CPSC 319 – Hashtables

UNIVERSITY OF CALGARY

Tables: Rows & Columns of Information

- A *table* has several *fields* (types of information)
  - A telephone book may have fields **name**, **address**, **phone number**
  - A user account table may have fields **user id**, **password**, **home folder**

Tables: rows & columns of information

- To find an *entry* in the table, you only need know the contents of one of the fields (not all of them).

- This field is the *key*

  - In a telephone book, the key is usually **name**
  - In a user account table, the key is usually **user id**

Tables: rows & columns of information

- Ideally, a key *uniquely identifies* an entry
    - If the key is **name** and no two entries in the telephone book have the same name, the key uniquely identifies the entries

The Table ADT: operations

- **insert**: given a key and an entry, inserts the entry into the table
- **find**: given a key, finds the entry associated with the key
- **remove**: given a key, finds the entry associated with the key, *and* removes it

Also:

- **getIterator**: returns an iterator, which visits each of the entries one by one (the order may or may not be defined)

etc.

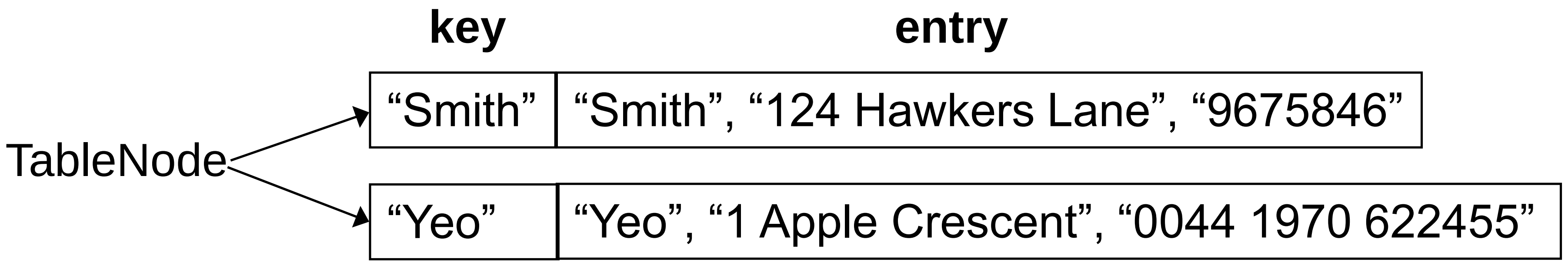
How should we implement a table?

Our choice of representation for the Table ADT depends on the answers to the following

- How often are entries inserted and removed?
- How many of the possible key values are likely to be used?
- What is the likely pattern of searching for keys?
  - e.g. Will most of the accesses be to just one or two key values?
- Is the table small enough to fit into memory?
- How long will the table exist?

TableNode: a key and its entry

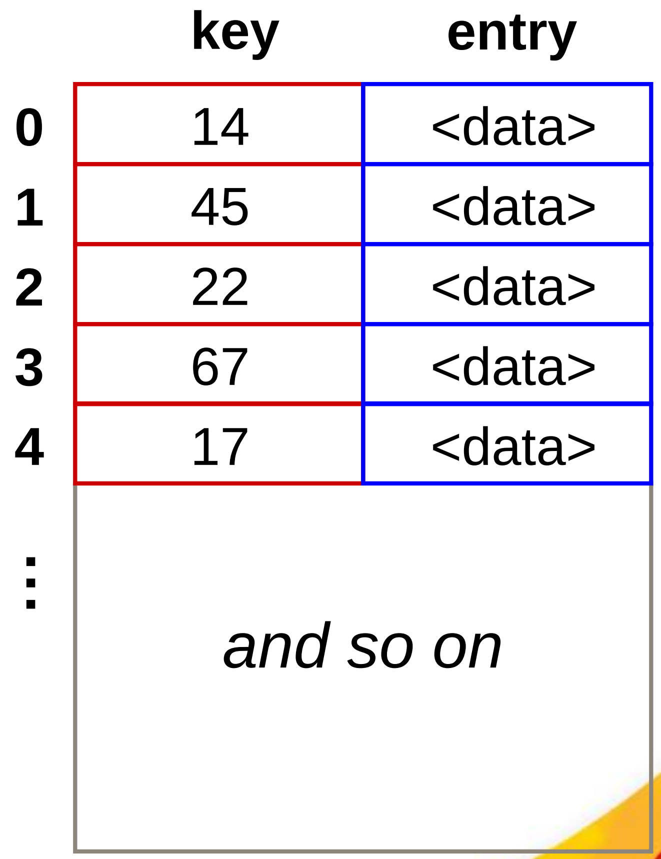
For searching purposes, it is best to store the key and the entry separately (even though the key’s value may be inside the entry)


key
entry
TableNode
“Smith”
“Smith”, “124 Hawkers Lane”, “9675846”
“Yeo”
“Yeo”, “1 Apple Crescent”, “0044 1970 622455”

Implementation 1: Unsorted Sequential Array

- An array in which TableNodes are stored consecutively in *any* order

- **insert**: add to back of array; O(1)
- **find**: search through the keys one at a time, potentially all of the keys; O(*n*)
- **remove**: find + replace removed node with last node; O(*n*)



	key	entry
0	14	<data>
1	45	<data>
2	22	<data>
3	67	<data>
4	17	<data>
⋮	*and so on*	

Implementation 2:
Sorted Sequential Array

- An array in which TableNodes are stored consecutively, *sorted* by key

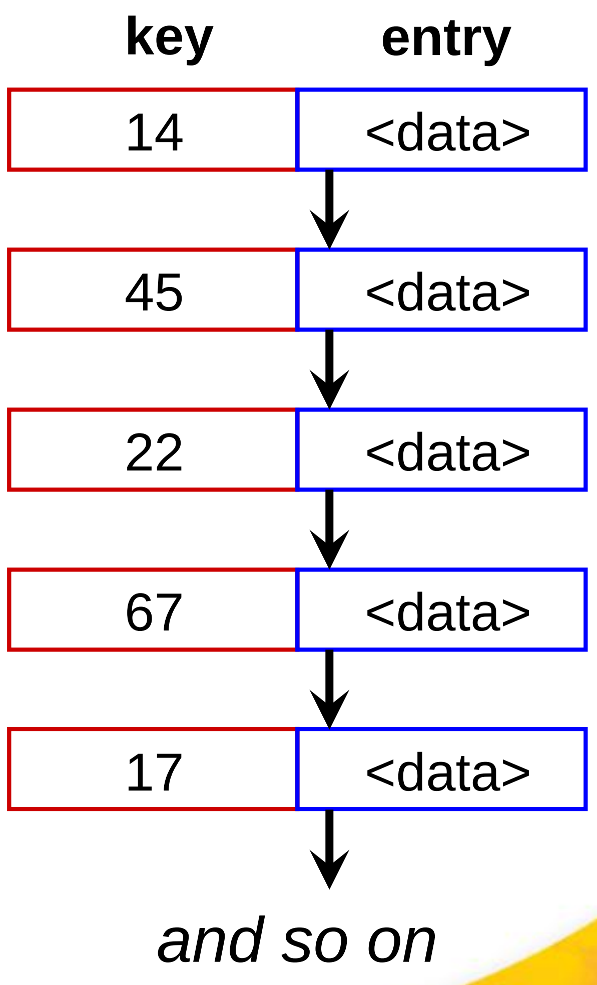
- **insert**: add in sorted order; O(*n*)
- **find**: binary chop; O(log *n*)
- **remove**: find, remove node and shuffle down; O(*n*)

	key	**entry**
0	15	<data>
1	17	<data>
2	22	<data>
3	45	<data>
4	67	<data>
⋮	*and so on*	

