# DATA STRUCTURES & ALGORITHMS

## DICTIONARIES AND HASHING

issntt@nus.edu.sg

# What have we had so far?

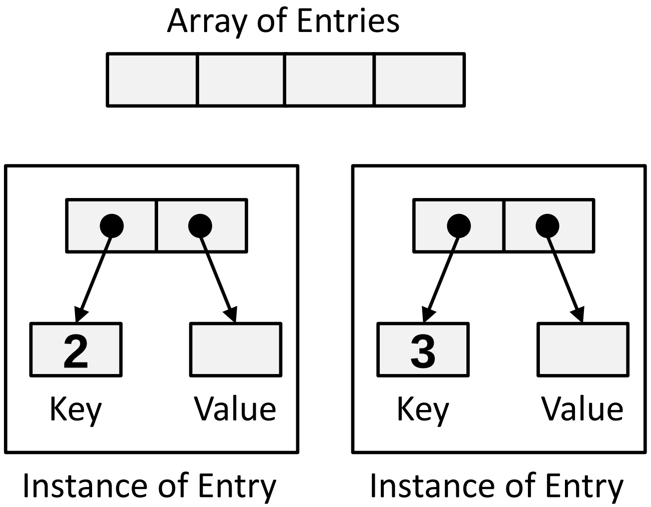
The following table summarized the worst-case efficiencies of some dictionary operations

| Operations | Array implementation | Linked List implementation |
|---|---|---|
| Add | O(n) | O(n) |
| Get | O(n) | O(n) |
| Remove | O(n) | O(n) |
| Update | O(n) | O(n) |

In many apps where get - retrieving the respective value from a key - is the primary operation. Is there any way to make it faster?

# Question

- Let’s solve a simpler problem: **only integers** can be used as **keys**
- Can we find a way to make **retrieving a value given a key very fast**?
  - *Hint*: **retrieving an array element given an index is very fast**

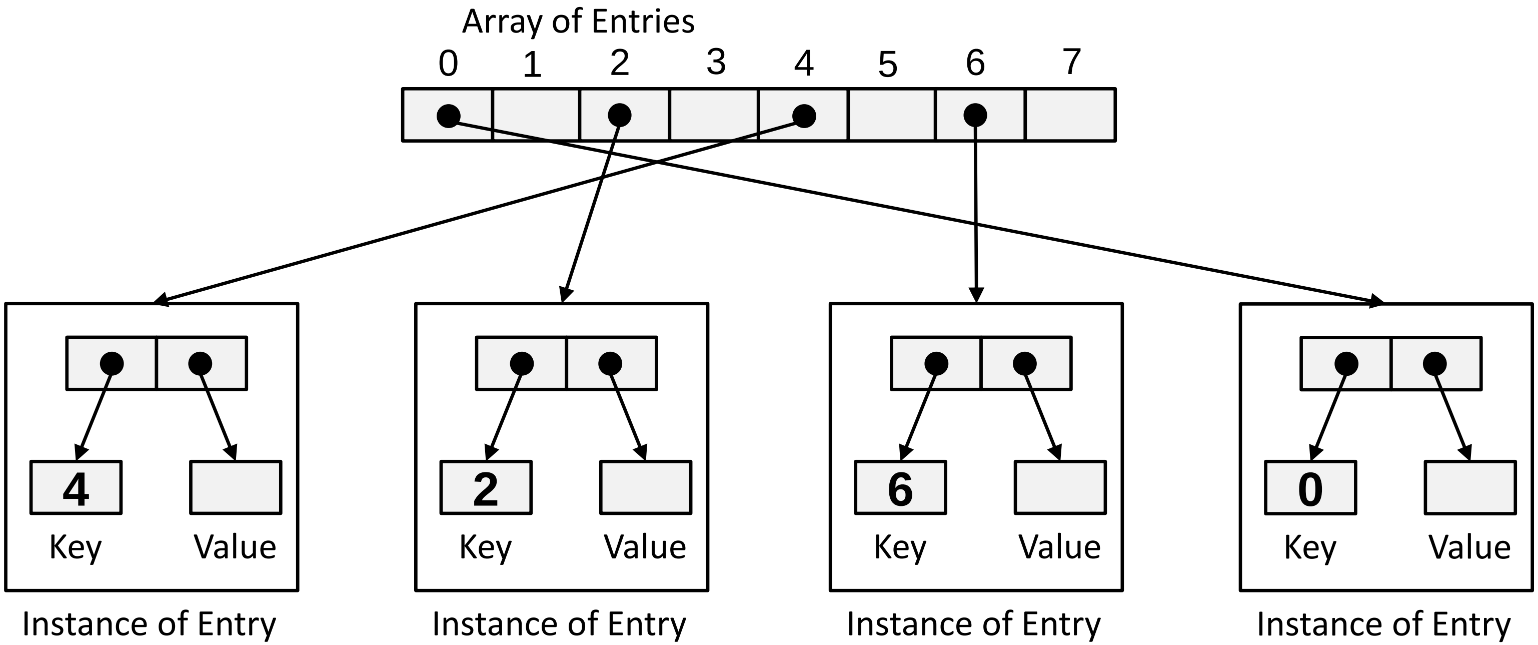


# Outline

- **Implementing Dictionary ADT**
  - **Using Direct Addressing technique**
    - **Issues with Direct Addressing**
  - Using Hash Tables

# Direct Addressing

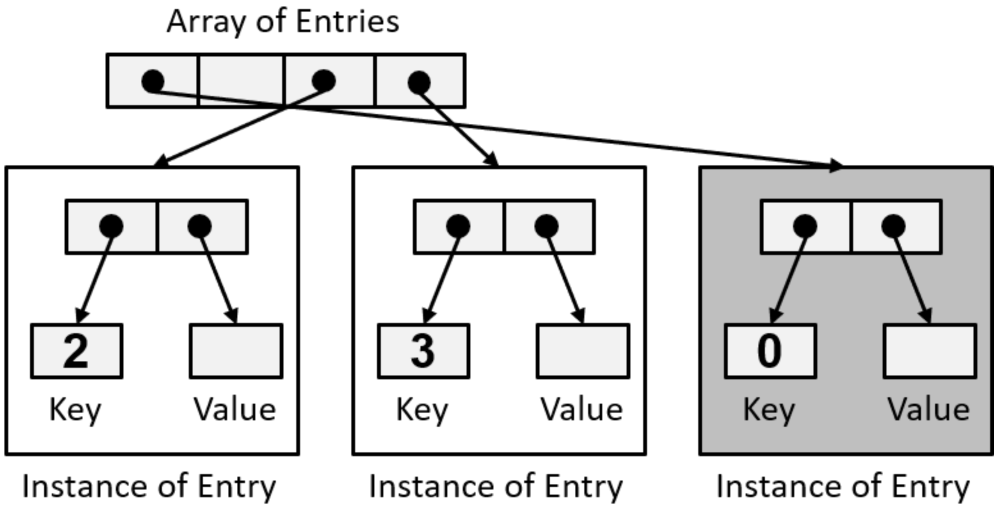
When storing an entry in an array, instead of at index 0, 1, 2..., **use the key as index**



# Implementing Operation Add

*Algorithm for Add(key, value)*
*// Adds a new key-value entry to the dictionary. If key already exists,*
*// throws an ArgumentException*

```
Entry entry = arr[key]
if (entry != null)
   Throw an ArgumentException
else
   arr[key] = entry
```



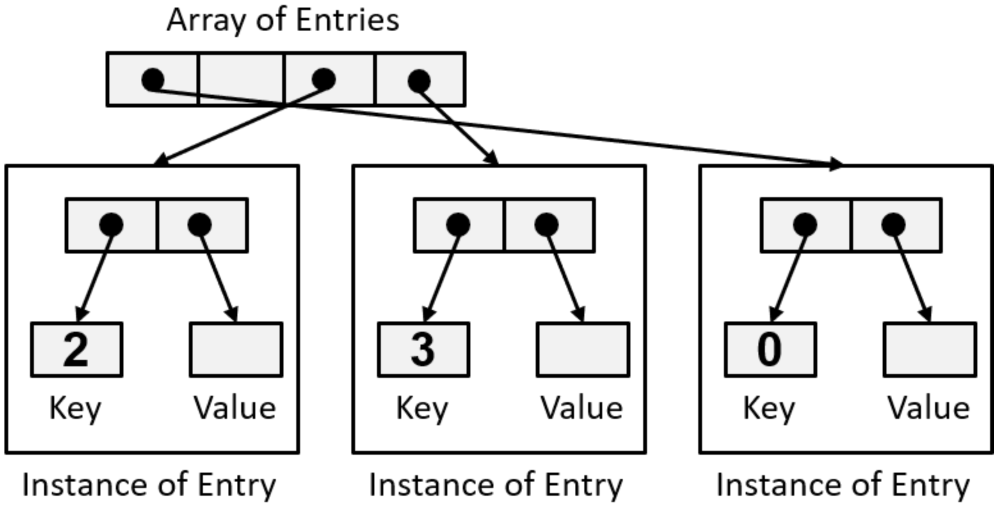
What is the running time of this algorithm?

# Implementing Operation Get

*Algorithm for Get(key)*

*// Retrieve the respective value for the given key. If key does not exist, return null*

```
Entry entry = arr[key]
if (entry == null)
   return null
else
   return entry.Value
```



What is the running time of this algorithm?

So, **given** the array and the **key**, **we will always get** the respective entry, then the respective **value**. From now, to simplify, we only show the key.

