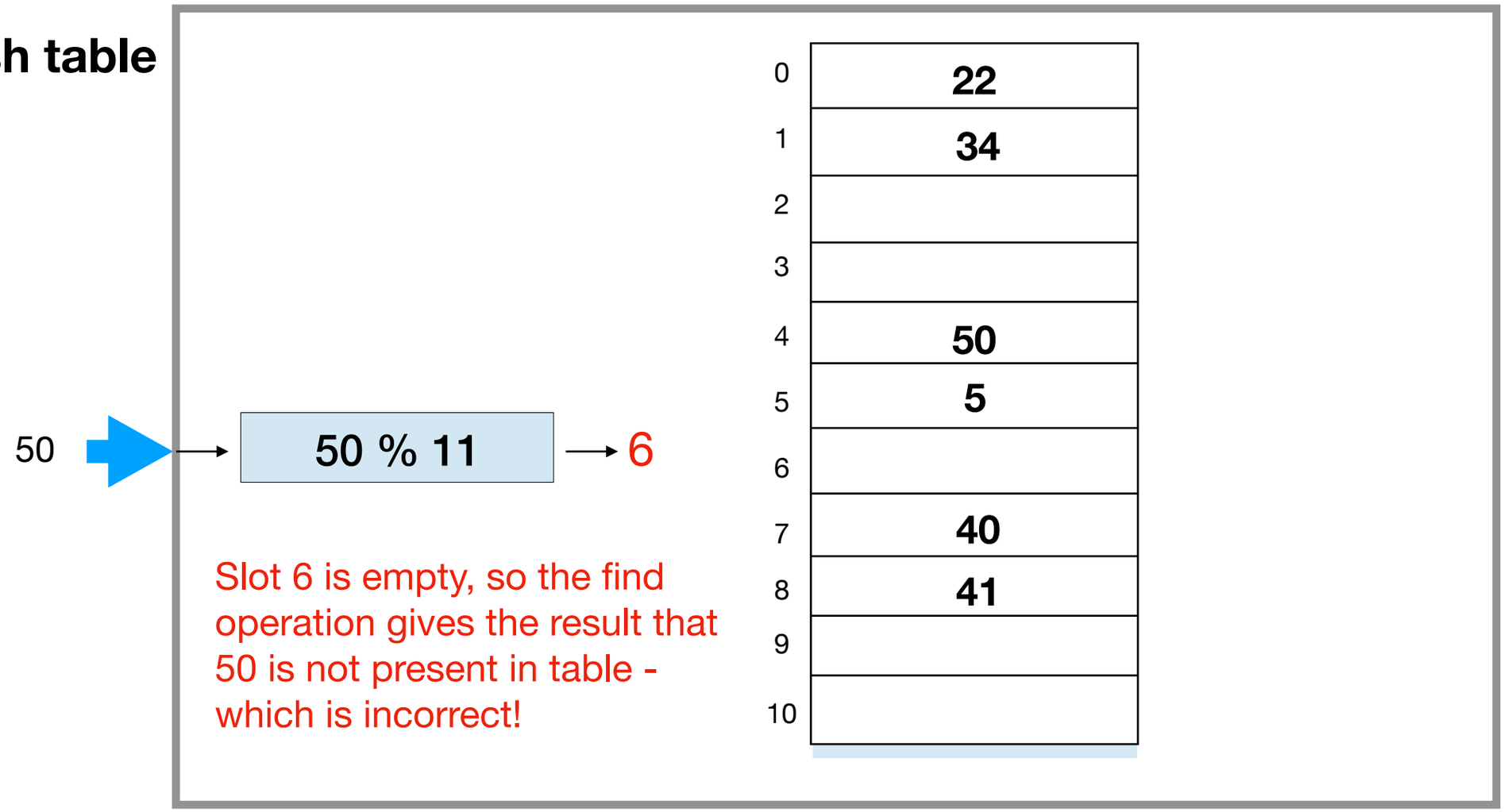
- With secondary clustering, keys that hash to the same location probe the same alternate cells and lead to multiple collisions.

- Example, inserting 50 into the hash table in previous example led to 4 collisions - the slots involved in these collisions form a secondary cluster.

- A secondary cluster is formed by non-consecutive filled slots unlike a primary cluster in which the filled slots are consecutive.

Example: Deletion in closed hashing

- Delete key 6 at index 6, then try to find key 50.

