8 Hash Tables

For COMSC 132

Sam Bowne

Oct 12, 2024

Arrays and lists

- Arrays and lists store data elements in sequence
- They are addressed by index number

Hash tables

- Elements are accessed by a keyword rather than an index number
- Data items are stored in key-value pairs similar to dictionaries
- Dictionaries are often built using hash tables
- Hash tables store data in a very efficient way
  - Data retrieval can be very fast

Dictionary

```python
my_dict={"Basant" : "9829012345", "Ram": "9829012346", "Shyam": "9829012347", "Sita":
"9829012348"}
print("All keys and values")
for x,y in my_dict.items():
    print(x, ":" , y)        #prints keys and values
my_dict["Ram"]
```

```
Basant : 9829012345
Ram : 9829012346
Shyam : 9829012347
Sita : 9829012348
'9829012346'
```

- Data stored in key:value pairs

Hash Table

- This hash function is
  - sum(ord) % 256

keyword	hash	Value
Basant	89	9829012345
Ram	32	9829012346
Shyam	2	9829012347
Sita	145	9829012348

```
d = {"Basant" : "9829012345",
"Ram": "9829012346", "Shyam":
"9829012347", "Sita":
"9829012348"}
for name in d:
  key = 0
  for c in name:
    key += ord(c)
  key = key % 256
  print(name, key)
```

Hashing functions

- Input is data of any size
- Output is a small integer value
- Consider this hash function
  - sum(map(ord, 'hello world'))
- It adds the ASCII values of the characters

h	e	l	l	o		w	o	r	l	d	
104	101	108	108	111	32	119	111	114	108	100	= 1116

Hash collision

- These two strings have the same hash value
  - hello, world
  - gello, xorld

h	e	l	l	o		w	o	r	l	d	
104	101	108	108	111	32	119	111	114	108	100	= 1116

g	e	l	l	o		x	o	r	l	d	
103	101	108	108	111	32	120	111	114	108	100	= 1116
-1						+1					

Perfect hashing function

- Produces a unique hash value for any input
- BUT perfection makes the hash function slow
- So we use a fast one and develop a strategy to handle the collisions

Add a multiplier

h	e	l	l	o		w	o	r	l	d	
104	101	108	108	111	32	119	111	114	108	100	= 1116
1	2	3	4	5	6	7	8	9	10	11	
104	202	324	432	555	192	833	888	1026	1080	1100	= 6736

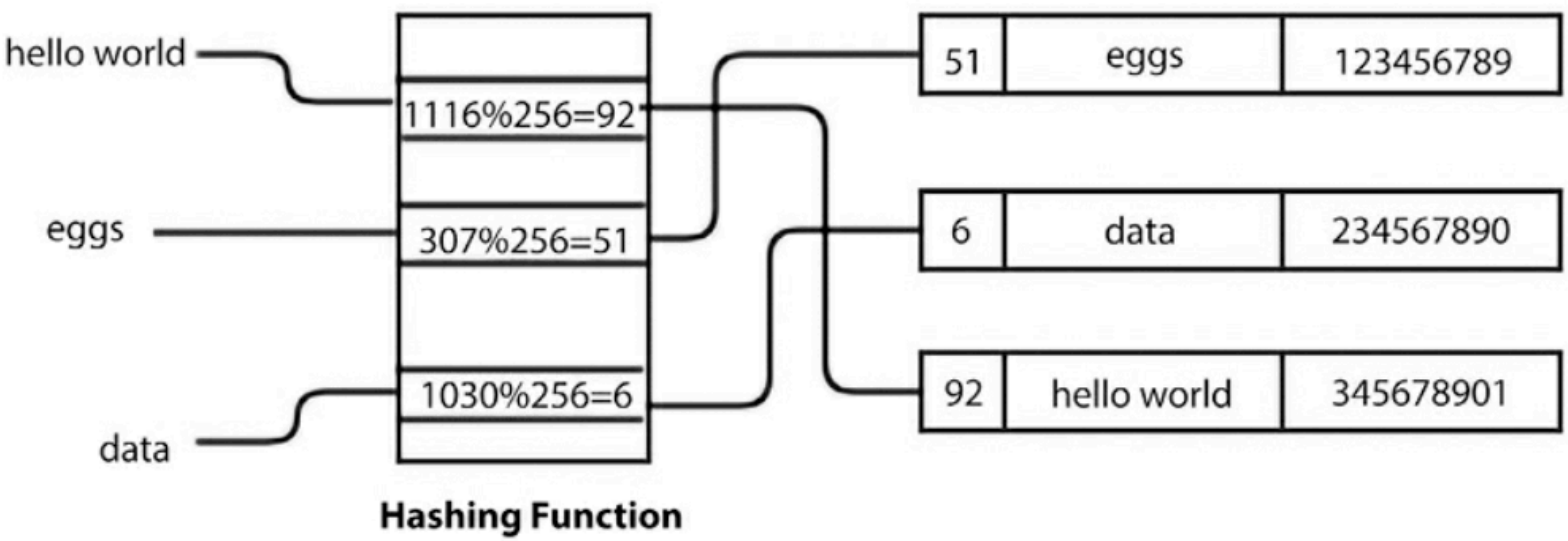
- Some collisions are prevented
- Some remain

```
hello world: 6736
world hello: 6616
gello xorld: 6742
```

```
ad: 297
ga: 297
```