We can use binary search because the array elements are sorted

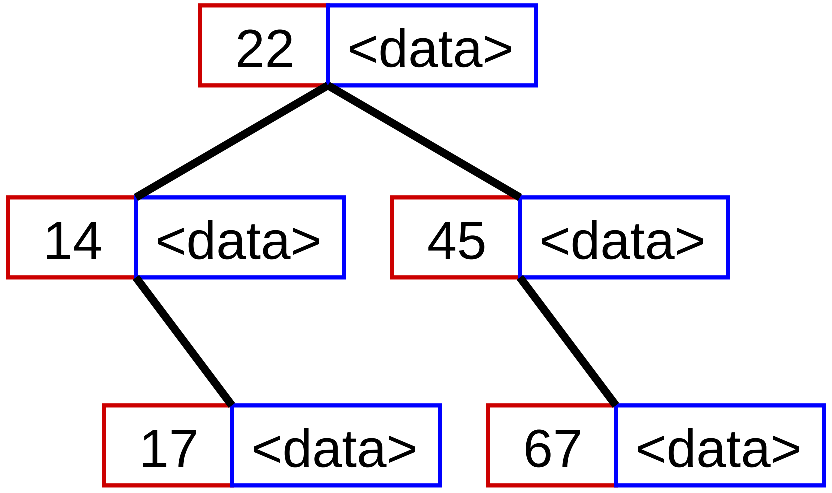
Implementation 3: Linked List (Unsorted or Sorted)

- TableNodes are again stored consecutively
- **insert**: add to front; O(1)
  or O(n) for a sorted list
- **find**: search through potentially all the keys, one at a time; O(*n*)
  still O(n) for a sorted list
- **remove**: find, remove using pointer alterations; O(*n*)



Implementation 4: AVL Tree

- An AVL tree, ordered by key
- **insert**: a standard insert; O(log *n*)
- **find**: a standard find (without removing, of course); O(log *n*)
- **remove**: a standard remove; O(log *n*)



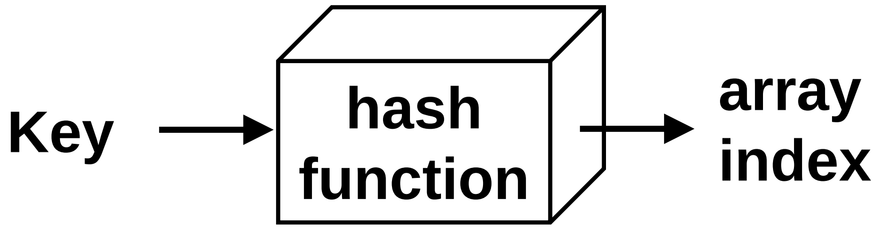
and so on

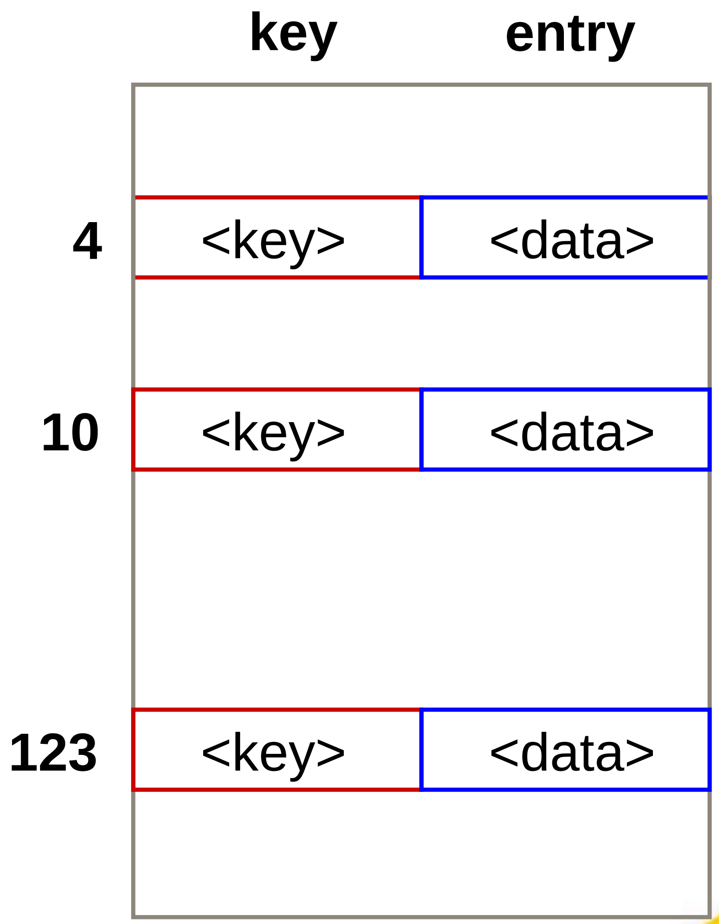
O(log *n*) is very good…

…but O(1) would be even better!

Implementation 5: Hashing

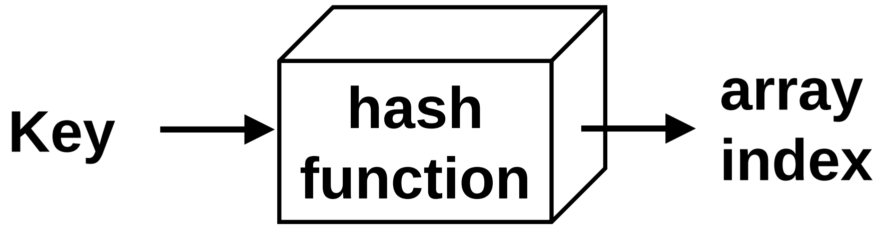
- An array in which TableNodes are ***not*** stored consecutively - their place of storage is calculated using the key and a *hash function*