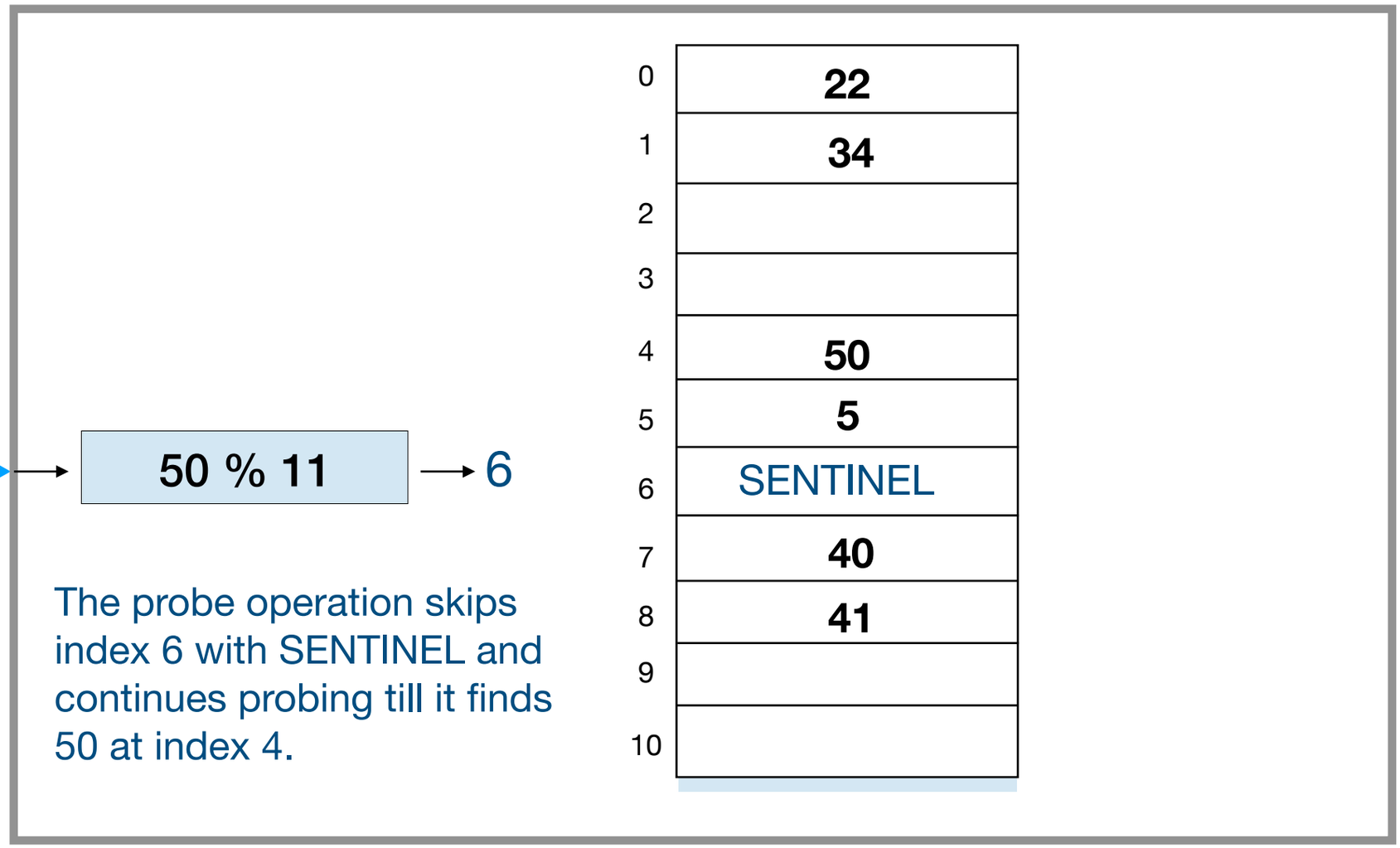


Deletion in closed hashing

- Cannot delete a key from a table that uses closed hashing - why?
    - Items may have skipped over certain keys during insertion, so if an item is deleted, the search will end prematurely at the empty slot left by the deletion.
- Two solutions:
    1. Replace the deleted key with a sentinel key that can be skipped during searches but can be used for insertion.
    2. Rehash the keys that are affected by the deleted key.

Example: Deletion in closed hashing using a sentinel

- Delete key 6 at index 6, and replace it with **SENTINEL**. Then try to find key 50.