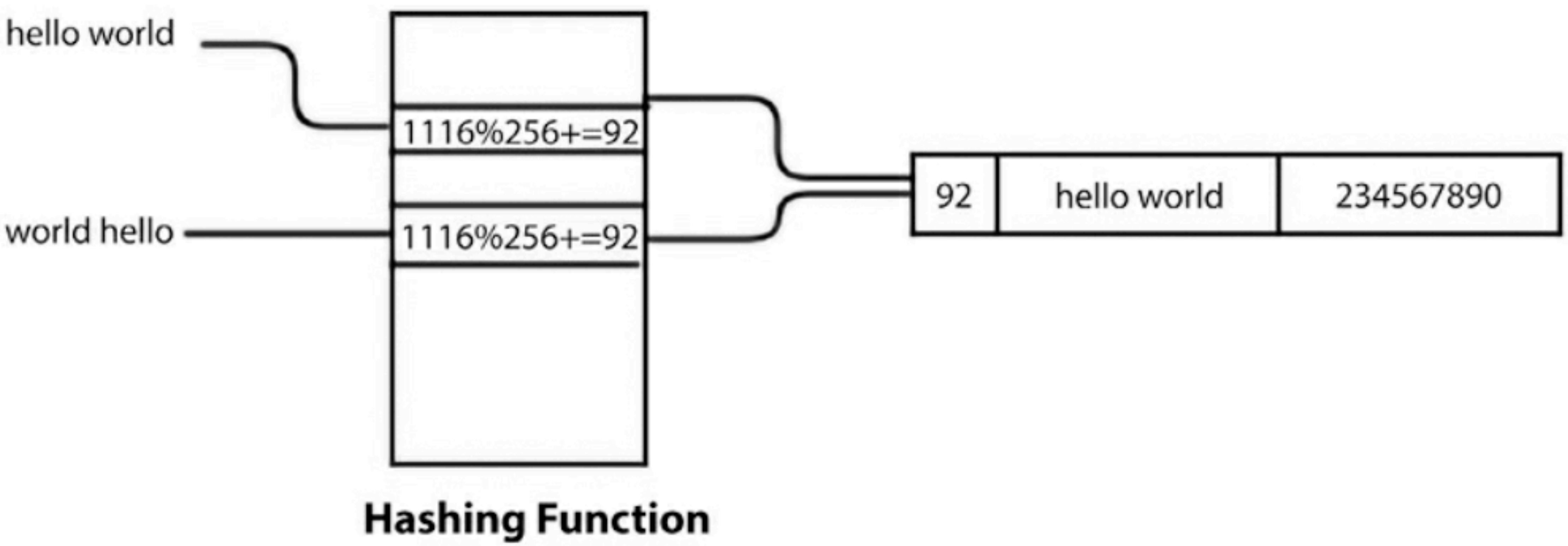
Resolving collisions

- Sample hash function
- Sum ASCII values, take mod 256



Resolving collisions

- **hello world** and **world hello** collide

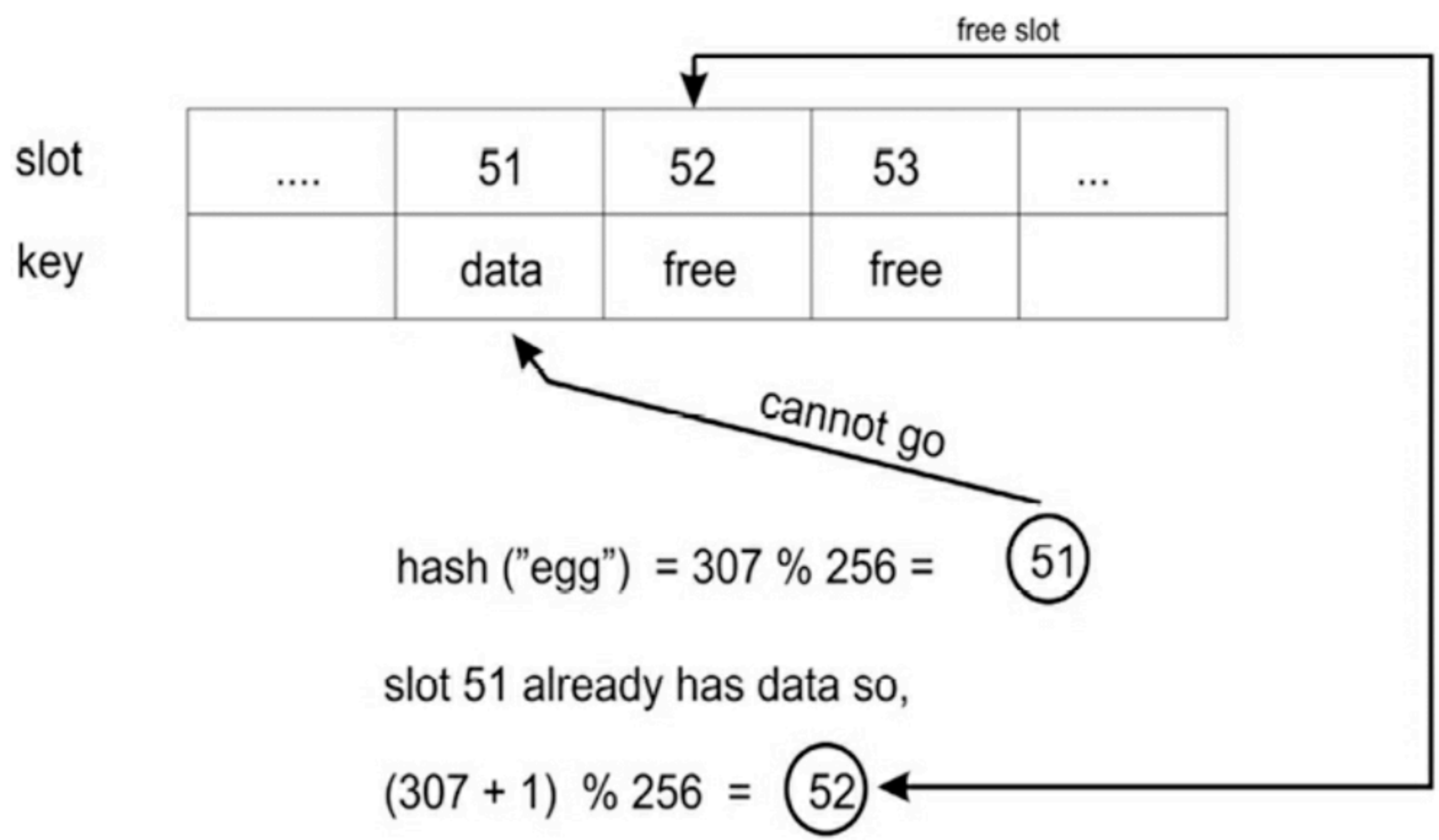


Open addressing

- Collisions are resolved by searching (**probing**) for an alternate position to store the data
- Three popular methods
  - **Linear probing**
  - **Quadratic probing**
  - **Double hashing**

Linear probing

- Add 1 to the hash value that collided
- repeat until a free hash value is found



Linear probing

- The hash table may have clusters of items
  - consecutive occupied positions
- Parts of the table may become dense
  - While other parts are empty
- Making the hash table inefficient

Implementing hash tables

- Stores data in a list of **size** 256
- **count** is the number of items actually stored

```
class HashItem:
    def __init__(self, key, value):
        self.key = key
        self.value = value
```

```
class HashTable:
    def __init__(self):
        self.size = 256
        self.slots = [None for i in range(self.size)]
        self.count = 0
```

Hash function

- Start with underscore
- To indicate a function intended for internal use

```
def _hash(self, key):
        mult = 1
        hv = 0
        for ch in key:
            hv += mult * ord(ch)
            mult += 1
        return hv % self.size
```

Storing elements in a hash table

- Implements linear probing
- **check_growth** method expands the size of the hash table if it's nearly full