# Quiz

Using direct addressing, add entries with the following keys into the given arrays

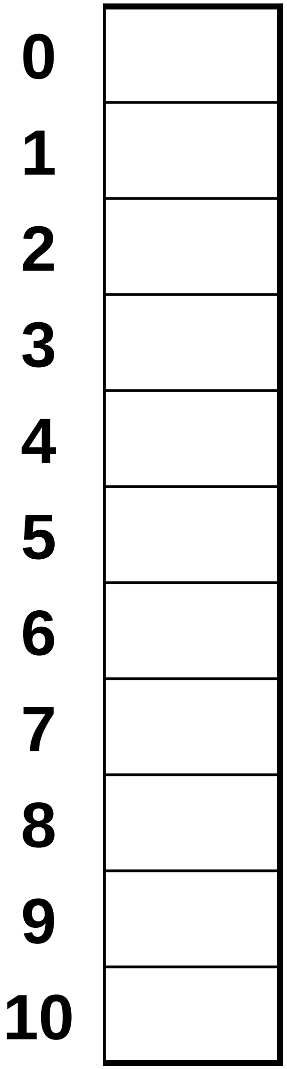
- Array size = 11
- **Add**: 3, 1, 5, 9

| | |
|---|---|
| 0 | |
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | |
| 7 | |
| 8 | |
| 9 | |
| 10 | |

# Quiz

Using direct addressing, add entries with the following keys into the given arrays

- Array size = 11
- **Add**: 4, 7, 2, 10, 18



# Question

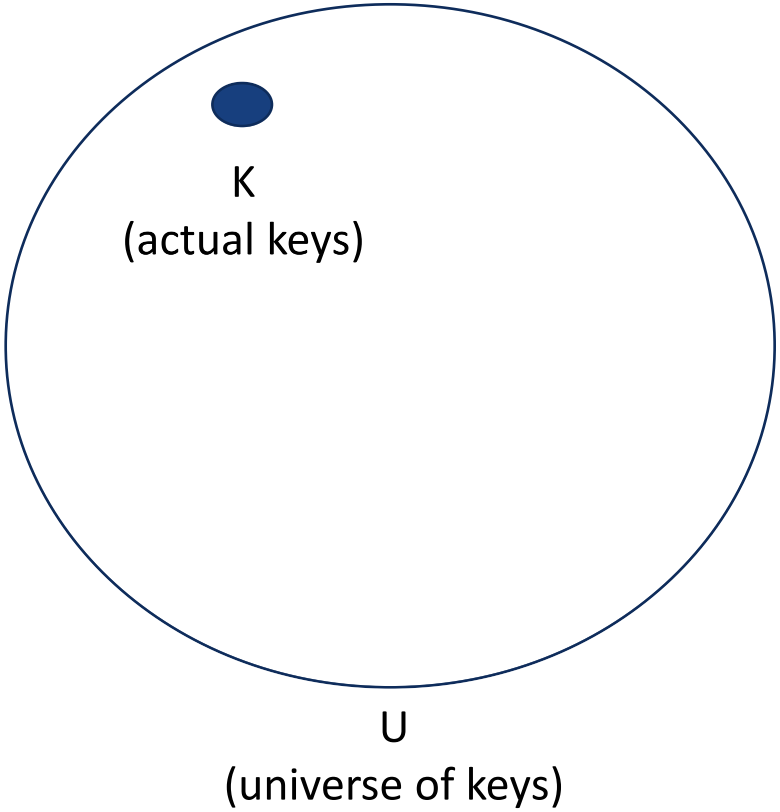
To put **key = 18** to the array successfully, **what change** should we make?

How about **key = 48**?

| | |
|---|---|
| **0** | |
| **1** | |
| **2** | 2 |
| **3** | |
| **4** | 4 |
| **5** | |
| **6** | |
| **7** | 7 |
| **8** | |
| **9** | |
| **10** | 10 |

# Direct Addressing issues

- Let U is the set of universe of keys and K is the set of actual keys
- When K is **much smaller** than U, Direct Addressing is very **inefficient**
  - E.g., an array of 1,000,000 records for ~1000 students



Of course, if we have **a lot** of memory, it's still alright

How to make it **more efficient**?

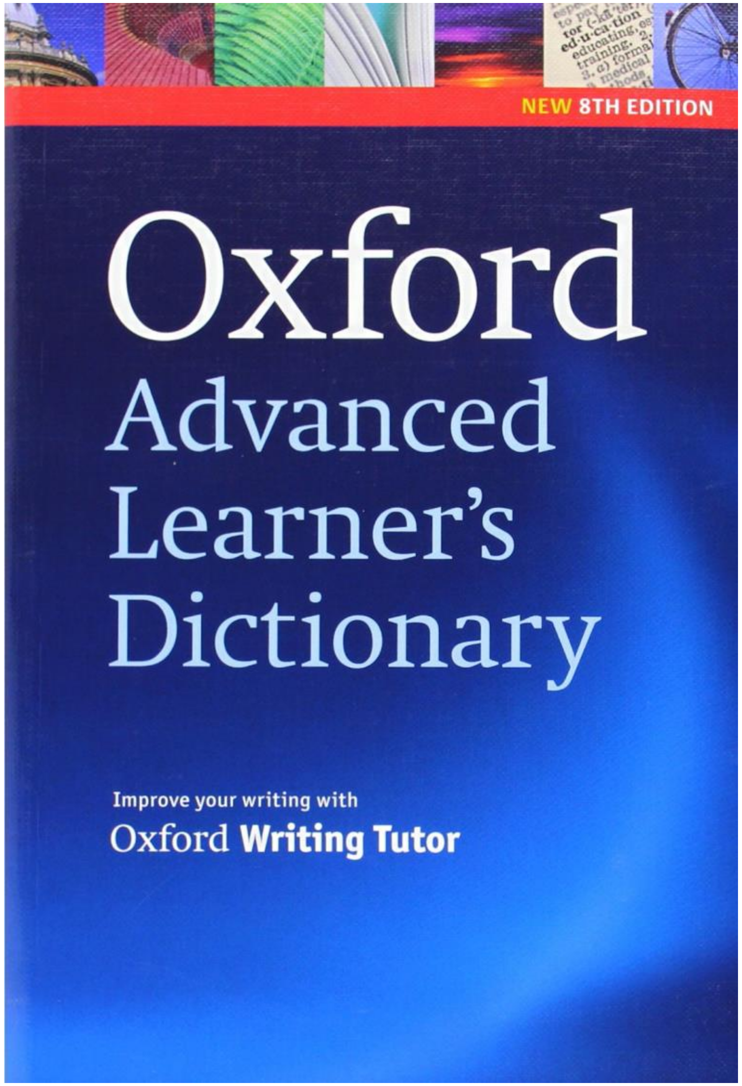
# Discussion

How do you **look up** a **word** in a **physical dictionary**?

A. Linear Search

B. Binary Search

C. A-Z tabs



# Outline

- **Implementing Dictionary ADT**
  - Using Direct Addressing technique
  - **Using Hash Tables**
    - Hash Functions
    - Hash Collisions
    - Resolving Hash Collisions
    - Rehashing
    - Hashing data types other integers
    - Dictionary ADT implementation

# Key Idea

Store **groups of entries**, not **single entries**

| | |
|---|---|
| **0** | |
| **1** | |
| **2** | 2 |
| **3** | |
| **4** | **4, 48** |
| **5** | |
| **6** | |
| **7** | **7, 18** |
| **8** | |
| **9** | |
| **10** | 10 |

# Key Idea

For an entry

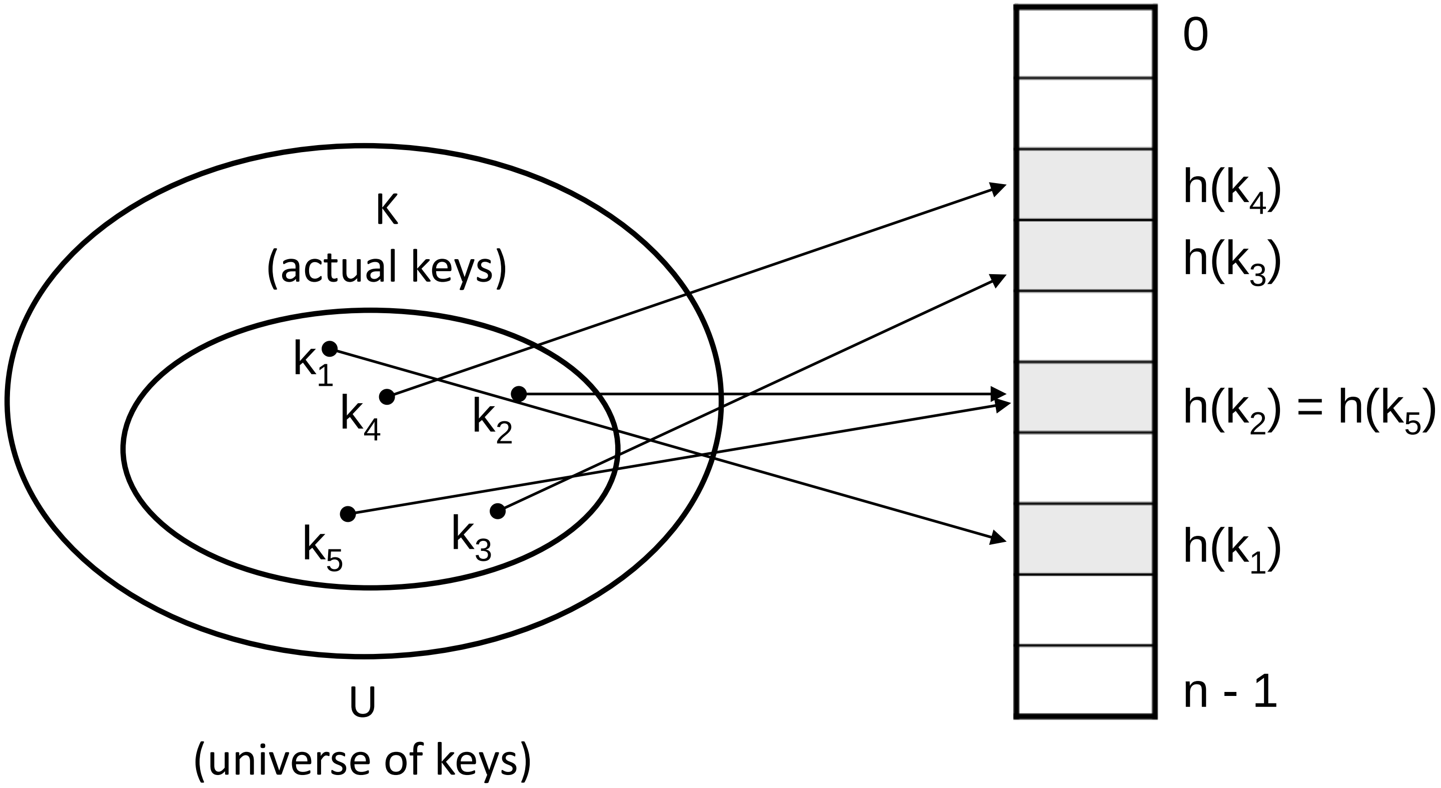
1. **Determine its group** using its key
2. **Store it** into the respective group

| | |
|---|---|
| **0** | |
| **1** | |
| **2** | 2 |
| **3** | |
| **4** | **4, 48** |
| **5** | |
| **6** | |
| **7** | **7, 18** |
| **8** | |
| **9** | |
| **10** | 10 |

Given an entry's key, how to determine its group?