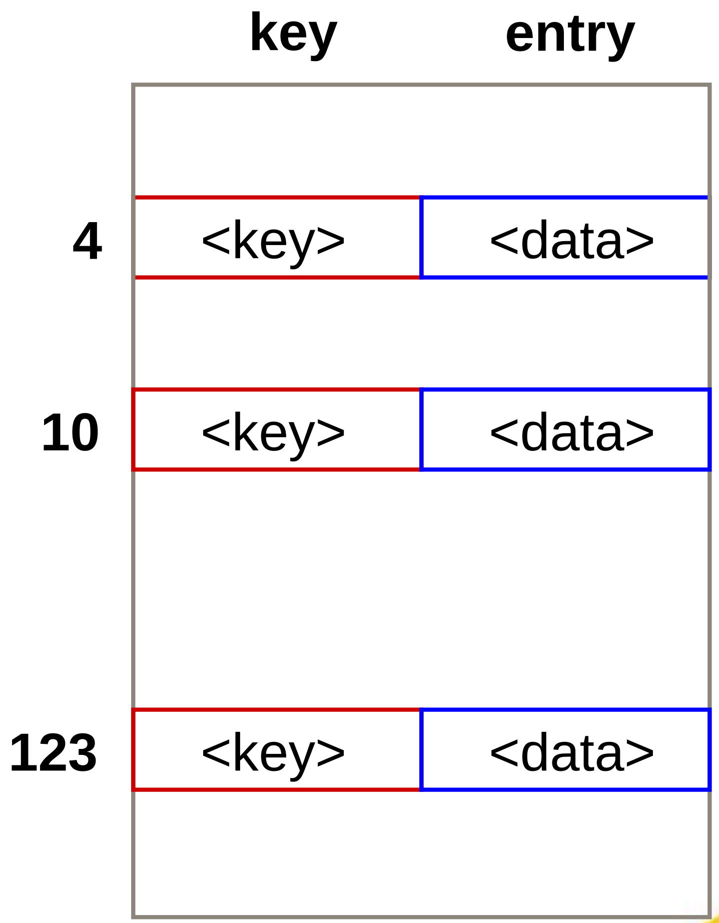




Hash values → mappings of the keys as the indexes in the hash table

Implementation 5: Hashing



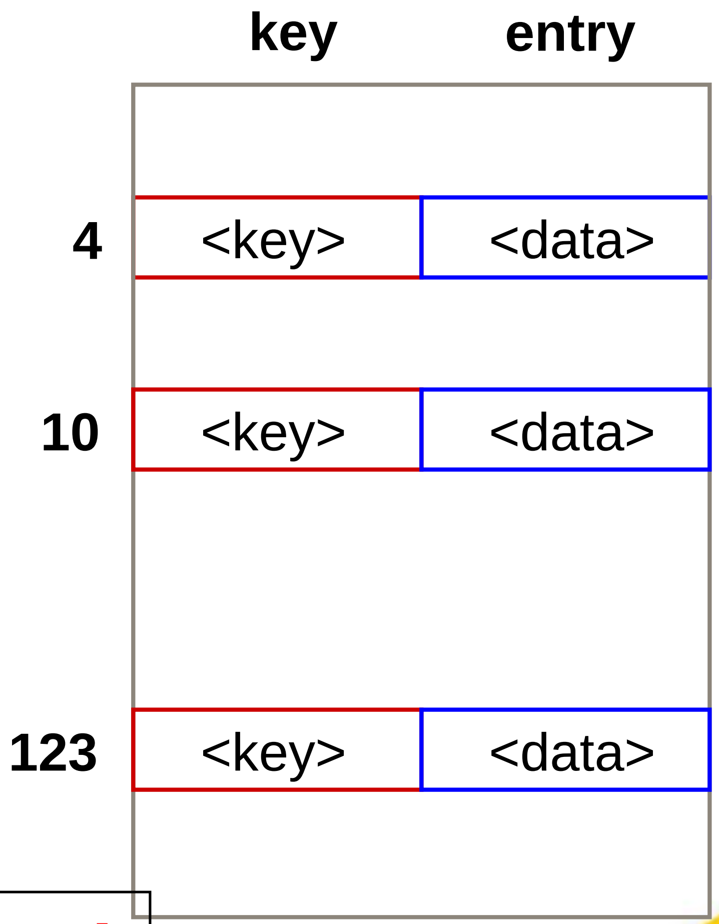


- *Hashed key*: the result of applying a hash function to a key
- Keys and entries are scattered throughout the array

Implementation 5: Hashing

- An array in which TableNodes are ***not*** stored consecutively - their place of storage is calculated using the key and a *hash function*

- **insert**: calculate place of storage, insert TableNode; O(1)
- **find**: calculate place of storage, retrieve entry; O(1)
- **remove**: calculate place of storage, set it to null; O(1)



All are O(1) ! Under good hash conditions!

Applications of Hashing

- Compilers use hash tables to keep track of declared variables

- Game playing programs use hash tables to store seen positions, thereby saving computation time if the position is encountered again

When are other representations more suitable than hashing?

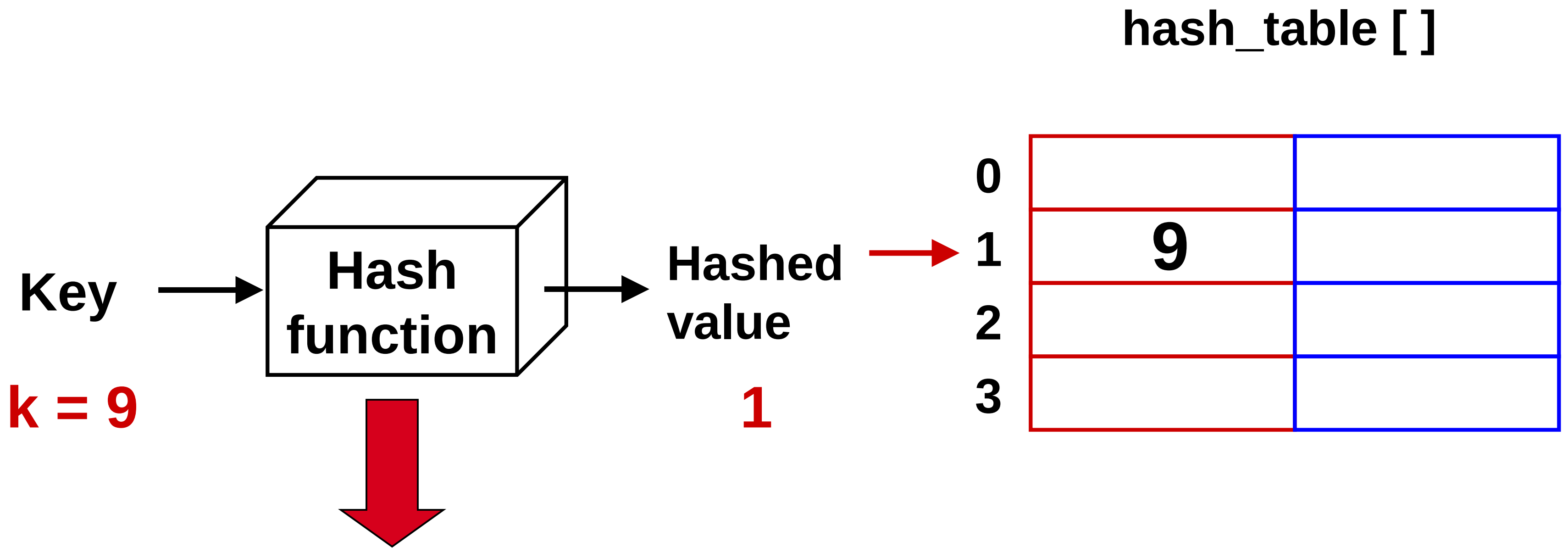
- Hash tables are very good if there is a need for many searches in a reasonably stable table

- Hash tables are not so good if there are **many insertions** and **deletions**, or if table **traversals** are needed — in this case, AVL trees are better

When are other representations more suitable than hashing?

- Also, hashing is very slow for any operations which require the entries to be **sorted**
  - e.g. Find the minimum key

Inserting a Key in a Hash Table



Modulus hash function for integers

Modulus_Hash_Func (k) = k % 4

Inserting a Key in a Hash Table

- In a hash table with a single array table (single slot bucket), two different keys may be hashed to the !!!SAME!!! hash value
    - Two different k1 and k2
    - Hash (k1) == Hash (k2)
- This is called COLLISION
- Example:
    - k1 = 9, k2 = 21
    - Hash (9) = 9 % 4 = 1
    - Hash (21) = 21 % 4 = 1

Three Factors Affecting the Performance of Hashing

(1) The hash function

- Ideally, it should distribute keys and entries evenly throughout the table
- It should minimize *collisions*, where the position given by the hash function is already occupied

Three Factors Affecting the Performance of Hashing

(2) The collision resolution strategy

- *Separate chaining*: chain together several keys/entries in each position
- *Open addressing*: store the key/entry in a different position