```
def put(self, key, value):
    item = HashItem(key, value)
    h = self._hash(key)
    while self.slots[h] != None:
        if self.slots[h].key == key:
            break
        h = (h + 1) % self.size
    if self.slots[h] == None:
        self.count += 1
    self.slots[h] = item
    self.check_growth()
```

Growing a hash table

- Load factor is (used slots) / (total slots)
- Here the MAXLOADFACTOR is 0.65

```
class HashTable:
    def __init__(self):
        self.size = 256
        self.slots = [None for i in range(self.size)]
        self.count = 0
        self.MAXLOADFACTOR = 0.65
```

Growing a hash table

- Checks load factor
- calls **self.growth** if necessary

```
def check_growth(self):
    loadfactor = self.count / self.size
    if loadfactor > self.MAXLOADFACTOR:
       print("Load factor before growing the hash table", self.count / self.size )
        self.growth()
        print("Load factor after growing the hash table", self.count / self.size )
```

Growing a hash table

- Doubles table size
- Inserts all the old values into the new table

```python
def growth(self):
    New_Hash_Table = HashTable()
    New_Hash_Table.size = 2 * self.size
    New_Hash_Table.slots = [None for i in range(New_Hash_Table.size)]

    for i in range(self.size):
        if self.slots[i] != None:
            New_Hash_Table.put(self.slots[i].key, self.slots[i].value)

    self.size = New_Hash_Table.size
    self.slots = New_Hash_Table.slots
```