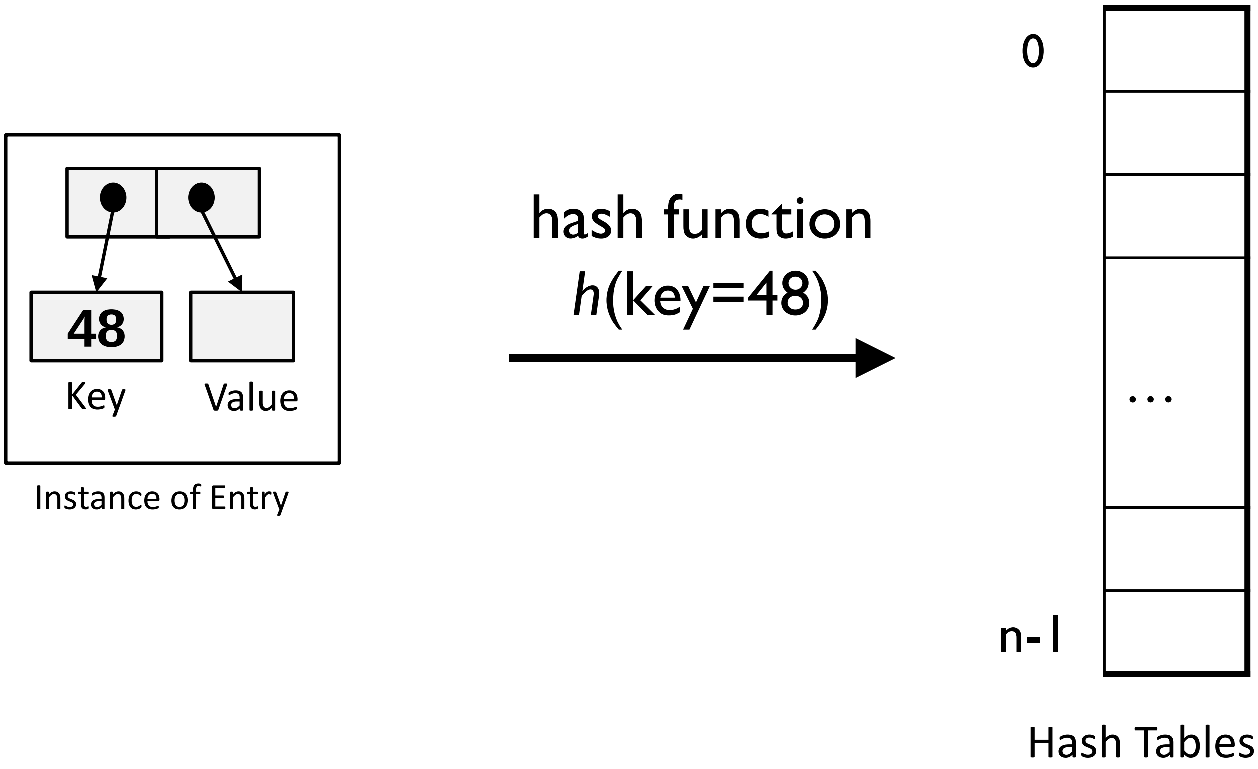
# Hash Tables

**Use** a **function** *h* to **compute the index (group)** for key and store the entry in *h(k)*



# Hash Tables

- Maintain n different “groups/slots/buckets/tabs” (numbered 0 to n-1)
- Any entry (with its respective key) will be put in one of the slots



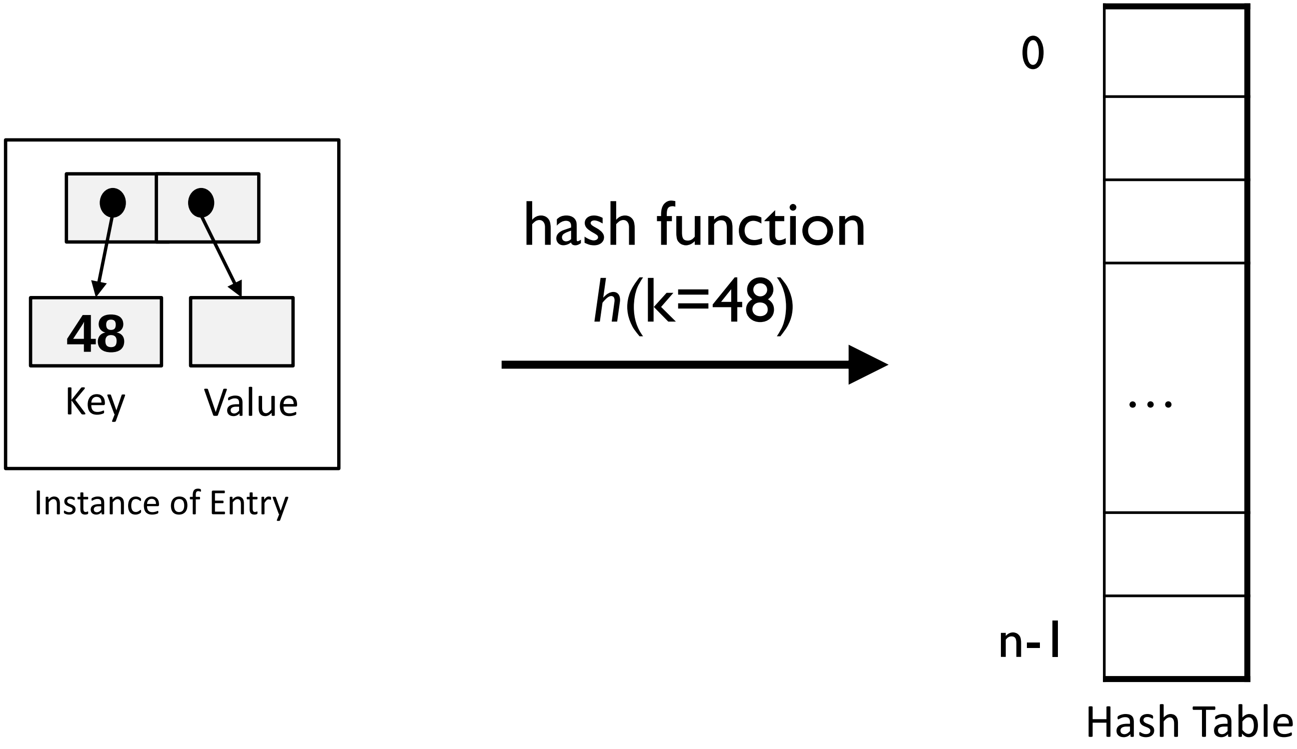
© National University of Singapore. All Rights Reserved.

# Outline

- **Implementing Dictionary ADT**
  - Using Direct Addressing technique
  - **Using Hash Tables**
    - **Hash Functions**
    - Hash Collisions
    - Resolving Hash Collisions
    - Rehashing
    - Hashing data types other integers
    - Dictionary ADT implementation

# Hash Functions

A hash function `h(k)`, given an entry's key, **returns a value from `0` to `n–1`**, determining which slot it belongs in



What should be the function ***`h(k)`*** like? Is ***`h(k) = k + 1`*** good?

# Hash Functions

A good hash function must

1. **Compute fast**
2. **Minimize collisions**
3. **Distribute** entries **uniformly** throughout the hash table

**Given** a key as an **integer**, what **arithmetic operator** will **surely** produce a value 0 to n - 1?

# Hash Functions

Typically, ***h(k) = k % n***

Where **n is a prime number**

- **Result** will then be **between 0 and n – 1**
- When **n** is a **prime number**, entries tends to be distributed **more uniformly**

# Hash Function Example

- Keys are integers
- Table size = 7
- $h(k) = k \% 7$
- **Add**: 25, 7, 51, 33

| | |
|---|---|
| 0 | |
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | |

# Hash Function Example

- Keys are integers
- Table size = 7
- *h(k) = k % 7*
- **Add**: 25, 7, 51, 33

| | |
|---|---|
| **0** | 7 |
| **1** | |
| **2** | 51 |
| **3** | |
| **4** | 25 |
| **5** | 33 |
| **6** | |

# Hash Function Example

- Keys are integers
- Table size = 7
- *h(k) = k % 7*
- **Get**: 51, 9, 1

| | |
|---|---|
| **0** | 7 |
| **1** | |
| **2** | 51 |
| **3** | |
| **4** | 25 |
| **5** | 33 |
| **6** | |

How many operations are needed before a record, e.g. 9, 42, is found?

# Quiz What is the hash table like?

- Keys are integers
- Table size = 11
- *h(k) = k % 11*
- **Add**: 35, 5, 11, 7, 24

# Outline

- **Implementing Dictionary ADT**
  - Using Direct Addressing technique
  - **Using Hash Tables**
    - Hash Functions
    - **Hash Collisions**
    - Resolving Hash Collisions
    - Rehashing
    - Hashing data types other integers
    - Dictionary ADT implementation

# Hash Collisions

**Two keys** can **map** into the **same slot** in the hash table

- *h(k) = k % 11*
- **Add**: 35, 5, 11, 7, **24**

| | |
|---|---|
| **0** | 11 |
| **1** | |
| **2** | 35 |
| **3** | |
| **4** | |
| **5** | 5 |
| **6** | |
| **7** | 7 |
| **8** | |
| **9** | |
| **10** | |

What to do? Should we **replace** 35 by 24 in the slot 2?