Double hashing

- Collision resolution method that uses a second hash function:

$$f(i) = i * \text{hash}_2(key)$$

  - Apply a second hash function to *key* and probe at $\text{hash}_2(key)$, $2 * \text{hash}_2(key)$, …
  - The second hash function must never evaluate to 0, otherwise, no new locations in the table will be probed.
  - Example: $f(i) = i$ * (R - key % R), where R is prime that is smaller that the size of the hash table.

Rehashing

- If the table has a large number of entries, collisions will increase and insertions may fail.

- Solution is to build a table that is twice as large with a new hash function and re-insert all the items into the new hash table - called rehashing.

- Expensive operation that takes time O(n).

- When to rehash?

  - When table is half full, or

  - When an insertion fails, or

  - When table reaches a certain *load factor*

Load factor

- For separate chaining, with N = number of keys, M = number of linked lists:
  - Load factor α = N/M and α is usually greater than 1

- For closed hashing, load factor α is number of slots that are occupied and must be less than 1.
  - If table is nearly full (α is close to 1) then search can take large number of probes, or never terminate if table is full.

Hashing without collisions

- If we know all the keys *apriori* (before building the hash table), can we create a hashing function that will distribute them uniformly without collision?
    - GNU gperf (https://www.gnu.org/software/gperf/) reads a set of keywords and attempts to build a perfect hash function that does not result in any collisions in the hash table
- May not know all keys *apriori*.
    - Collision is likely to occur. Why?

GNU gperf command line options

Three different types of hash functions are generated by GNU gperf:

- Minimal-Perfect: *Perfect* property means that a table entry is located with no collisions. *Minimal* property means that the table size is the optimal size for the given set of keywords.
- Non-minimal Perfect: Have the *Perfect* property but are non-minimal.
    - faster to generate than minimal-perfect hash functions.
    - may execute faster than minimal functions when searching for values not in table.
- Near-Perfect: Allow collisions and may not have the minimal property.
    - produce smaller lookup tables.

Open Hashing C Code

```
/*
 * From Data Structures and Algorithms in C by Weiss
 */

typedef struct list_node *node_ptr;
typedef unsigned int INDEX;

struct list_node {
    int element;
    node_ptr next;
};


typedef struct list_node * LIST;
typedef node_ptr position;

struct hash_tbl {
    unsigned int table_size;
    LIST *the_lists;
};
```

Open Hashing C Code continued

```
HASH_TABLE initialize_table(unsigned int table_size) {
     HASH_TABLE H;
     int i;

     if( table_size < MIN_TABLE_SIZE) {
          perror("Table size too small");
          return NULL;
     }

     // allocate table
     H = (HASH_TABLE) malloc (sizeof(struct hash_tbl));
     if (H == NULL)
          perror("Out of space!!");

// H->table_size = next_prime((table_size);

// allocate list pointers
H->the_lists = (position *) malloc(sizeof (LIST) * H->table_size);

if (H->the_lists == NULL)
     perror("Out of space!!!");

for (i = 0; i < H->table_size; i++) {
     H->the_lists[i] = (LIST) malloc (sizeof(struct list_node));

     if (H->the_lists[i] == NULL)
          perror("Out of space!!!");
     else
          H->the_lists[i]->next = NULL;
}
return H;
}
```

```
INDEX hash(char *key, unsigned int
H_SIZE) {
    unsigned int hash_val = 0;

    while (*key != '\0')
        hash_val = (hash_val << 5)
+ *key++;

    return (hash_val % H_SIZE);
}
```

Open Hashing C Code continued

```
position find(int key, HASH_TABLE H)
      position p;
    LIST L;

      L = H->the_lists[hash(key, H->table_size)];
      p = L->next;

      while ( (p != NULL && p->element != key))
          p = p->next;

      return p;
}
```

Open Hashing C Code continued

```
void insert(int key, HASH_TABLE H) {
    position pos, new_cell;

    LIST L;
    pos = find(key, H);

    if (pos == NULL) {
        new_cell = (position) malloc(sizeof(struct list_node));

        if (new_cell == NULL)
            perror("Out of space!!!");

        else {
            L = H->the_lists[hash(key, H->table_size)];
            new_cell->next = L->next;
            new_cell->element = key;
            L->next = new_cell;
        }
    }
}
```

Applications

- Used in symbol tables by compilers to keep track of declared variables.
- Graph theory problems where nodes have string based names instead of numbers.
- Spelling checkers
- Blockchain hashing - the transactions are taken as input and run through a hashing algorithm (Bitcoin uses SHA-256) which gives an output of a fixed length.
- Many others…