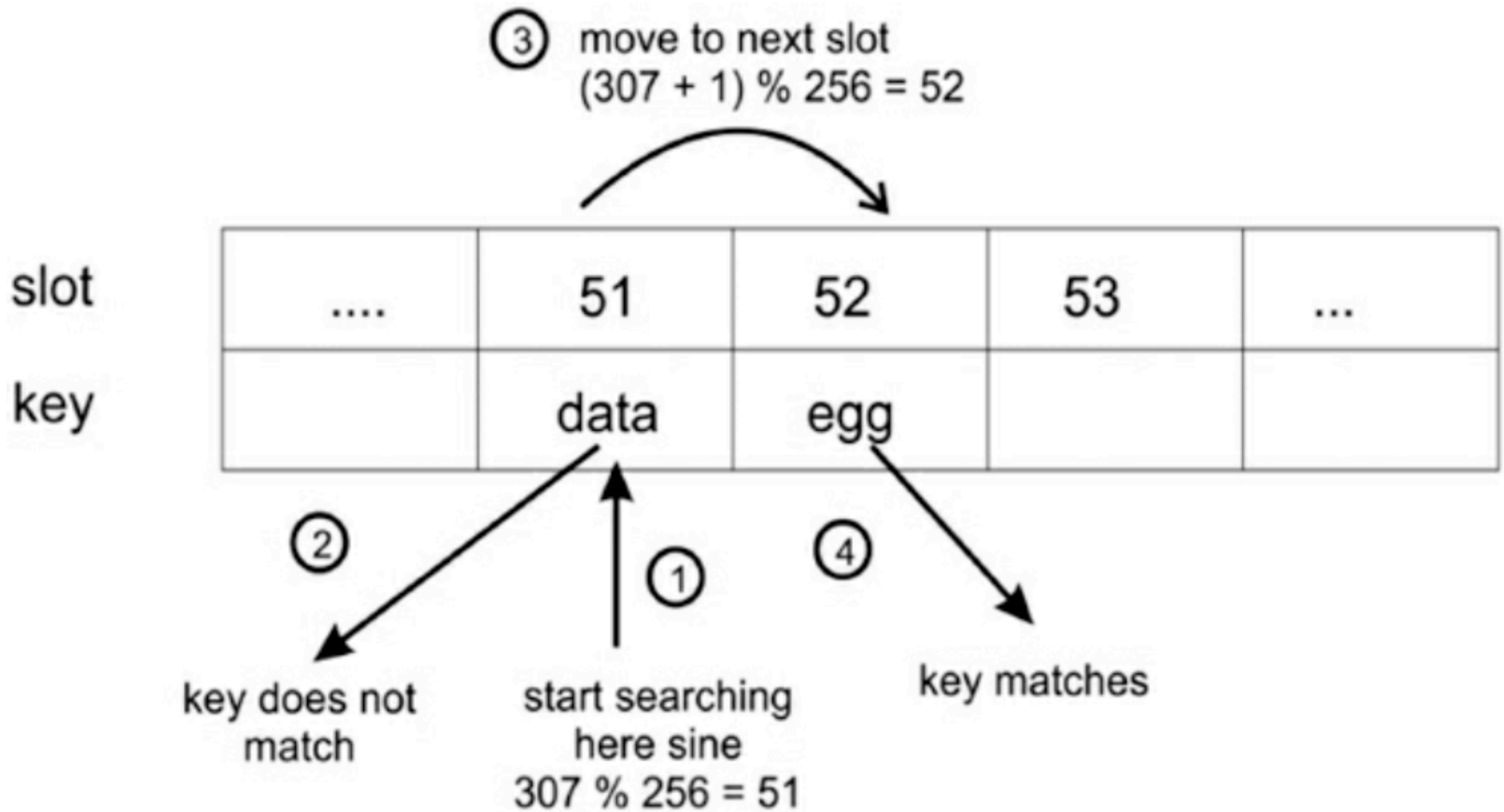
Growing a hash function

- Load factor is (used slots) / (total slots)
- Here the MAXLOADFACTOR is 0.65

Retrieving elements from the hash table

- Compute hash of key
- Look up data at that hash value
- If key item in table matches desired key, we're done
- Otherwise, add 1 repeatedly until desired key is found

Retrieving elements from the hash table



Retrieving elements from the hash table

```
def get(self, key):
    h = self._hash(key)    # computed hash for the given key
    while self.slots[h] != None:
        if self.slots[h].key == key:
            return self.slots[h].value
        h = (h+ 1) % self.size
    return None
```

Implementing a hash table as a dictionary

- Up to now, we use **put()** and **get()** to store and retrieve items from a hash table
- If we implement it as a dictionary, we can retrieve data with
  - **ht["good"]** instead of **ht.get("good")**
- Use special methods
  - **__setitem__()**
  - **__getitem__()**

Implementing a hash table as a dictionary

```
def __setitem__(self, key, value):
      self.put(key, value)
def __getitem__(self, key):
      return self.get(key)
```

- Example

```
ht = HashTable()
ht["good"] = "eggs"
ht["better"] = "ham"
ht["best"] = "spam"
ht["ad"] = "do not"
ht["ga"] = "collide"
for key in ("good", "better", "best", "worst", "ad", "ga"):
     v = ht[key]
     print(v)
print("The number of elements is: {}".format(ht.count))
```

