CSCI 136:
Data Structures
and
Advanced Programming

Lecture 32

Hash tables, part 4

Instructor: Dan Barowy

Williams

Announcements

Lab 10 Part 1 extended: due with Part 2

Part 2: MTF hashtable removed

All graded labs back this week (exc. 10)

Outline

Hashtable recap

Hashtables: big picture

We use **hash tables** when our problem meets **two criteria**:

1. A **table-like structure** is convenient (i.e., a `Map<K,V>`).
2. We are willing to trade an **ordering** of elements for **high performance** (**O(1)** put/get), on average.

Hashtables: big picture

What we **want**:

key → value

lunchMenu =

Monday	nuggets
Tuesday	mac n chez
Wednesday	pizza
Thursday	burgers
Friday	fish stix
Saturday	taco salad
Sunday	soup

Intuitively:
You can find what you're looking for quickly.

More generally, a **table ADT** is a `Map<K,V>`.

Hashtables: big picture

In a table, the **key** must be **unique**.

key → value

lunchMenu =

Monday	nuggets
Tuesday	mac n chez
Wednesday	pizza
Thursday	burgers
Friday	fish stix
Saturday	taco salad
Sunday	soup

Hashtables: big picture

Also, the **order of entries** are not particularly important.

key → value

lunchMenu =

Friday	fish stix
Tuesday	mac n chez
Sunday	soup
Monday	nuggets
Wednesday	pizza
Thursday	burgers
Saturday	taco salad

Hashtables: big picture

We always build complex data structures **from more primitive parts**.

What we **have**:

0	1	2	3	4	5	6	7

An **array**. **Not a table!**

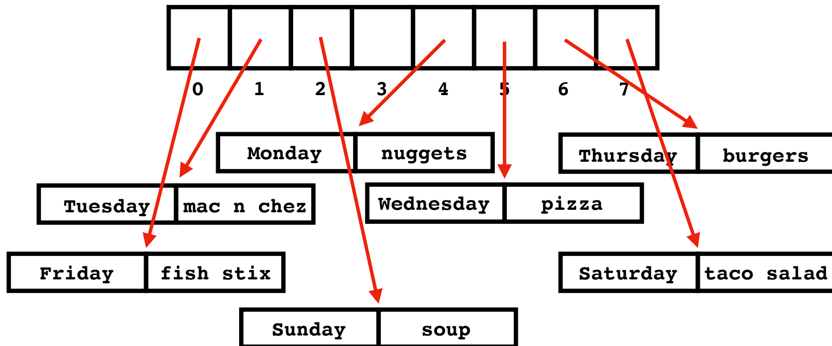
Monday	nuggets

Also, **associations**. Also **not a table!**

But with one more ingredient: **a table!**

Hashtables: big picture

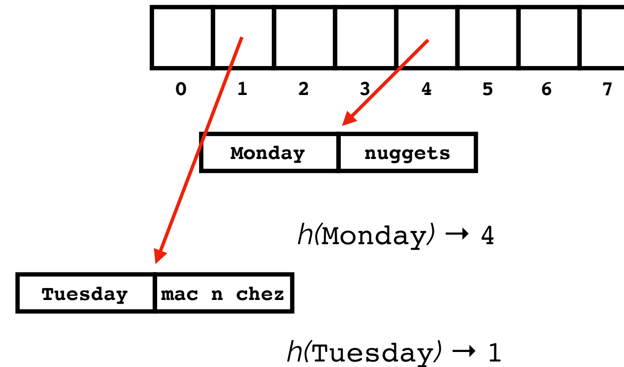
Let's **build a table** from our **parts**.



Close. But why did we **insert** in those **buckets**?