Three Factors Affecting the Performance of Hashing

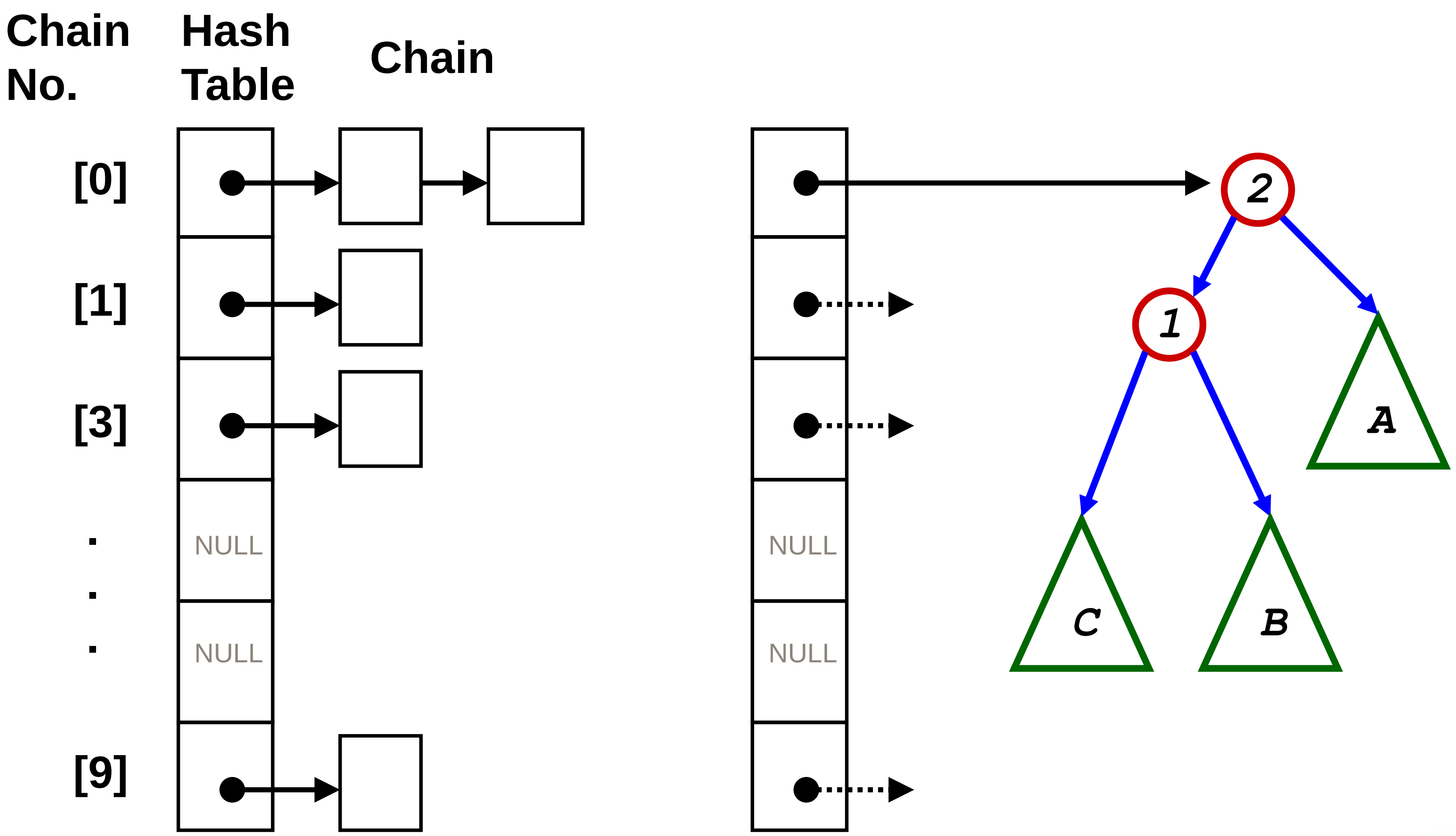
(3) The size of the table

- Too big will waste memory; too small will increase collisions and may eventually force *rehashing* (copying into a larger table)
- Should be appropriate for the hash function used

Hash Table Implementation

- **Hash table with CHAINING**
  - Array is used for the hash table and each bucket is singly or doubly linked list or even a tree of any variable number of slots (called *nodes*)
  - Each chain node is large enough to contain one data element (record)

Hash Tables with Chaining



Choosing a Hash Function: Turning a Key into a Table Position

- **Truncation**
  - Ignore part of the key and use the rest as the array index (converting non-numeric parts)
  - A fast technique, but check for an even distribution throughout the table

Choosing a Hash Function: Turning a Key into a Table Position

- **Folding**
  - Partition the key into several parts and then combine them in any convenient way
  - Unlike truncation, uses information from the whole key

Choosing a Hash Function: Turning a Key into a Table Position

- **Modular arithmetic**
  (used by truncation & folding, and on its own)
  - To keep the calculated table position within the table, divide the position by the size of the table, and take the remainder as the new position

Examples of Hash Functions (1)

- **Truncation**
  - Ignore part of the key and use the rest as the array index (converting non-numeric parts)
  - A fast technique, but check for an even distribution throughout the table

If students have an 9-digit identification number, take the last 3 digits as the table position

- e.g. 925371622 becomes 622

Examples of Hash Functions (1)

- **Folding**
  - Partition the key into several parts and then combine them in any convenient way
  - Unlike truncation, uses information from the whole key

Split a 9-digit number into three 3-digit numbers, and add them

- e.g. 925371622 becomes 925 + 371 + 622 = 1923

Examples of Hash Functions (1)

- **Modular arithmetic**
  - To keep the calculated table position within the table, divide the position by the size of the table, and take the remainder as the new position

If the table size is 1000, the first example always keeps within the table range

- e.g. 1923 mod 1000 = 923 (in Java: 1923 % 1000)

Examples of Hash Functions (2)

- Using a telephone number as a key
  - The area code is not random, so will not spread the keys/entries evenly through the table (many collisions)
  - The last 3-digits are more random

Examples of Hash Functions (2)

- **Using a name as a key**
  - Use full name rather than surname (surname not particularly random)
  - Assign numbers to the characters (e.g. a = 1, b = 2)
  - **Strategy 1:** Add the resulting numbers. Bad for large table size.

Examples of Hash Functions (2)

- **Using a name as a key**
  - Use full name rather than surname (surname not particularly random)
  - Assign numbers to the characters (e.g. a = 1, b = 2)
  - **Strategy 2:** Call the number of possible characters *c* (e.g. *c* = 54 for alphabet in upper and lower case, plus space and hyphen). Then multiply each character in the name by increasing powers of *c*, and add together.

Hashing example: a fruit shop

- 10 stock details, 10 table positions
- Stock numbers are between 0 and 1000
- Use hash function: stock no. / 100
- What if we now insert stock no. 350?
  - Position 3 is occupied: there is a collision
- Collision resolution strategy:
  - insert in the next free position (linear probing)

- Given a stock number, we find stock by using the hash function again, and use the collision resolution strategy if necessary

	key	entry
0	85	85, apples
1		
2		
3	323	323, guava
4	462	462, pears
5	350	350, oranges
6		
7		
8		
9	912	912, papaya

Rehashing: Enlarging the Table

- To *rehash*:
  - Create a new table of double the size (adjusting until it is again prime)
  - Transfer the entries in the old table to the new table, by recomputing their positions (using the hash function)