Implementing a hash table as a dictionary

- Output

```
eggs
ham
spam
none
do not
collide
The number of elements is: 5
```

Quadratic probing

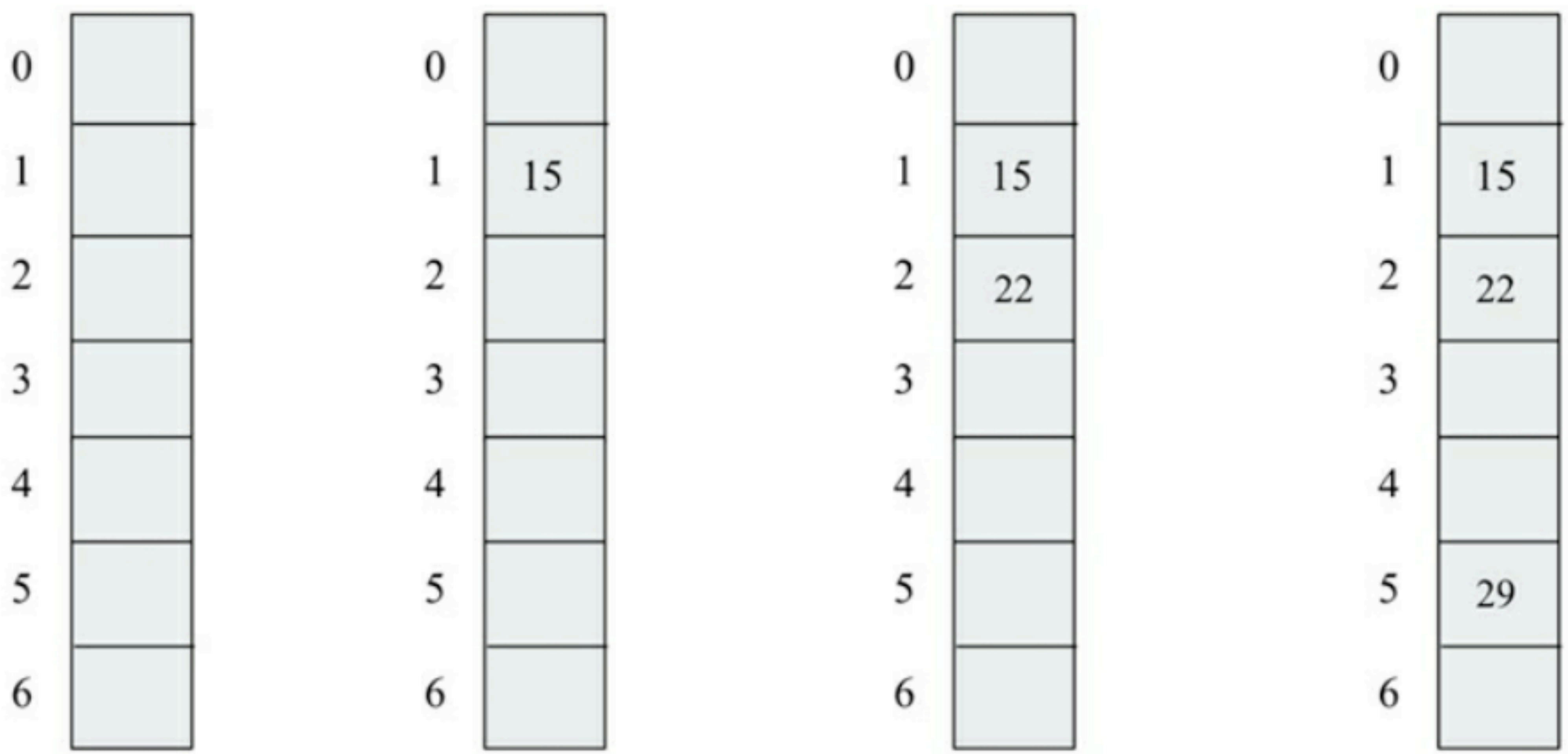
- If a collision occurs, try these locations:

$h + 1^2, h + 2^2, h + 3^2, h + 4^2$, and so on.

- Example: hash table with 7 elements
- Hash function:

`h(key) = key mod 7`.

Quadratic probing



Empty table

Add element - 15
(15 mod 7)= 1

Add element - 22
(22 mod 7)= 1.
Collision here.
New position = (1 + 1^2)

Add element-29, (29 mod 7)= 1.
Collision here.
New position = (1 + 1^2).
Collision again.
New position = (1 + 2^2) =5

Quadratic probing

- Suffers from *secondary clustering*
  - elements with the same hash value will have the same probe sequence

Quadratic probing

- Changed lines are highlighted

```
def get_quadratic(self, key):
    h = self._hash(key)
    j = 1
    while self.slots[h] != None:
        if self.slots[h].key == key:
            return self.slots[h].value
        h = (h+ j*j) % self.size
        j = j + 1
    return None
def put_quadratic(self, key, value):
    item = HashItem(key, value)
    h = self._hash(key)
    j = 1
    while self.slots[h] != None:
        if self.slots[h].key == key:
            break
        h = (h + j*j) % self.size
        j = j+1
    if self.slots[h] == None:
        self.count += 1
    self.slots[h] = item
    self.check_growth()
```

Double hashing

- Use two hashing functions
- When a collision occurs, use the second hash function to choose a new location

```
(h1(key)+i*h2(key))mod table_size
h1(key) = key mod table_size
```

- Second hash function should be
  - fast and easy to compute
  - Never result in 0
  - Be different from the first hash function