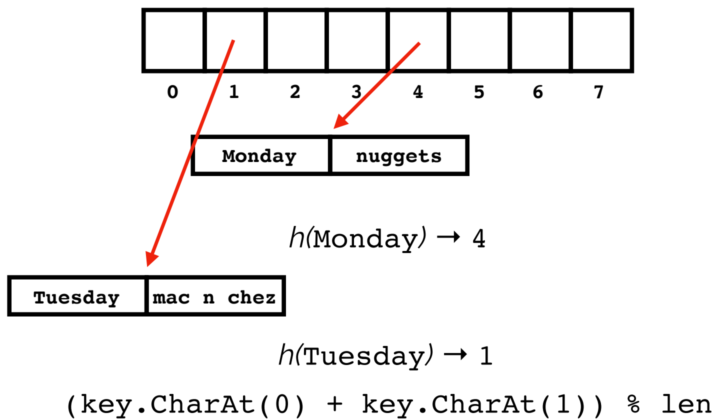
Hashtables: big picture

Last ingredient: a **function**
that maps **keys** to array **indices**.



Hashtables: big picture

How might we obtain such a function?
Any function from `T → int` will do.

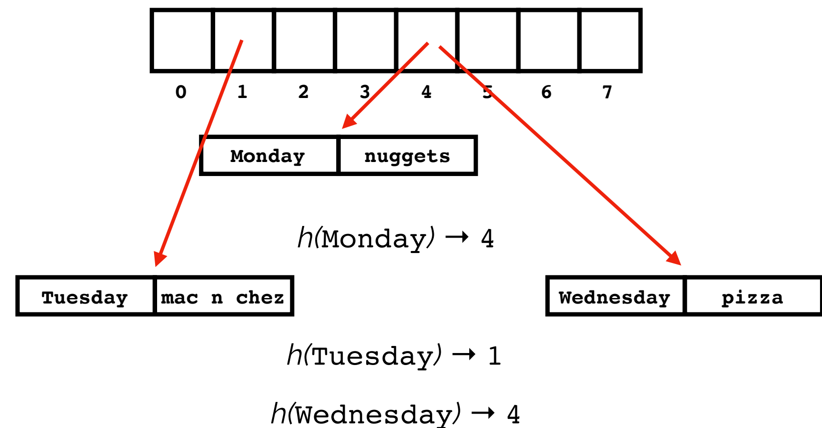



```
(key.CharAt(0) + key.CharAt(1)) % len
```

```
(key.CharAt(0) + key.CharAt(1)) % len
```

Hashtables: big picture

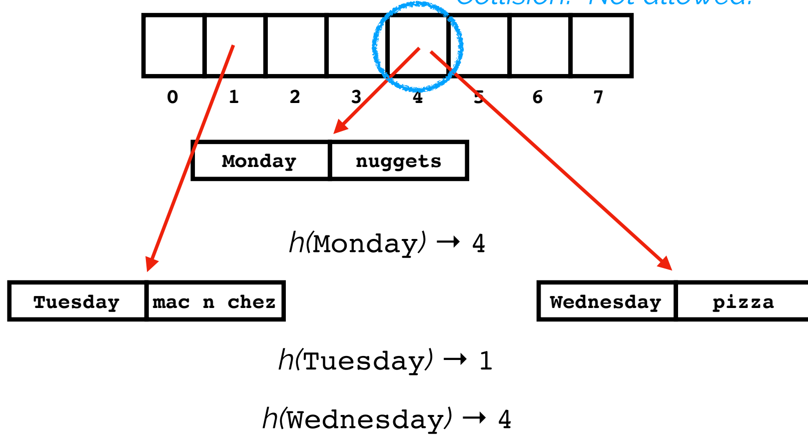
Some functions are **better** than others.



```
(key.CharAt(0) + key.CharAt(1)) % len
```