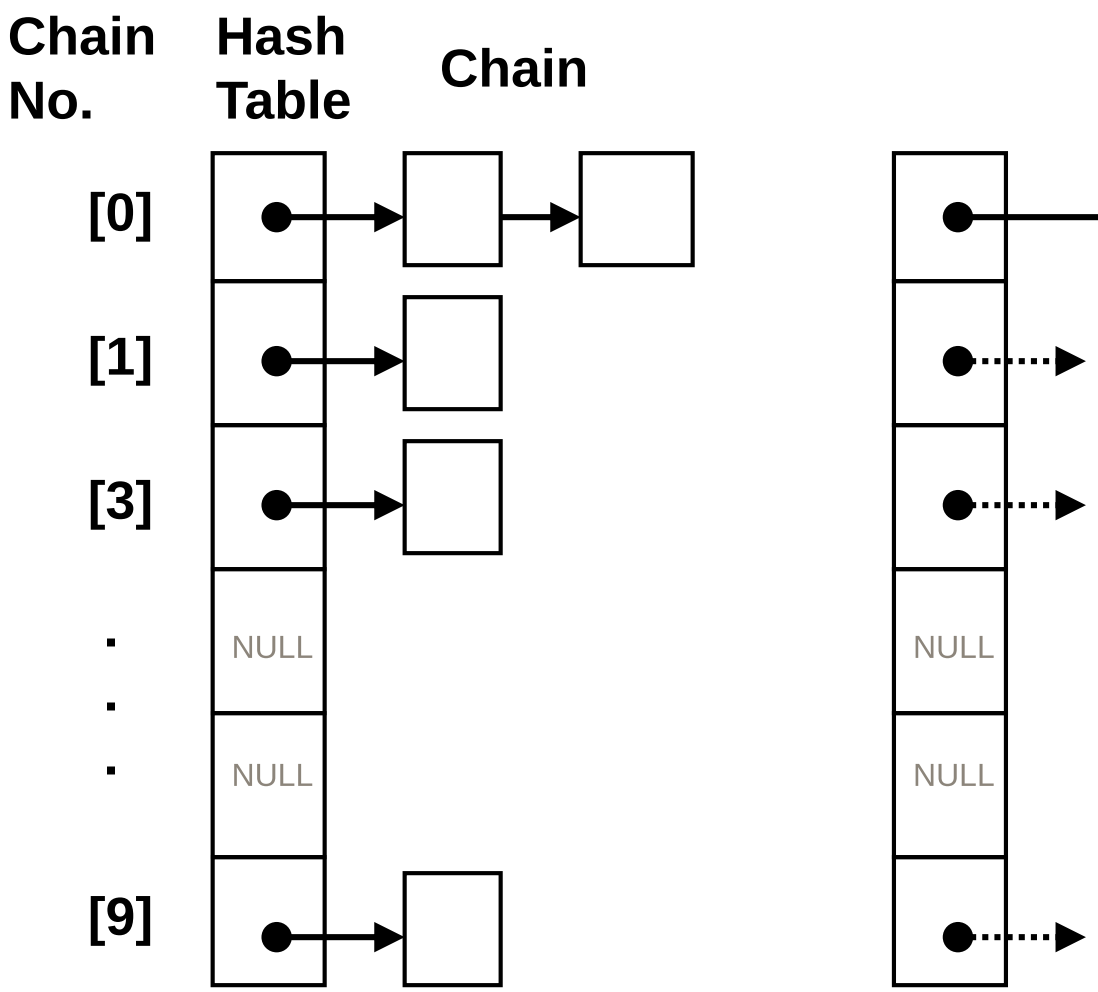
Rehashing: Enlarging the Table

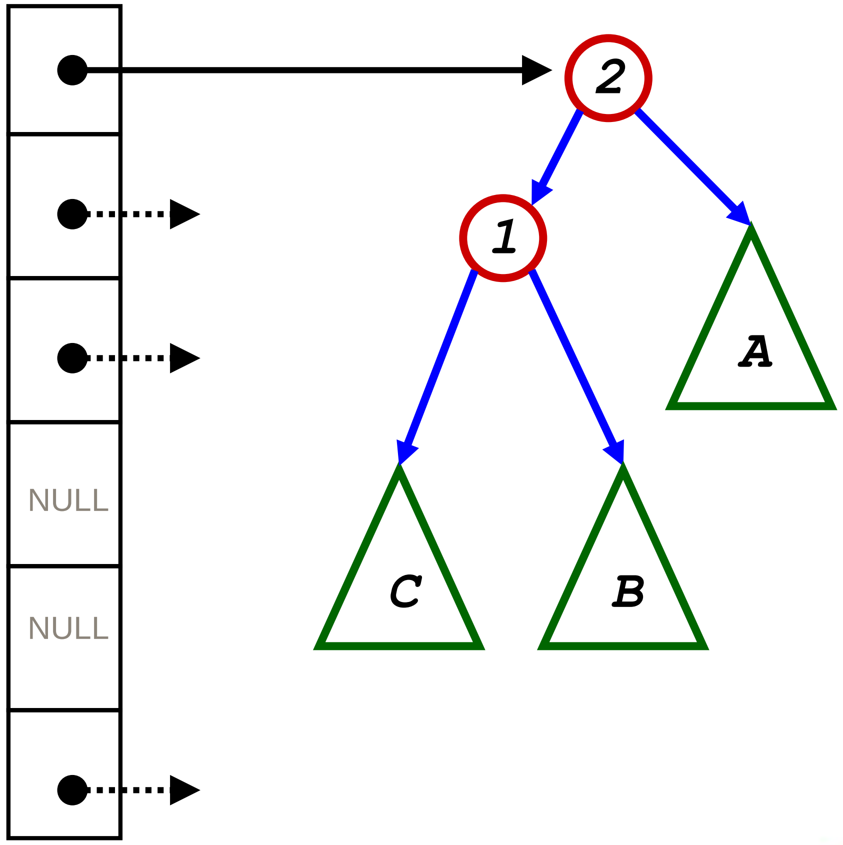
- When should we rehash?
  - When the table is completely full

Rehashing: Enlarging the Table

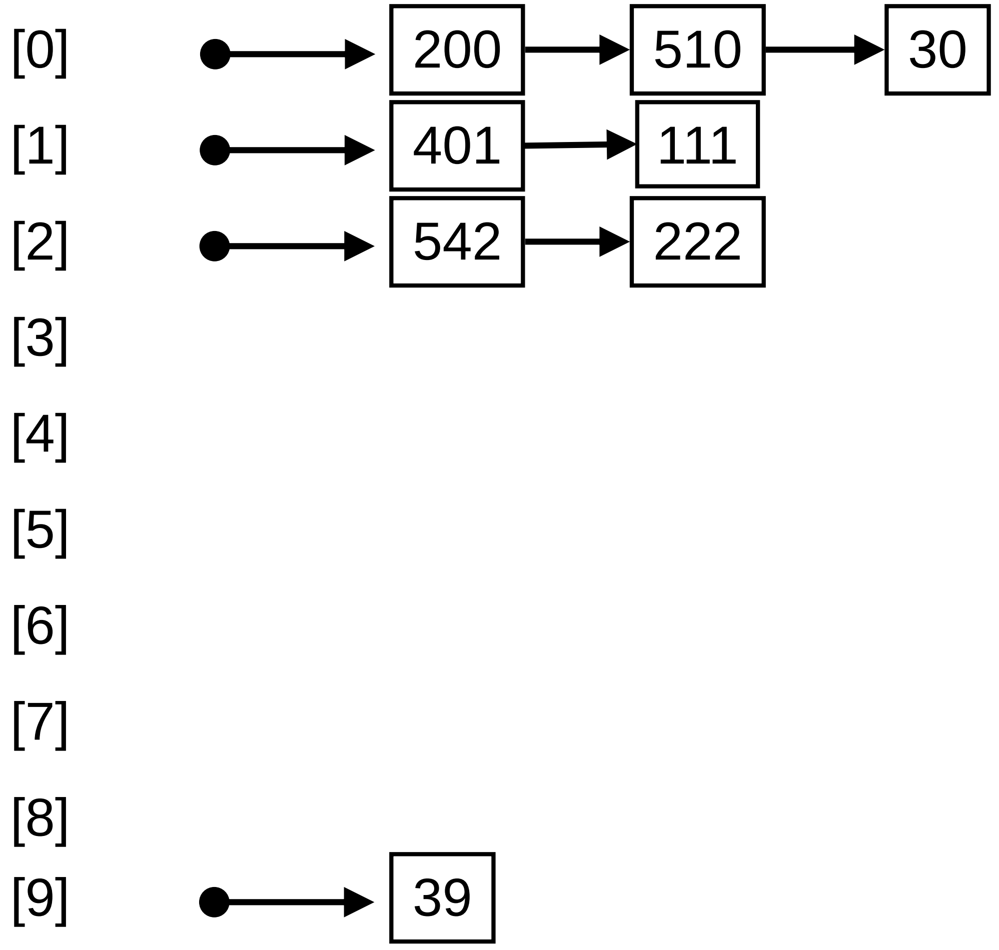
- Why double the size?
  - If *n* is the number of elements in the table, there must have been *n*/2 insertions before the previous rehash (if rehashing done when table full)
  - So by making the table size 2*n*, a constant cost is added to each insertion