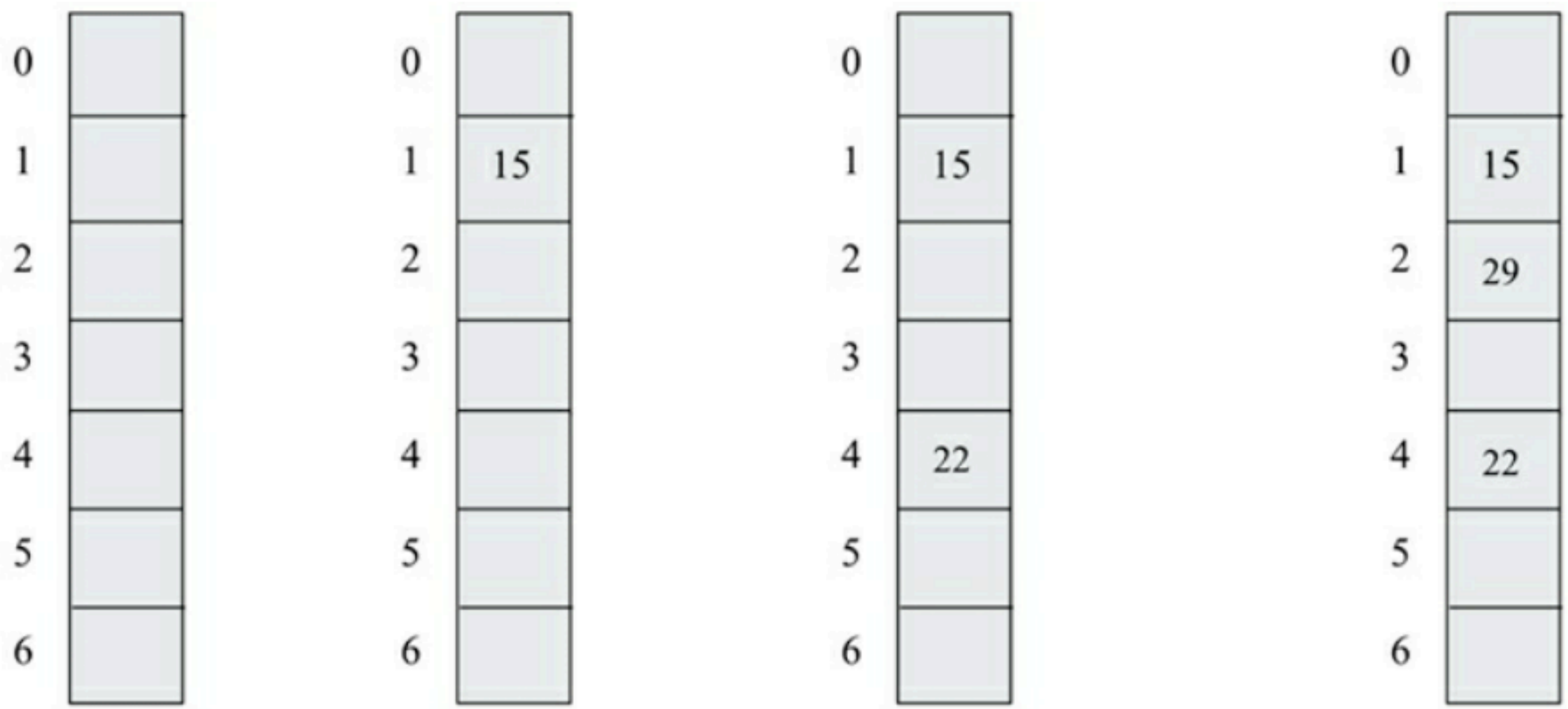
Double hashing

- A possible second hash function

```
h2(key) = prime_number - (key mod prime_number)
```

- Where **prime_number** is less than table size

Double hashing



Empty table

Add element - 15
(15 mod 7)= 1

Add element - 22
(22 mod 7)= 1.
Collision here. New position
= (1+ 1*(5-(22 mod 5))) mod 7
= (1+3)mod 7
= 4

Add element-29, (29 mod 7)= 1.
Collision here. New position
= (1+ 1*(5-(29 mod 5))) mod 7
= (1+1) mod 7
= 2

Double hashing

- Second hash function

```
def h2(self, key):
    mult = 1
    hv = 0
    for ch in key:
        hv += mult * ord(ch)
        mult += 1
    return hv
```

- HashTable defines **prime_num**

```
class HashTable:
    def __init__(self):
        self.size = 256
        self.slots = [None for i in range(self.size)]
        self.count = 0
        self.MAXLOADFACTOR = 0.65
        self.prime_num = 5
```

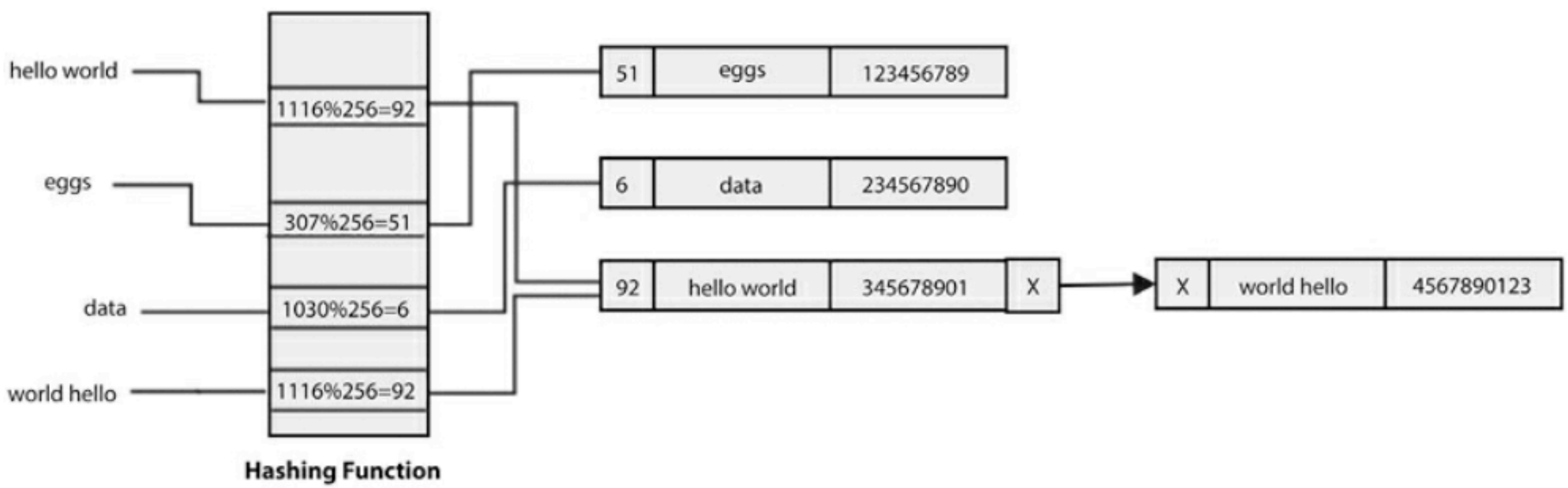
Double hashing

```
def put_double_hashing(self, key, value):
    item = HashItem(key, value)
    h = self._hash(key)
    j = 1
    while self.slots[h] != None:
        if self.slots[h].key == key:
            break
        h = (h + j * (self.prime_num - (self.h2(key) % self.prime_num))) % self.size
        j = j+1
    if self.slots[h] == None:
        self.count += 1
    self.slots[h] = item
    self.check_growth()

def get_double_hashing(self, key):
    h = self._hash(key)
    j = 1
    while self.slots[h] != None:
        if self.slots[h].key == key:
            return self.slots[h].value
        h = (h + j * (self.prime_num - (self.h2(key) % self.prime_num))) % self.size
        j = j + 1
    return None
```