Hashtables: big picture

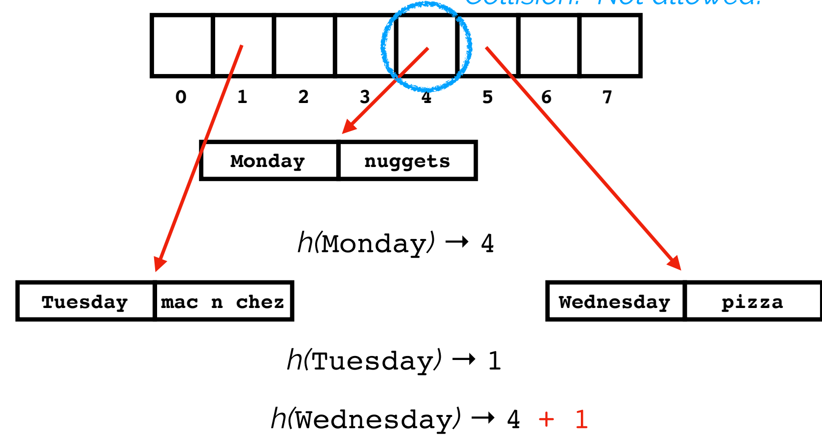
Some functions are **better** than others.




```
(key.CharAt(0) + key.CharAt(1)) % len
```

Hashtables: big picture

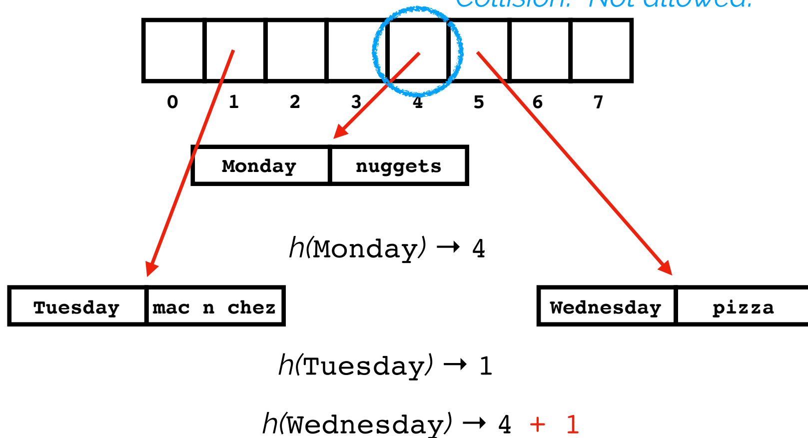
Some functions are **better** than others.




```
(key.CharAt(0) + key.CharAt(1)) % len
```

Hashtables: big picture

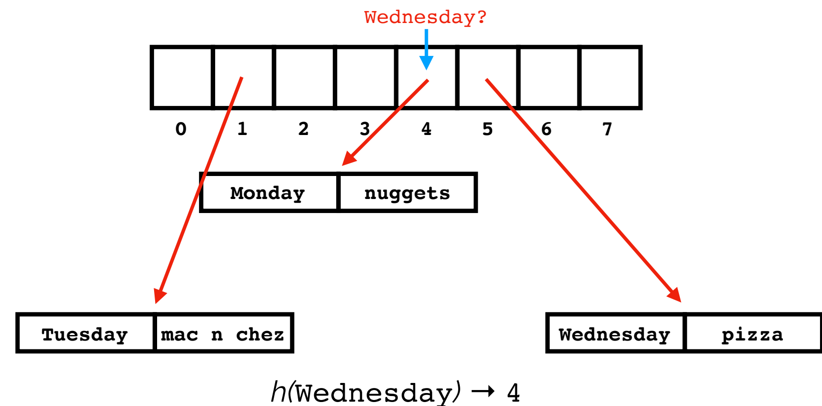
When hashed keys **collide**, we need a **recovery method**.




```
(key.CharAt(0) + key.CharAt(1)) % len
```

Hashtables: big picture

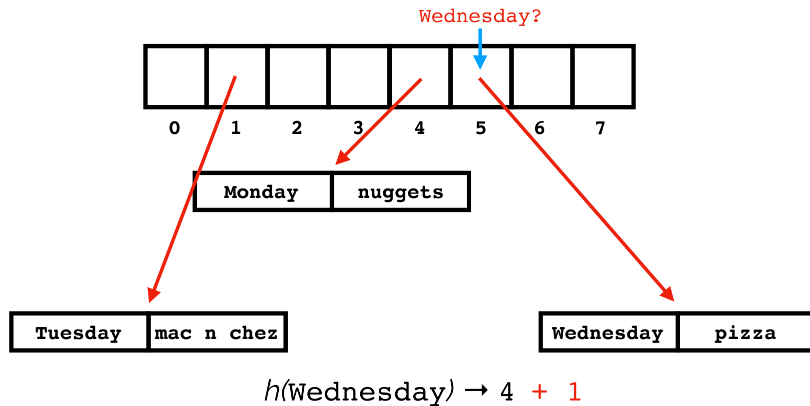
This turns out to be important on **lookup** too.