***Hint***: do the two entries have the same key?

# Outline

- **Implementing Dictionary ADT**
  - Using Direct Addressing technique
  - **Using Hash Tables**
    - Hash Functions
    - Hash Collisions
    - **Resolving Hash Collisions**
      - **Open Addressing with Linear Probing (self study)**
      - Open Addressing with Quadratic Probing (self exploration)
      - Open Addressing with Double Hashing (self exploration)
      - Separate Chaining
    - Rehashing
    - Hashing data types other integers
    - Dictionary ADT implementation

# Linear Probing

Self study

Resolve collisions in slot *i* by **putting the entry** into **next available slot** (*i+1, i+2, ...*)

- ➢ *h(k) = k % 11*
- ➢ **Add**: 35, 5, 11, 7, **24, 14, 25**

| | |
|---|---|
| **0** | 11 |
| **1** | |
| **2** | 35 |
| **3** | **24** |
| **4** | **14** |
| **5** | 5 |
| **6** | **25** |
| **7** | 7 |
| **8** | |
| **9** | |
| **10** | |

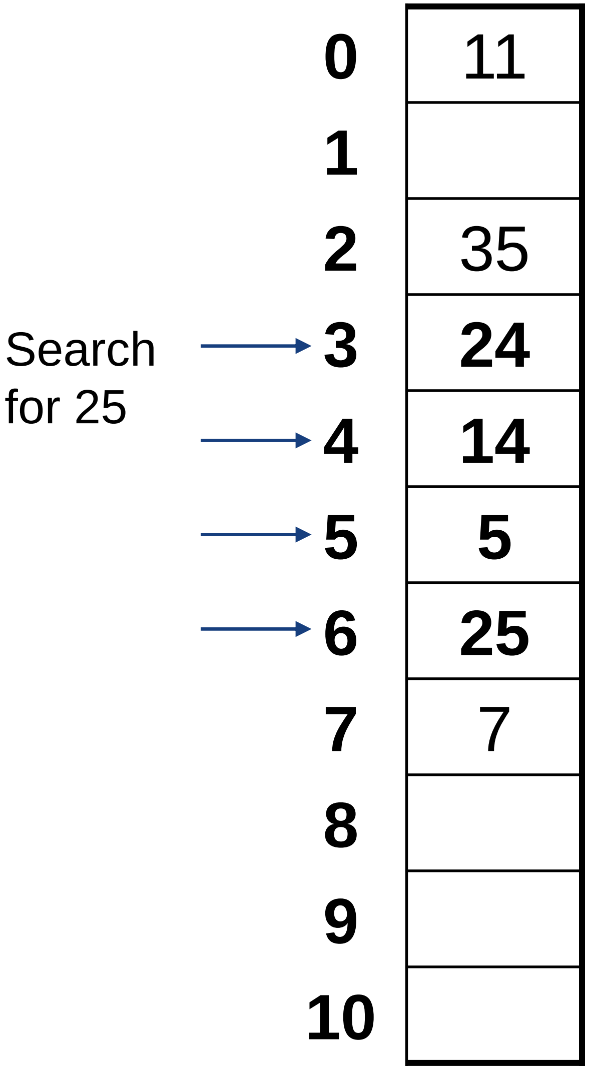
How can we **search** for ***key = 25***?

How many **operations** are there before the record is found?

# Primary Clustering Issue

Self study

Because so **many nodes** may be **grouped together** of consecutive locations, **performance** would be **affected**



Search for 25

| | |
|---|---|
| **0** | 11 |
| **1** | |
| **2** | 35 |
| **3** | **24** |
| **4** | **14** |
| **5** | **5** |
| **6** | **25** |
| **7** | 7 |
| **8** | |
| **9** | |
| **10** | |

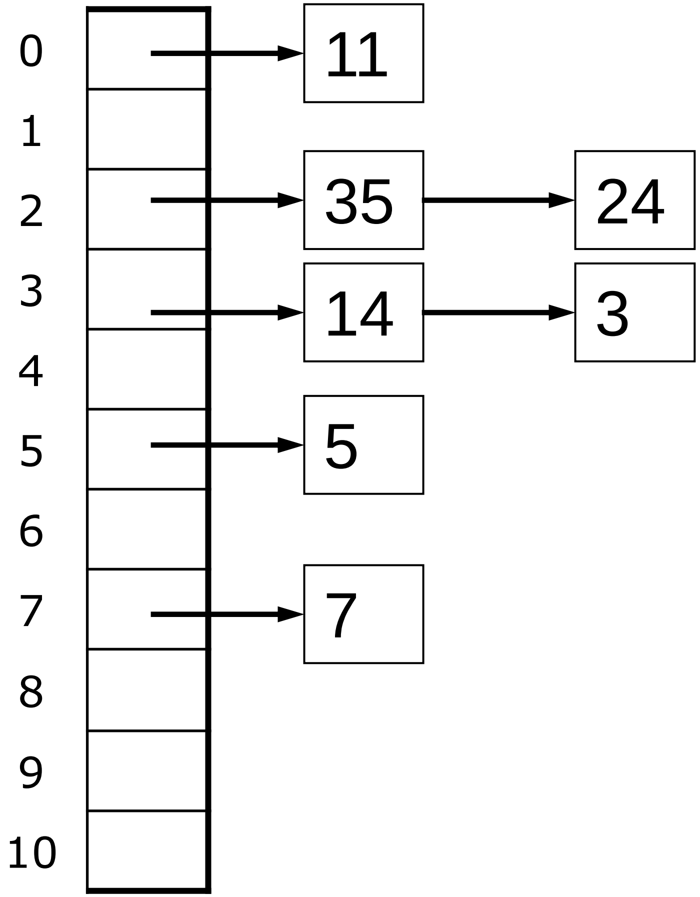
# Outline

- **Implementing Dictionary ADT**
  - Using Direct Addressing technique
  - **Using Hash Tables**
    - Hash Functions
    - Hash Collisions
    - **Resolving Hash Collisions**
      - Open Addressing with Linear Probing
      - Open Addressing with Quadratic Probing (explore-by-yourself)
      - Open Addressing with Double Hashing (explore-by-yourself)
      - **Separate Chaining**
    - Rehashing
    - Hashing data types other integers
    - Dictionary ADT implementation

# Separate Chaining

Instead of storing keys, **each slot** in the hash table **stores** a **linked list**

- *h(k) = k % 11*
- **Add**: 35, 5, 11, 7, **24,** 14, **3**



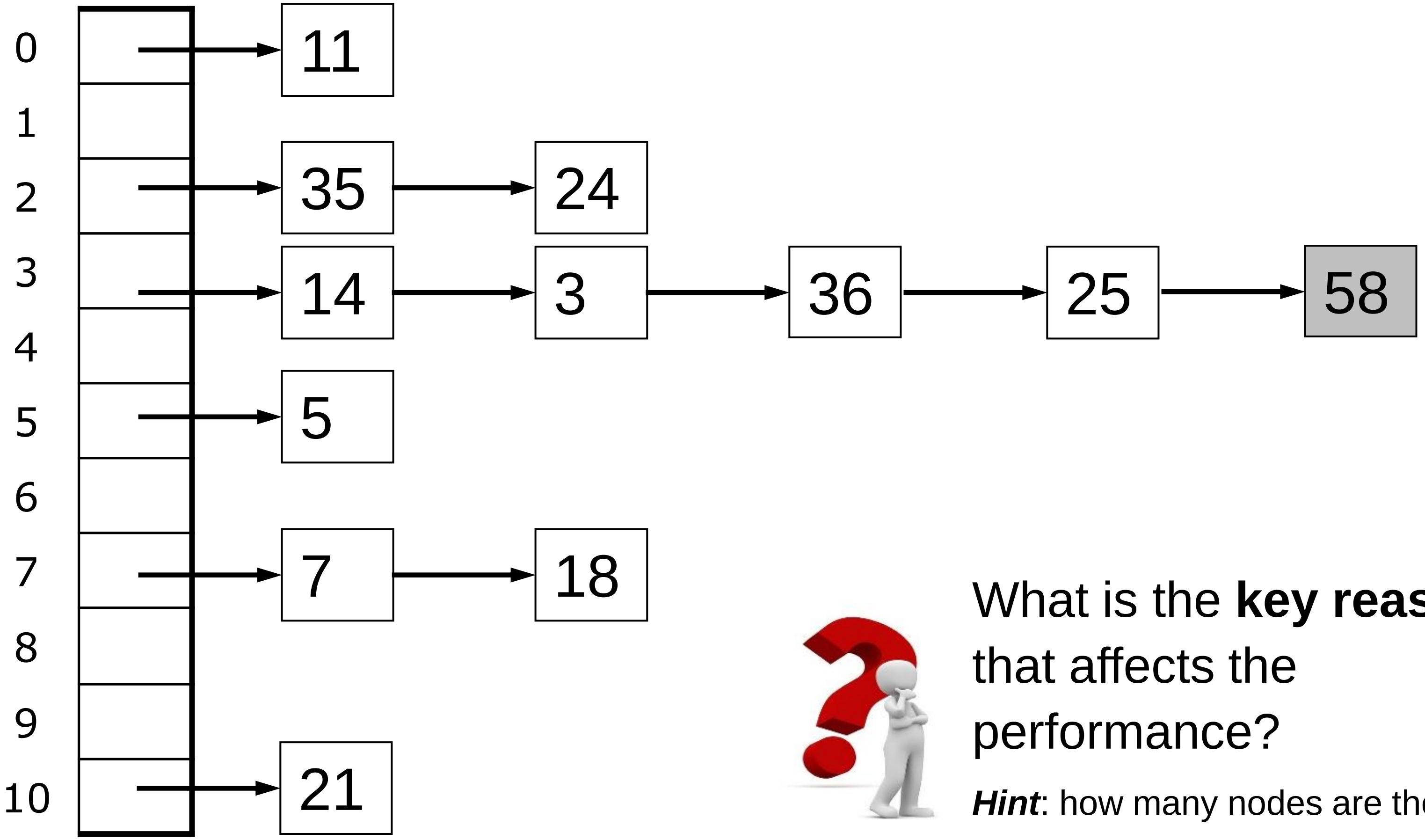
How can we **search** for ***key = 3***?