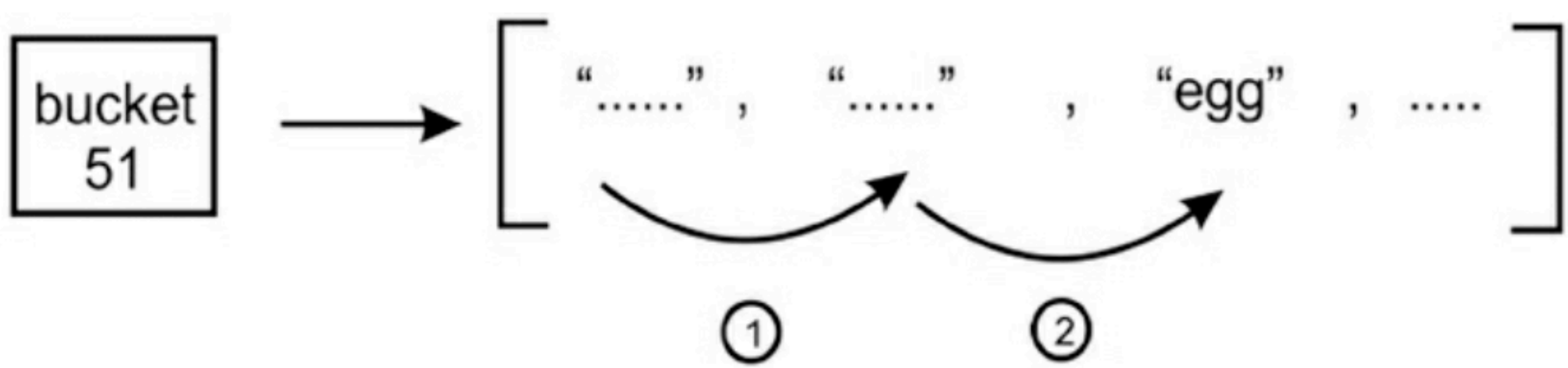
Separate chaining

- Another way to handle collisions
- Each slot in the hash table points to a list of stored values