`(key.CharAt(0) + key.CharAt(1)) % len`

Hashtables: big picture

This turns out to be important on **lookup** too.



h(Wednesday) → 4 + 1

Hashtable recipe

Ingredients:

1. Array (or array-like structure).
2. Association of key and value.
3. Hash function.
4. Collision recovery method.

Insertion (or lookup) procedure:

1. Hash the key to obtain index.
2. If index is occupied (not what we're looking for), find new index using recovery method.
3. Insert (or return) key.

Hashtable expansion

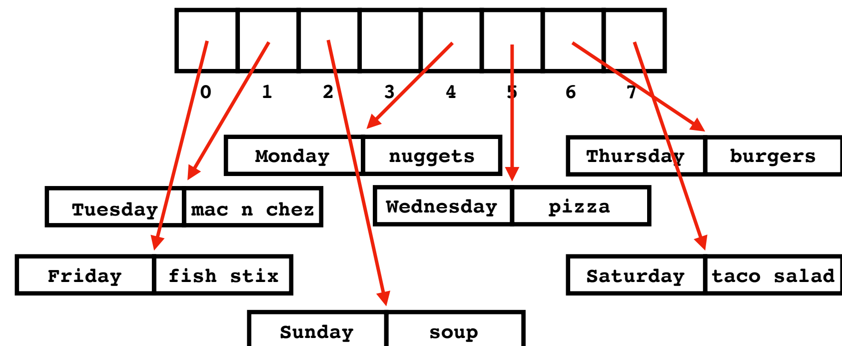
A `Vector` is a convenient way to store key-value Associations in a hash table because, when the hash table fills up, a `Vector` can expand.

But: on expansion, all keys need to be **hashed again and reinserted**. Why? Because hash value depends on `Vector` **length**.

`(key.CharAt(0) + key.CharAt(1)) % len`

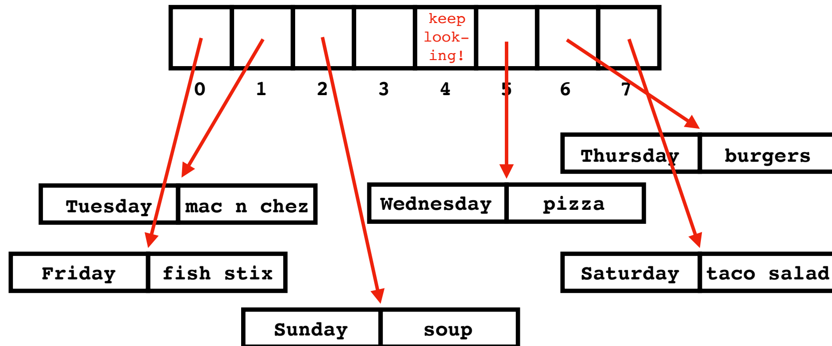
Hashtable deletion

Also: **deletion** of a key may require leaving a **placeholder** behind.



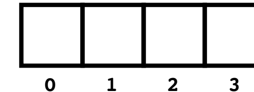
Hashtable deletion

Also: **deletion** of a key may require leaving a **placeholder** behind.



`HashAssociation` has a `reserved` flag for this purpose.

Hashtable activity




```
(key.CharAt(0) + key.CharAt(1)) % len
```

Harry Potter, Gryffindor
Draco Malfoy, Slytherin
Luna Lovegood, Ravenclaw
Gilderoy Lockhart, Ravenclaw
Cedric Diggory, Hufflepuff
Hermione Grainger, Gryffindor
Dolores Umbridge, Slytherin

Recap & Next Class

Today we learned:

Hashtable recap

Next class:

Shortest paths