How many **operations** are there before the record is found?

# Question

Self study

Look at the following scenario. How many operations are there before the entry with ***key 58*** is found?



What is the **key reason** that affects the performance?

***Hint***: how many nodes are there?

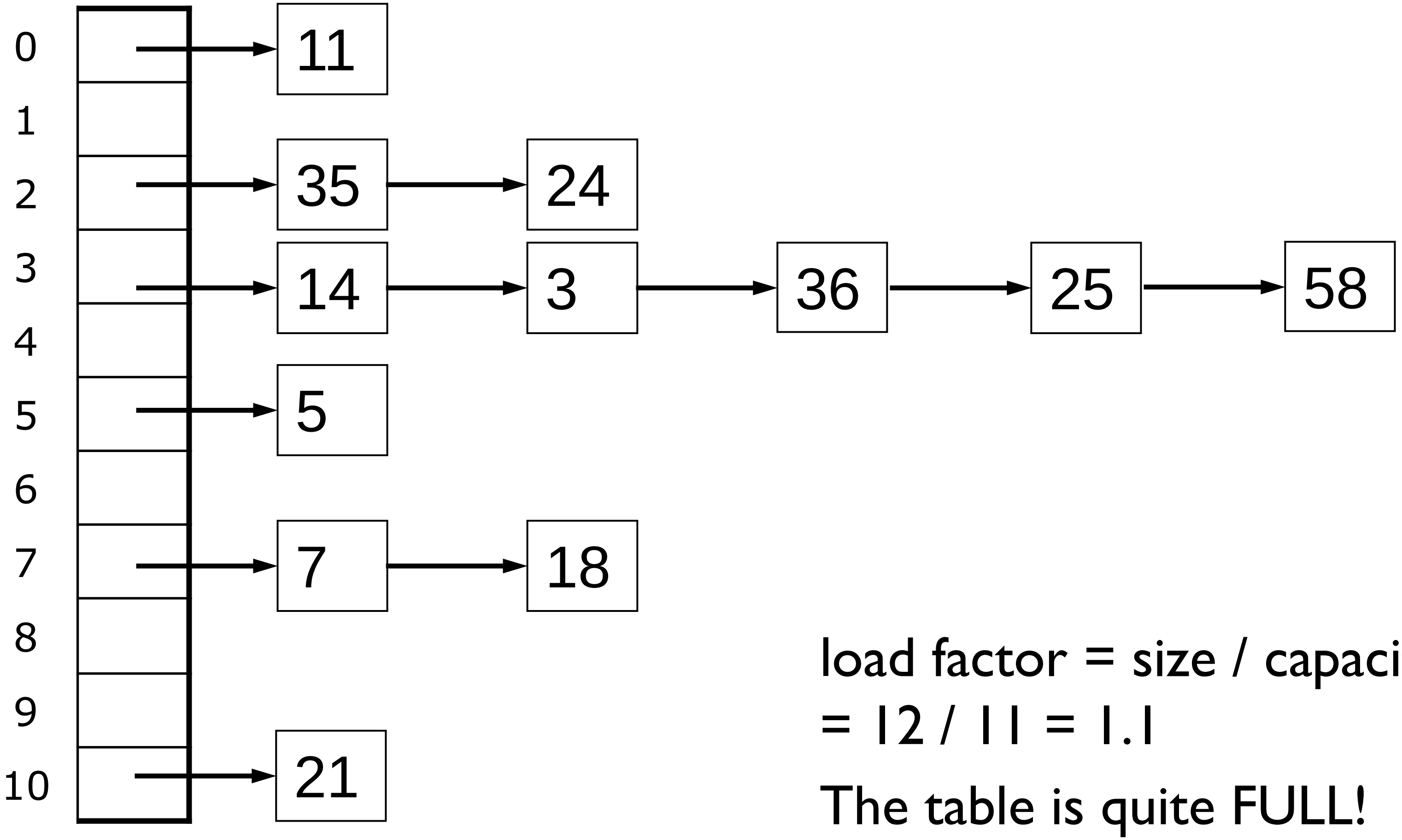
# Outline

- **Implementing Dictionary ADT**
  - Using Direct Addressing technique
  - **Using Hash Tables**
    - Hash Functions
    - Hash Collisions
    - Resolving Hash Collisions
    - **Rehashing (self study)**
    - Hashing data types other integers
    - Dictionary ADT implementation

# Load Factor

Self study

The load factor **λ** is the **ratio** of the **size of the dictionary** to the **capacity of a hash table**, indicating **how full** the hash table is



load factor = size / capacity
= 12 / 11 = 1.1
The table is quite FULL!

# Analysis of Hash Table Search

Self study

- Unsuccessful: $\lambda$
  - The **average length** of a list at h(*k*)
- Successful: 1 + ($\lambda$/2)
  - One node, plus half the average length of a list (not including the item)
- **Reasonable efficiency requires only $\lambda$ < 1**

| λ | Unsuccess | Success |
|---|---|---|
| 0.1 | 0.1 | 1.1 |
| 0.5 | 0.5 | 1.3 |
| 0.9 | 0.9 | 1.5 |
| 1.3 | 1.3 | 1.7 |
| 1.7 | 1.7 | 1.9 |
| 2.0 | 2.0 | 2.0 |

What should we do when $\lambda$ becomes **too large**?

# Rehashing

Self study

When **load factor** becomes too **large**, we should **expand** the hash table

1. **Double the current size** and increase the result to the **next prime number**
   - E.g., if current size is 5, new size will be 11
2. **Place** the current **entries into new hash table**, using method *`Add()`*

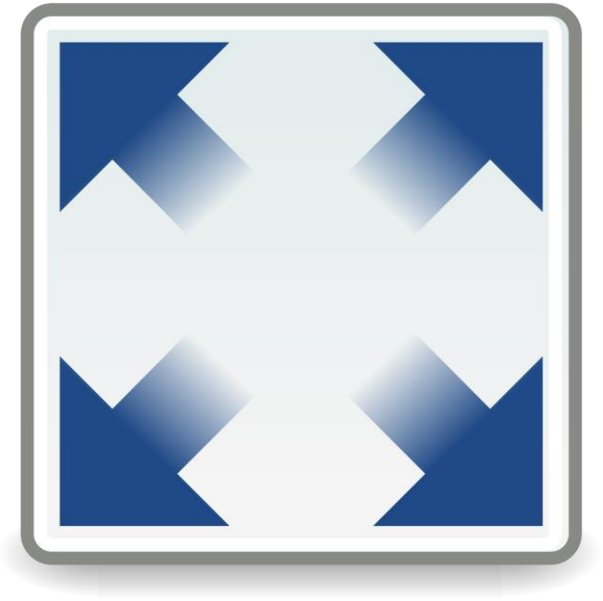
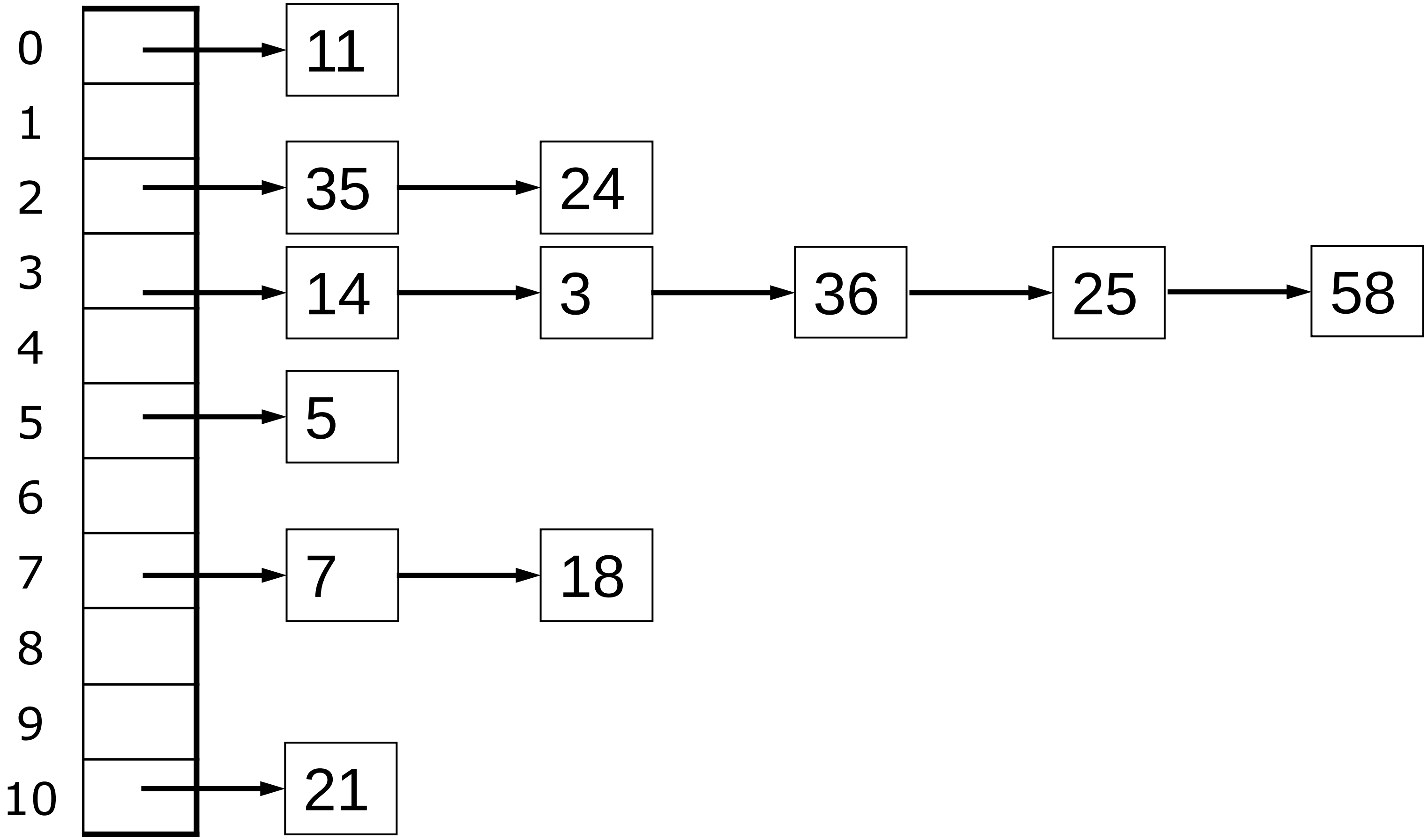