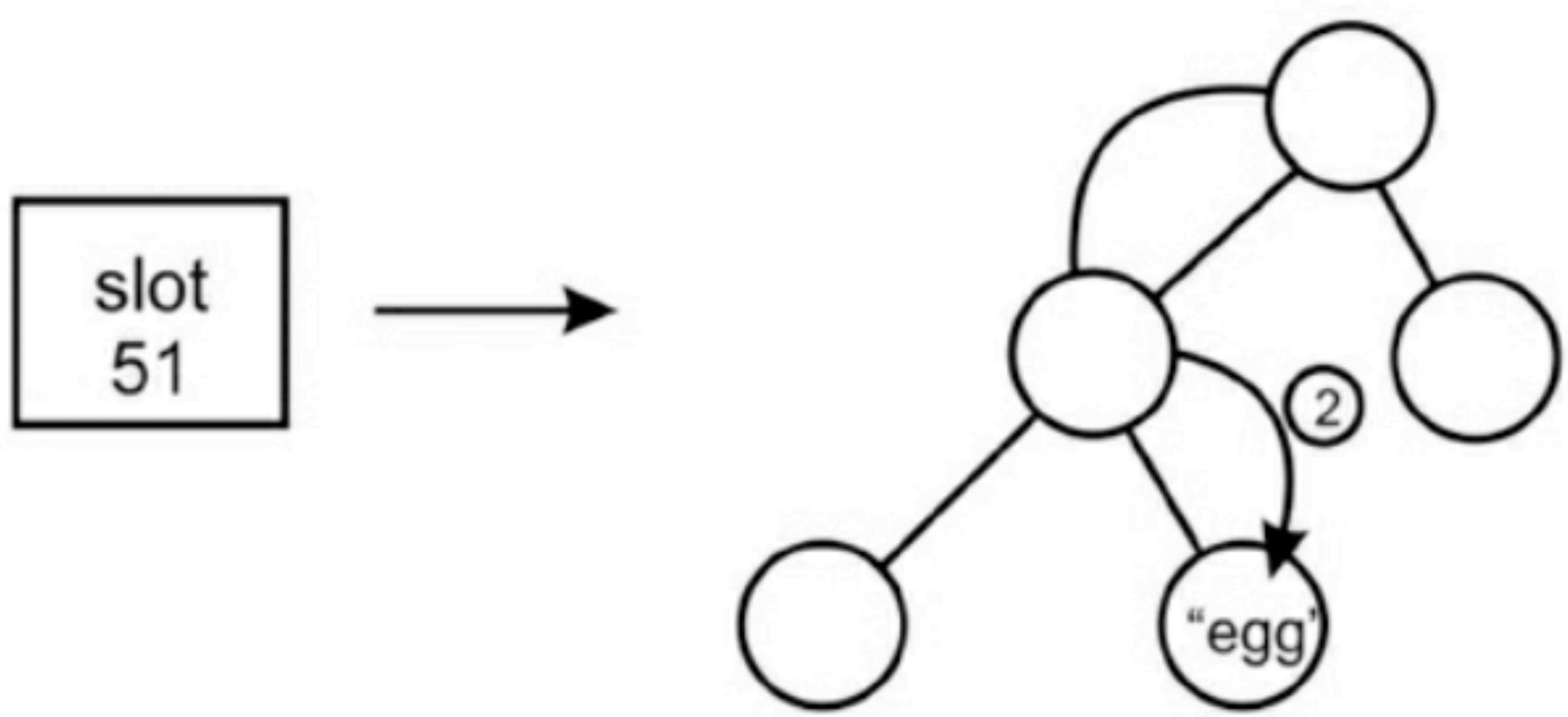
Separate chaining

- Slows down if hash table is full
- List searches can be $O(\boldsymbol{n})$



Separate chaining

- Better to use **Binary Search Trees** instead of lists



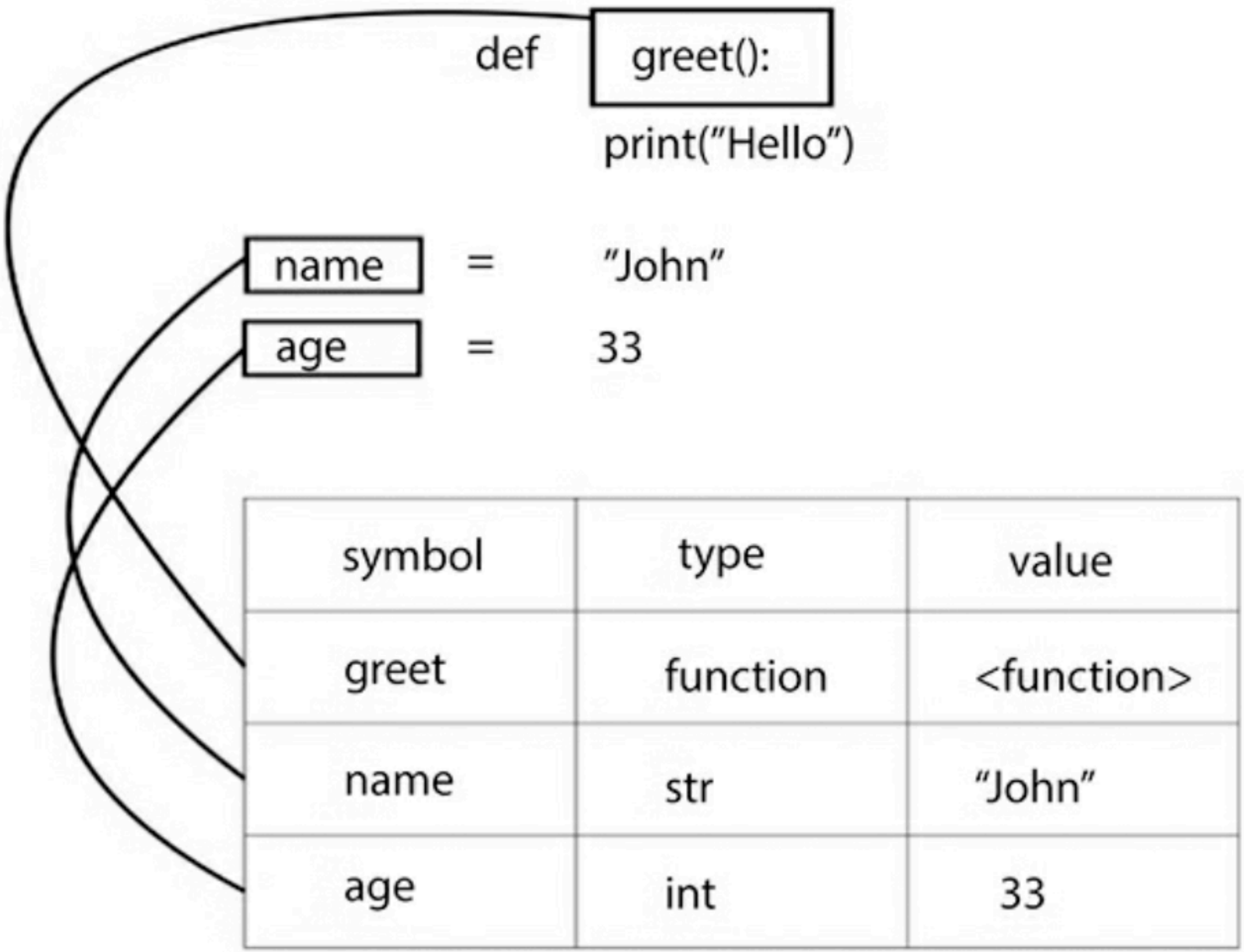
Symbol tables

- Used by compilers and interpreters
  - To keep track of the symbols and other entities in a program
    - Objects, classes, variables, function names
- Example
  - This program has two symbols
    - **name** and **age**

```
name = "Joe"
age = 27
```

Symbol tables

- x



Kahoot!

Ch 8