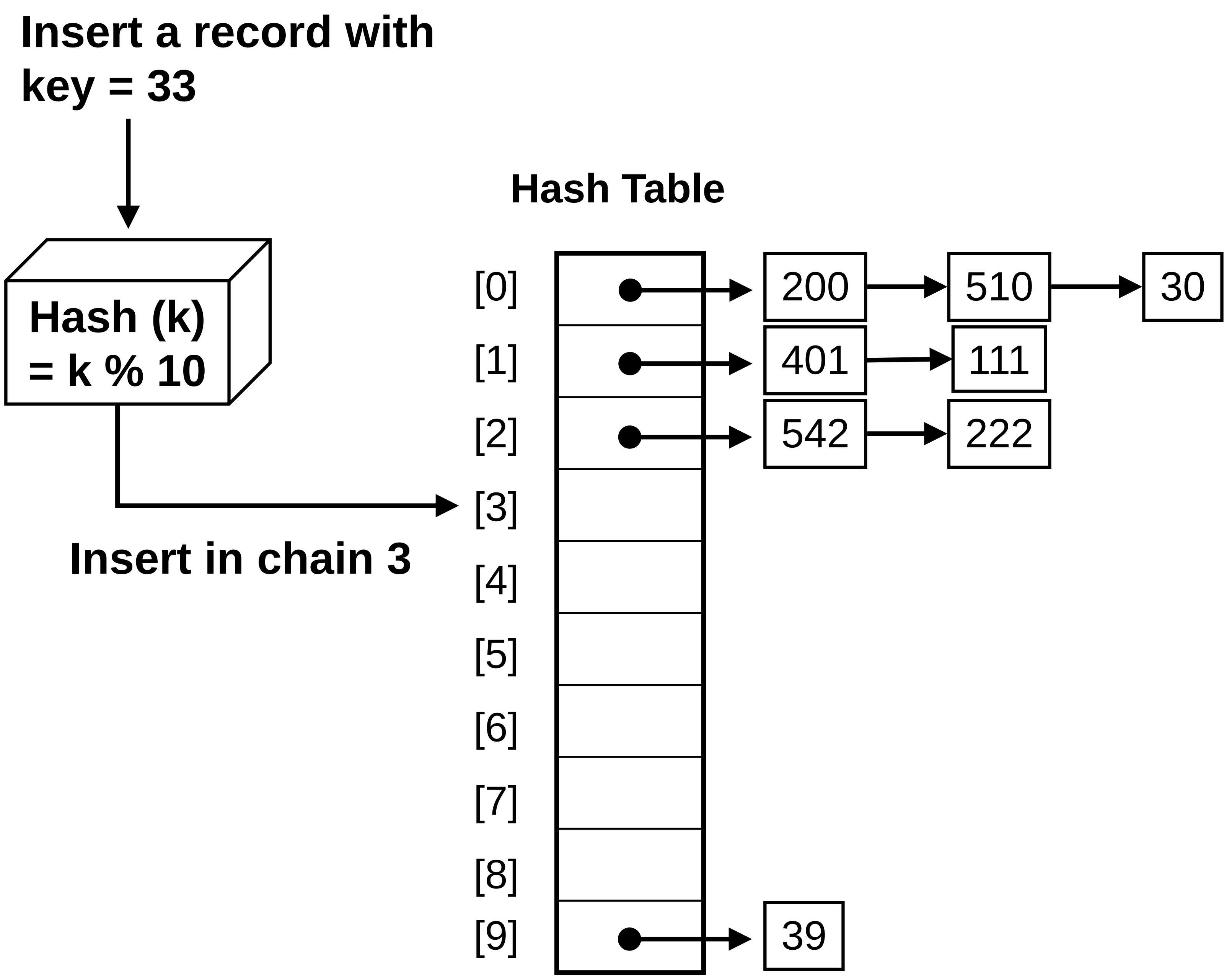
Hash Tables with Chaining

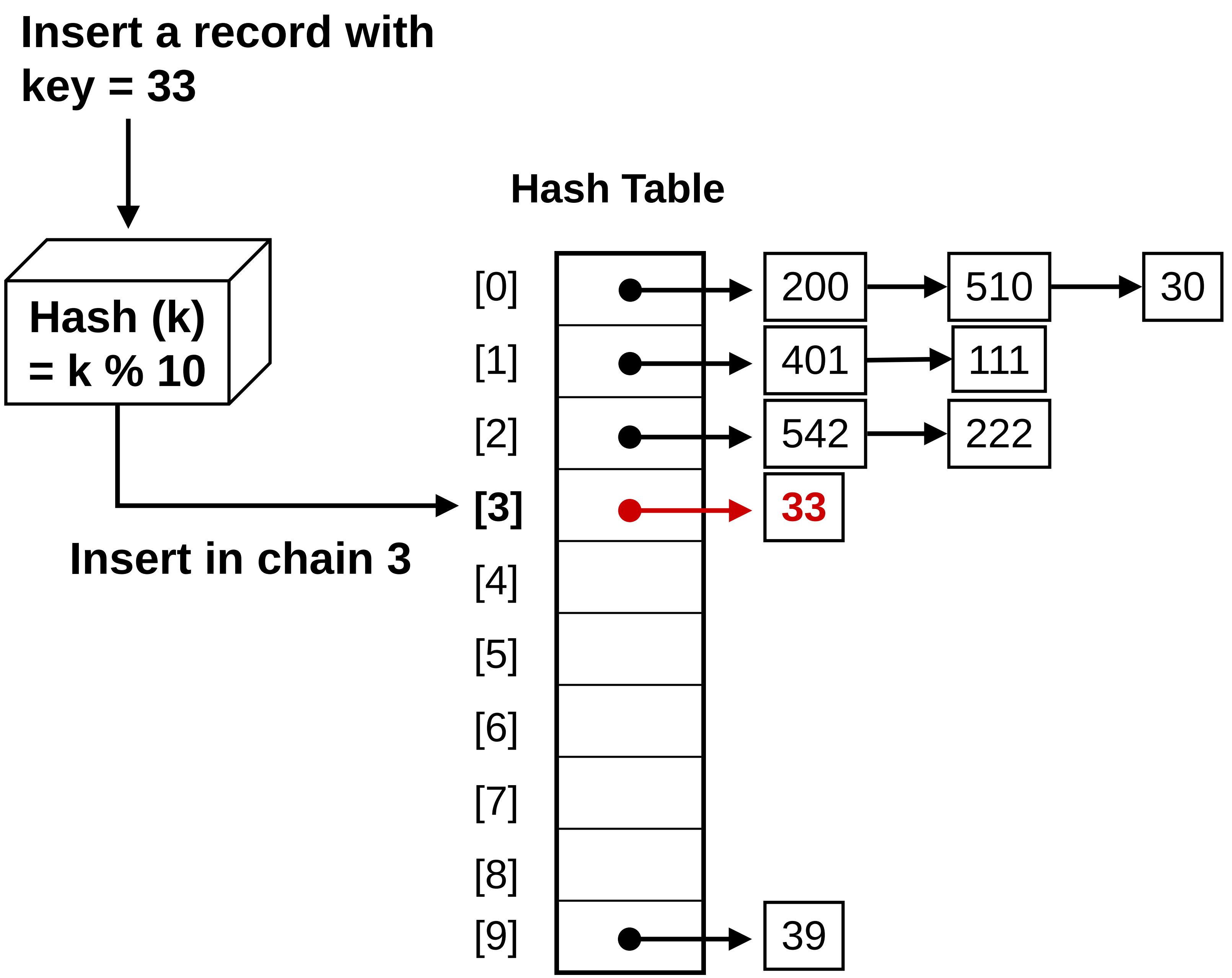


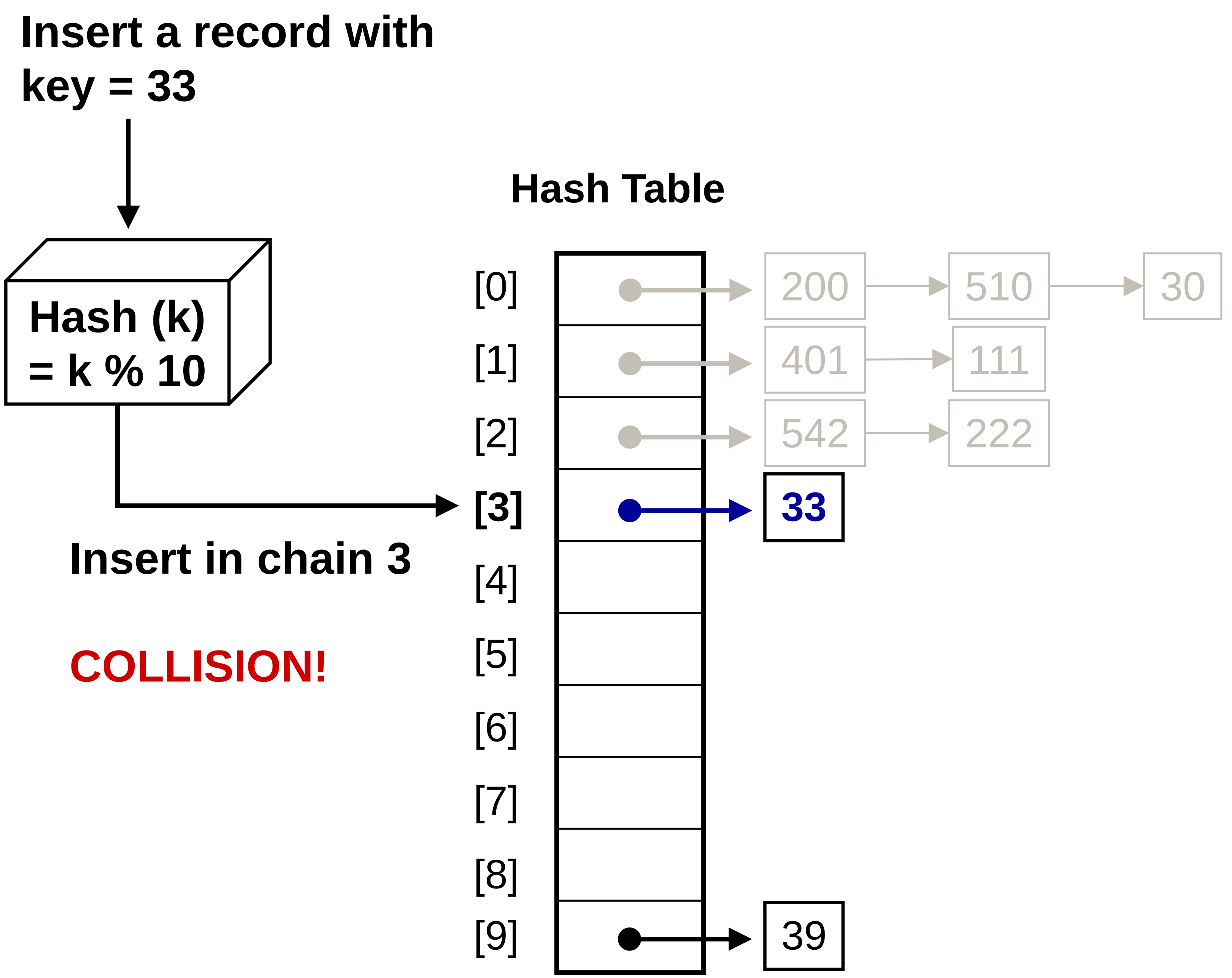


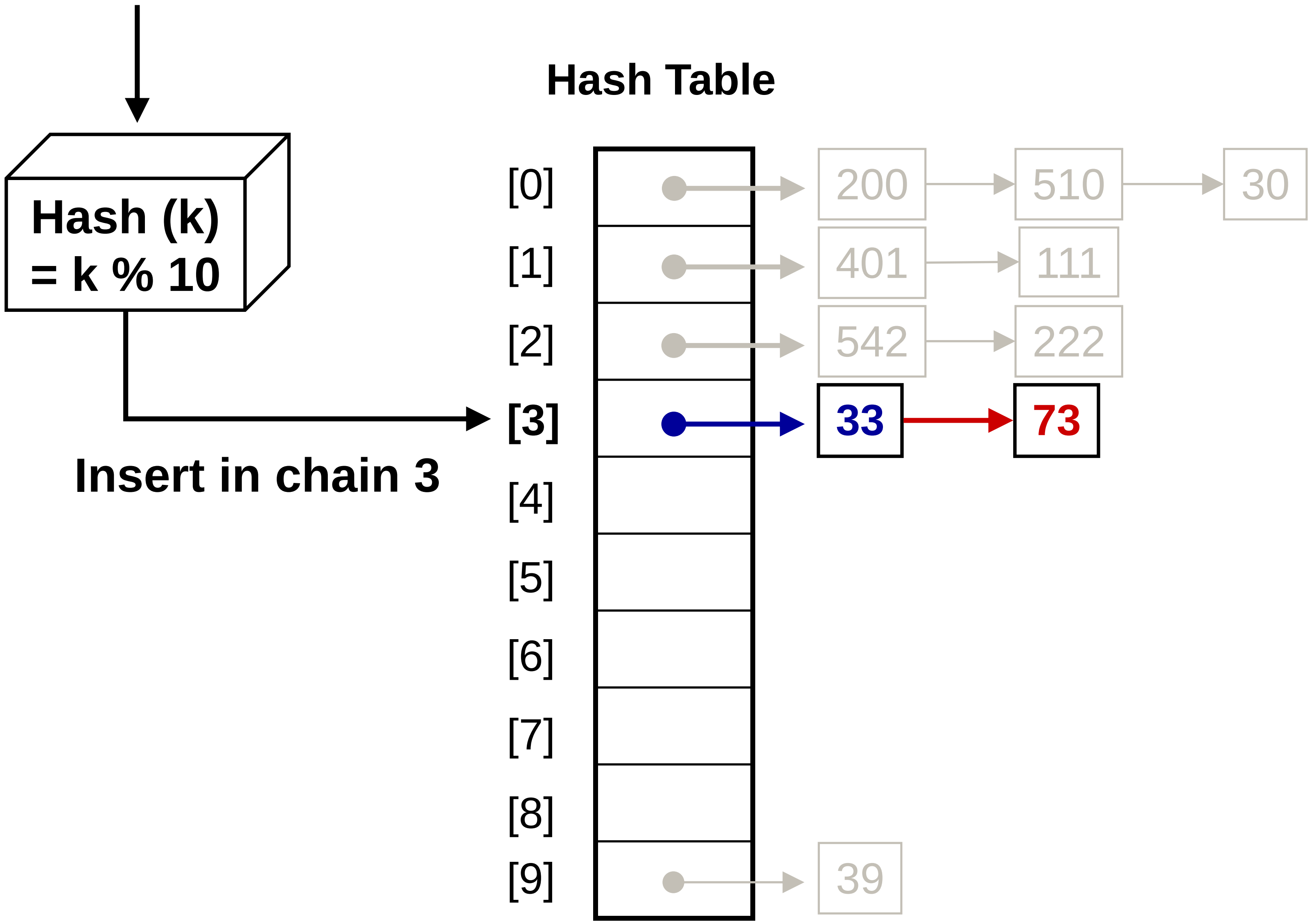
Hash Table



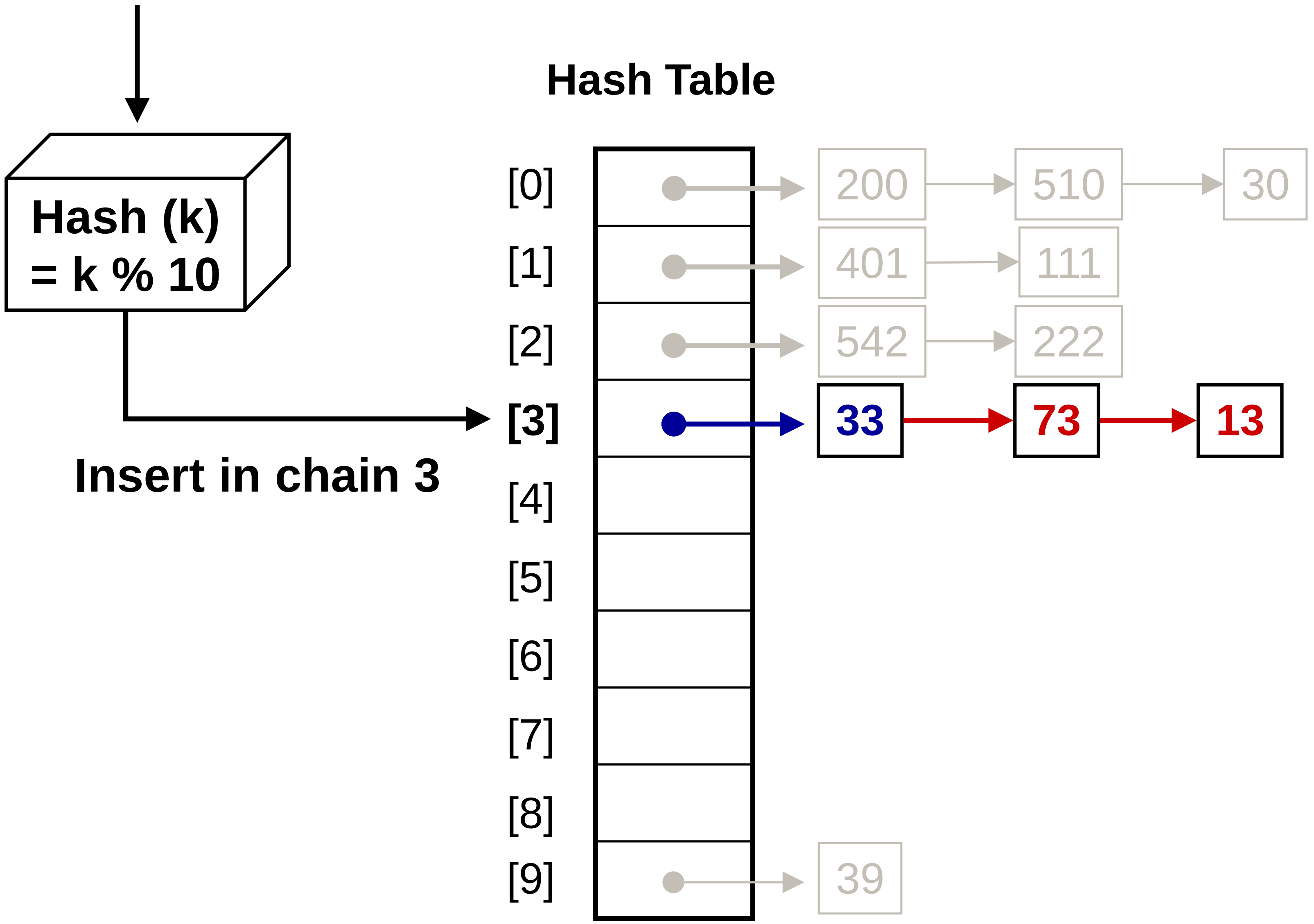

Insert a record with key = 33
Hash (k) = k % 10
Insert in chain 3
Hash Table
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
200
510
30
401
111
542
222
39

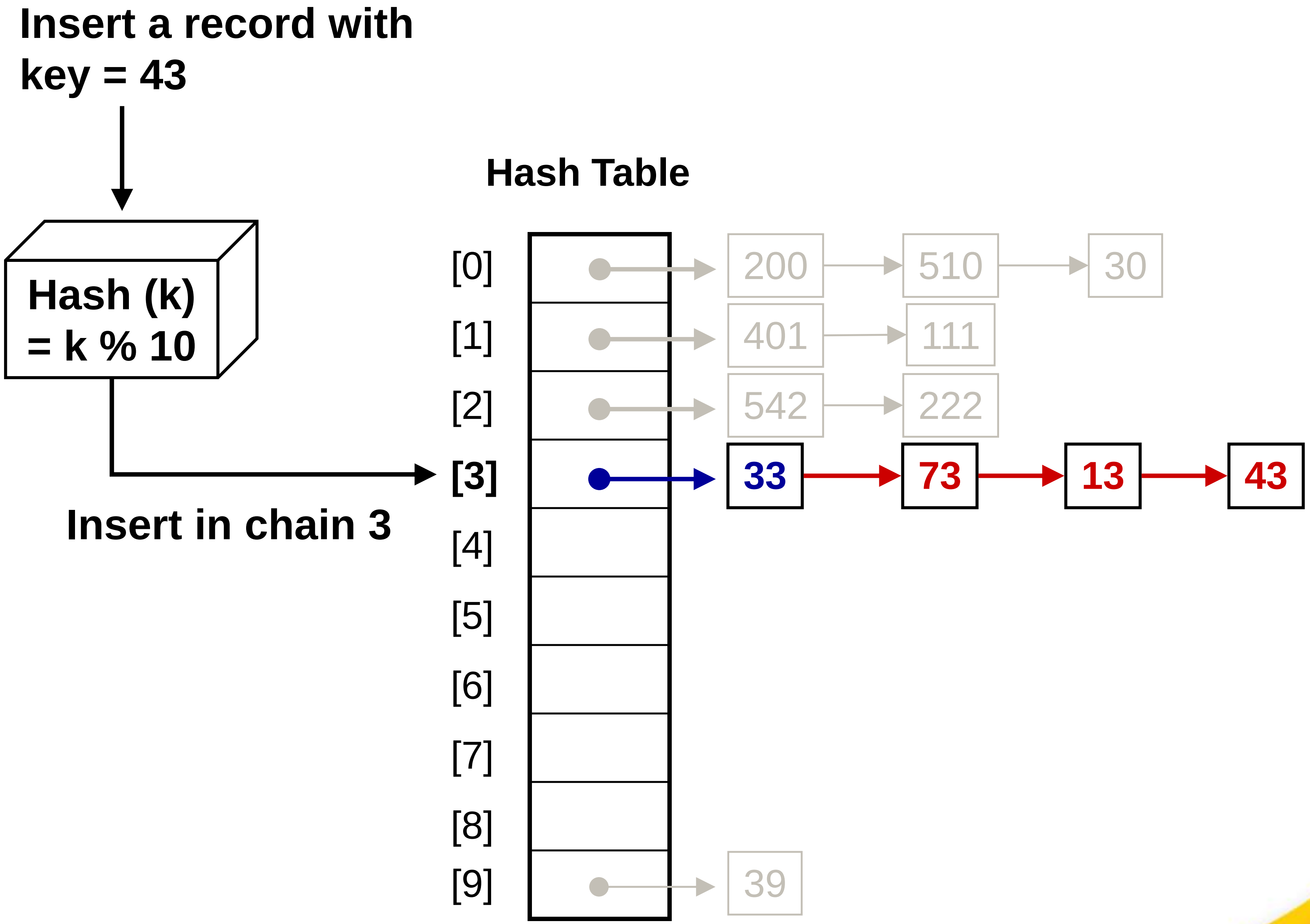

Insert a record with key = 33
Hash (k) = k % 10
Insert in chain 3
Hash Table
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
200
510
30
401
111
542
222
33
39


Insert a record with
key = 33
Hash (k)
= k % 10
Insert in chain 3
COLLISION!
Hash Table
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
200
510
30
401
111
542
222
33
39



Insert a record with key = 73
Hash (k) = k % 10
Insert in chain 3
Hash Table
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
200
510
30
401
111
542
222
33
73
39


Insert a record with key = 13
Hash (k) = k % 10
Insert in chain 3
Hash Table
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
200
510
30
401
111
542
222
33
73
13
39


Insert a record with key = 43
Hash (k) = k % 10
Insert in chain 3
Hash Table
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
200
510
30
401
111
542
222
33
73
13
43
39

Hash Function

- Sometimes the hash function is suggested by the structure of the key.
- In other cases, the best hash function can only be determined by experimentation.
- Perfect minimal hash functions do exist.

Hash Function

- **Hash functions which use all of the key are almost always better than those which use only some of the key.**

- **When only portions are used**, information is lost and therefore the number of possibilities for the final key are reduced.

- Similarly, if we work with only integers or characters, there is a limit to what can be done.

Hash Function

- A **good hash function** gives an average-case lookup that is a small constant, independent of the number of search keys.

- **We hope records are distributed uniformly among the buckets.**

Thank You

- Have Great Summer