Image by OpenIcons from Pixabay

# Quiz

Self study

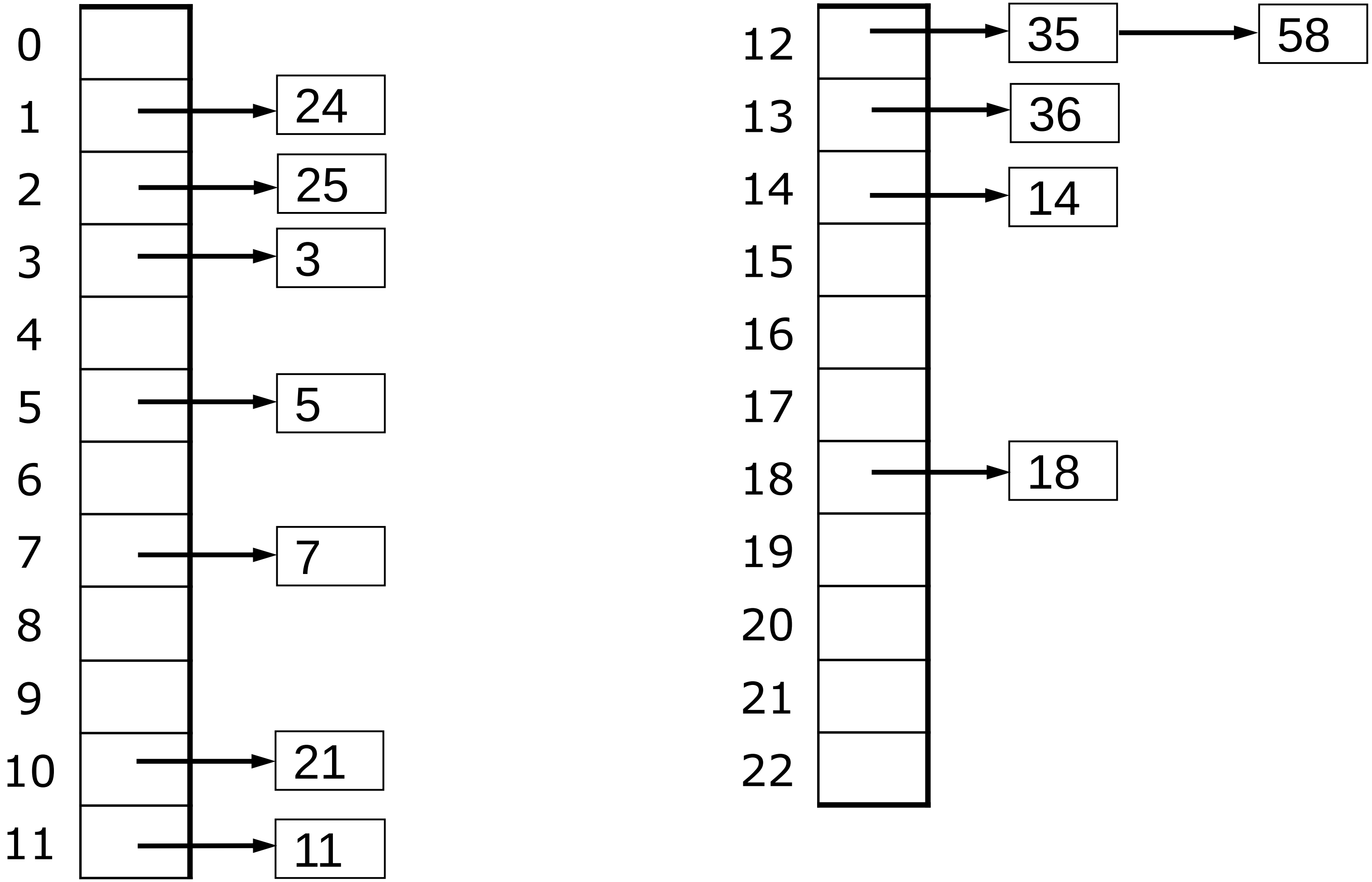
Rehash the following hash table



# Quiz Solution

Self study

Current size is 11, double size is 22 and next prime number is 23



# Next

Self study

So far, we have discussed hash functions when search keys are integers. How about when **keys** are **strings** or some **general objects**?

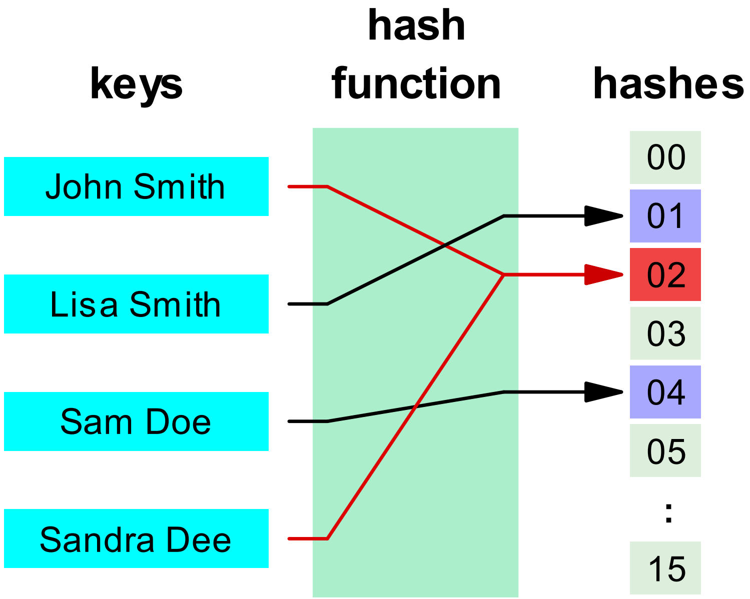


Image by Jorge Stolfi, Wikipedia

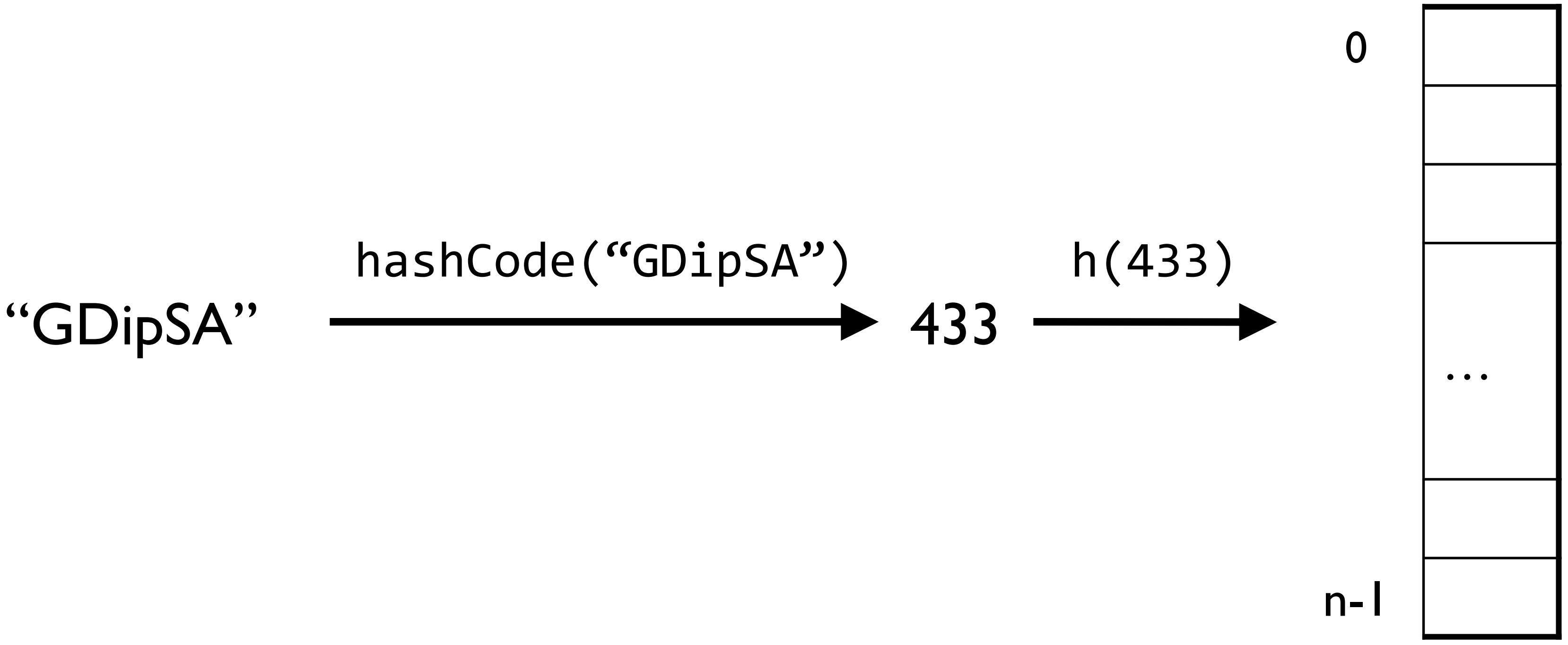
# Outline

- **Implementing Dictionary ADT**
  - Using Direct Addressing technique
  - **Using Hash Tables**
    - Hash Functions
    - Hash Collisions
    - Resolving Hash Collisions
    - Rehashing
    - **Hashing data types other integers (self study)**
    - Dictionary ADT implementation

# Hashing a String

Self study

We need **another step** to **convert** the **string into** an **integer.** The **result** is called **hash code**



Note: 433 is merely an example for illustrative purpose

# Hash Codes for Strings

Self study

For example, a simple way **to calculate Hash Code**:

1. **Assign integer** to **each character** in string
   - Use 1 – 26 for ‘a’ to ‘z’, or
   - Use Unicode integer
2. **Sum** the **integers** of the characters for the hash code

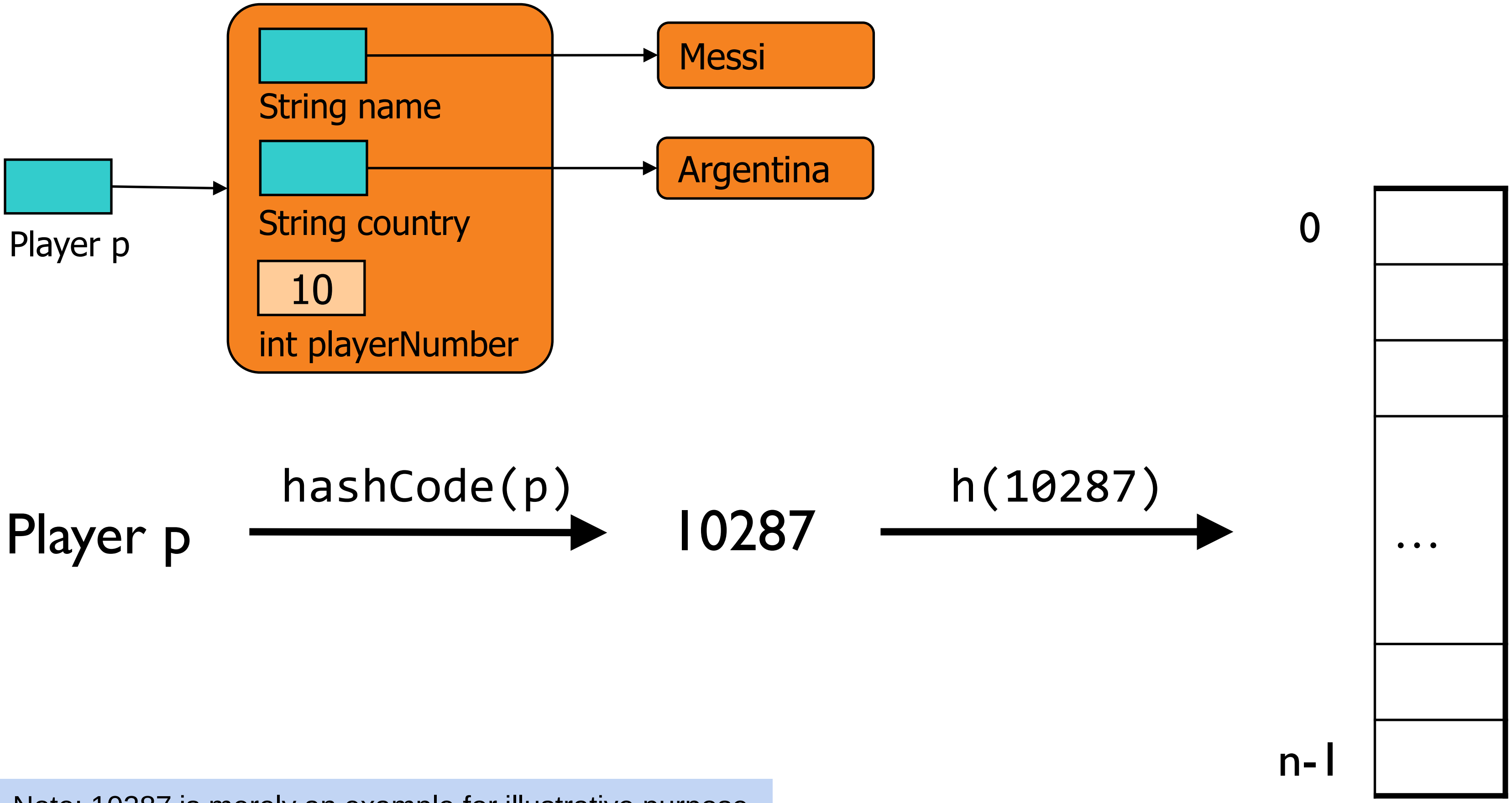
| Char | Unicode |
|---|---|
| D | 68 |
| i | 105 |
| p | 112 |
| S | 83 |
| A | 65 |
| Hash code | 433 |

Implementation is **reality** may be **more complicated**

# Hashing a General Object

Self study

Just like strings, we need another step to **convert** the **object into** an **integer**



Note: 10287 is merely an example for illustrative purpose

# Hash codes for objects

Self study

Following is a C# implementation **example** to **compute** the **hash codes** for ***Player*** **objects**

```
class Player {
   public string Name { set; get; }
   public string Country { set; get; }
   public int PlayerNumber { set; get; }

   public Player(string name, string country, int number) {
      Name = name;
      Country = country;
      PlayerNumber = number;
   }

   public override int GetHashCode() {
      return HashCode.Combine(
         Name, Country, PlayerNumber);
   }
}
```

# Hash codes for an objects

Self study

Let’s test with two *Player* objects

```
public static void Main()
{
   Player p1 = new Player("Ronaldo", "Portugal", 7);
   Console.WriteLine("{0}, {1}, {2}, {3}",
           p1.Name.GetHashCode(), p1.Country.GetHashCode(),
           p1.PlayerNumber.GetHashCode(), p1.GetHashCode());

   Player p2 = new Player("Messi", "Argentina", 10);
   Console.WriteLine("{0}, {1}, {2}, {3}",
           p2.Name.GetHashCode(), p2.Country.GetHashCode(),
           p2.PlayerNumber.GetHashCode(), p2.GetHashCode());
}
```

```
1029606906, -1174981222, 7, -189148130
1098167073, -520689015, 10, 633900568
```

C# implementation is a bit more complicated than our Hash Code example

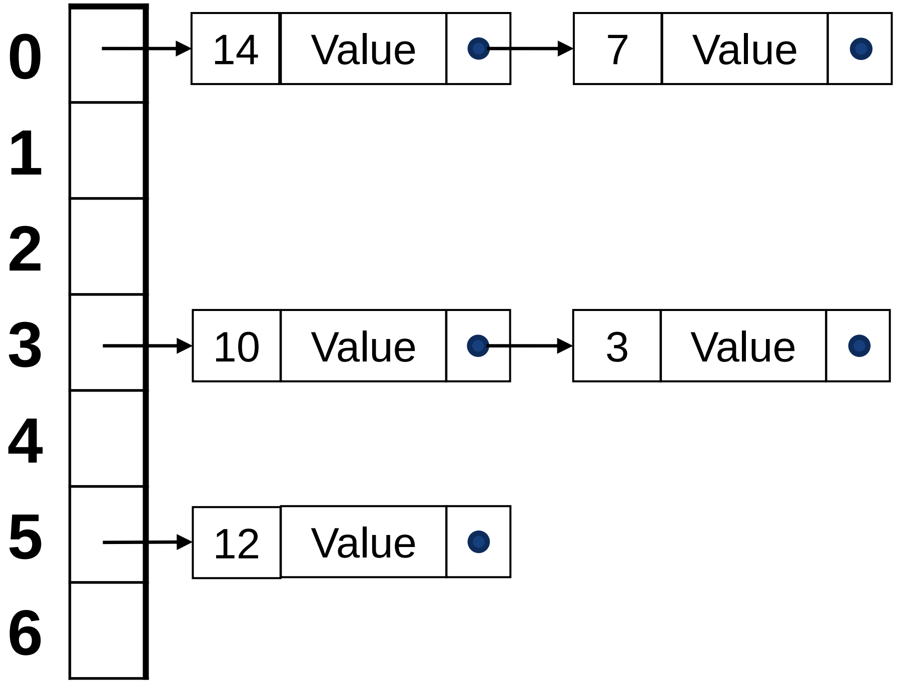
# Outline

- **Implementing Dictionary ADT**
  - Using Direct Addressing technique
  - **Using Hash Tables**
    - Hash Functions
    - Hash Collisions
    - Resolving Hash Collisions
    - Rehashing
    - Hashing data types other integers
    - **Dictionary ADT implementation**

# Implement Dictionary ADT

**Key ideas:**

1. **Keeps** an array of entries as the **hash table**
2. **Each** array **entry** references a **Linked List (group)**
3. **Rehash** when load factor becomes large

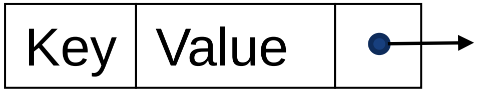


# Implement Dictionaries

**Each entries** contains: **Key**, **Value** and **Link to the next** entry in the same bucket

```
class Entry
{
   public int Key { set; get; }
   public string Value { set; get; }
   public Entry Next { set; get; }

   public Entry (int key, string value)
   {
      Key = key;
      Value = value;
      Next = null;
   }
}
```

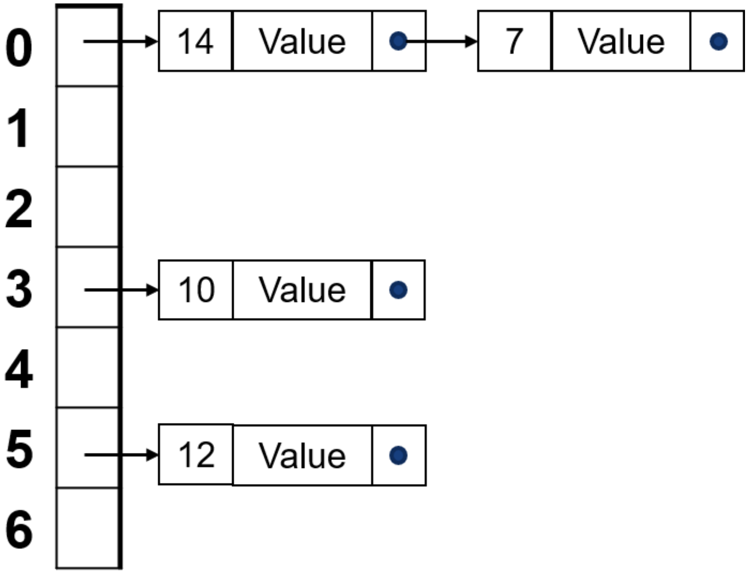


# Implementing Operation Add

*Algorithm for Add(key, value)*

*// Adds a new key-value entry to the dictionary. If key already exists, throws an ArgumentException*

```
Apply hash function to key and find the entry in the respective bucket
if (an entry containing key is found)
    Throw an ArgumentException
else
    Insert the key, value pair into the hash table as a new entry
    After adding the new element, if load factor is too large, rehash
```



How can we **add** entry with ***key=2***?

How about ***key=21***?

How about ***key=7***?

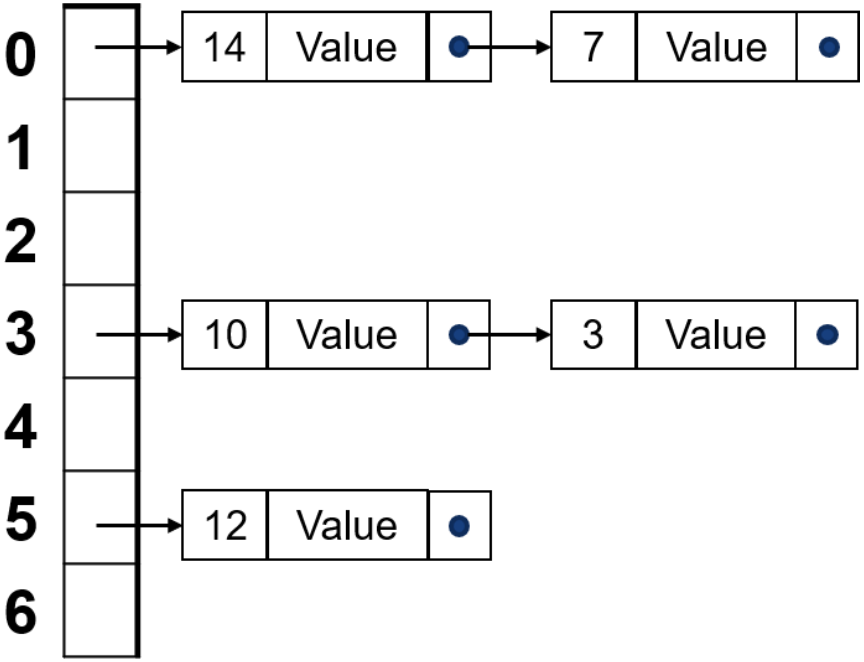
# Implementing Operation Get

*Algorithm for Get(key)*

*// Retrieve the respective value for the given key. If key does not exist, return null*

**Apply hash function to key** and find the entry in the respective bucket

```
If an entry is found, return the value
Otherwise, return null
```



How can we **search** for ***key=3***?

How about ***key=17***?

# Quiz

Self study

Draw the **final dictionary data** if we insert the following data to a Dictionary, which is implemented using Hash Table with Linked List, and:

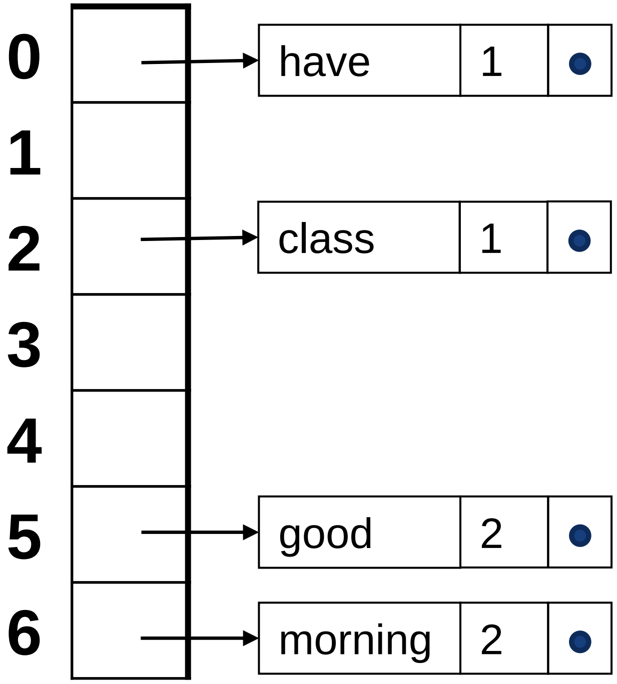
- **Original table size: 7**
- **Load factor threshold: 0.55**

| Key (string) | Key Hash Code | Code %7 | Code %17 | Value (int) |
|---|---|---|---|---|
| good | 425 | 5 | 0 | 2 |
| morning | 762 | 6 | 14 | 2 |
| class | 534 | 2 | 7 | 1 |
| have | 420 | 0 | 12 | 1 |
| a | 97 | 6 | 12 | 1 |
| great | 531 | 6 | 4 | 1 |
| day | 318 | 3 | 12 | 1 |

# Quiz Solution

Self study

After the first 4 records, the array is as follows



Array of entries
(Hash table)

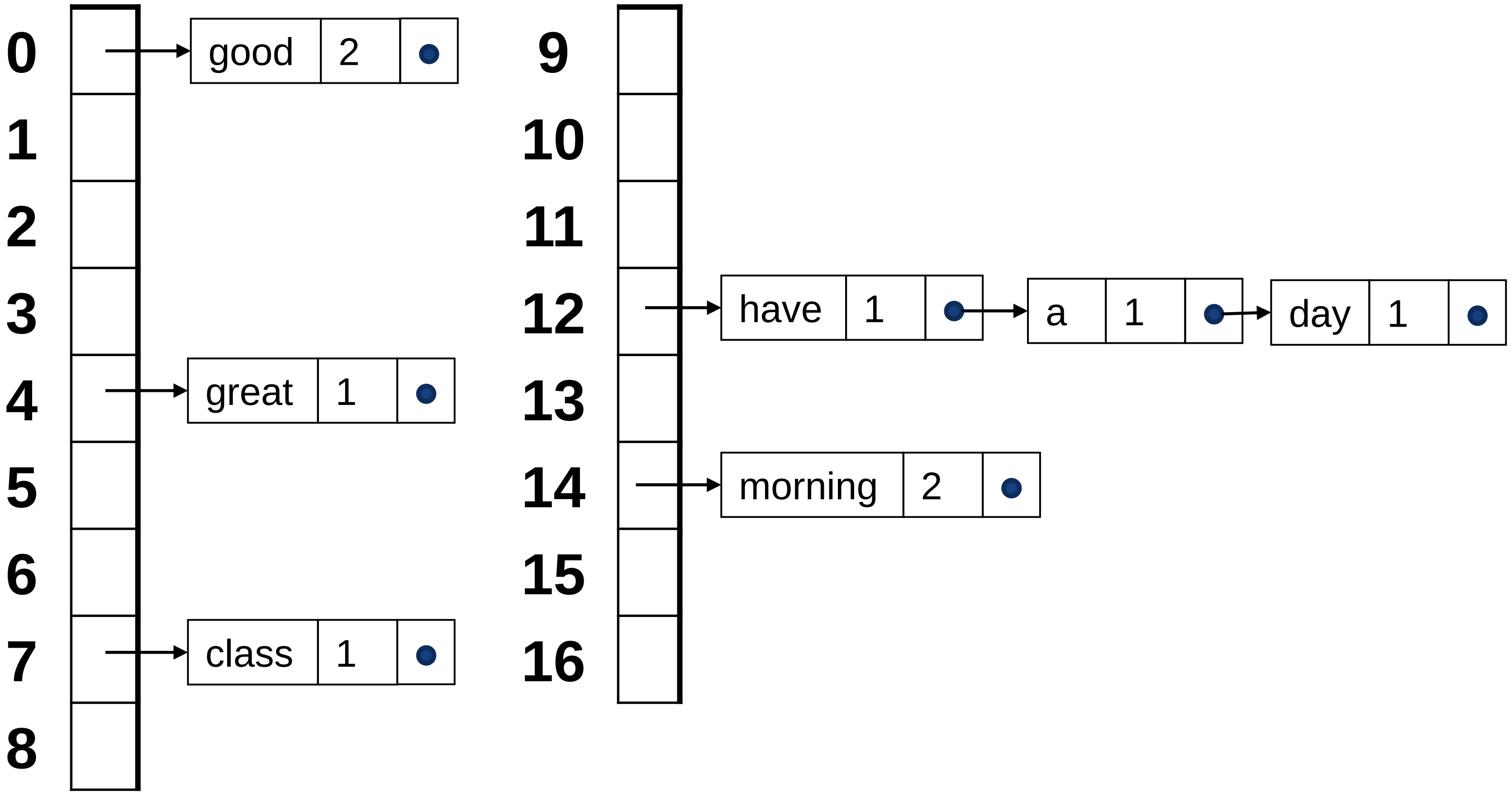
Load factor λ = size / capacity = 4 / 7 = 0.57

Need Rehashing!

# Quiz Solution

Self study

The new array size is 17. And after 7 records:



Array of entries
(Hash table)

# Runtime Efficiency

`Add(key, value)`

- Most of the time: **O(1)**
- Rehashing take **O(n)**
  - **How often** do we need to do it?
  - If **knowing** our **data size**, we can set the hash table’s array size **in advance**

`Get(key)`

- In theory, worst case **O(n)**
  - Does it even happen ☺?
- Most of the time:
  - Unsuccessful search: λ
  - Successful search: 1 + (λ/2)
  - Both are **O(1)**

# Readings

- Data structures and abstractions with Java, 4ed – Chapter 21, Introducing Hashing, *Frank M.Carrano and Timothy M. Henry*
- Data structures and abstractions with Java, 4ed – Chapter 22, Hashing as a Dictionary implementation, *Frank M.Carrano and Timothy M. Henry*