Collections, Part Three

Outline for Today

- ***Lexicon***
  - Storing a collection of words.
- ***Set***
  - Storing a group of whatever you'd like.
- ***Map***
  - A powerful, fundamental container.

Lexicon

Lexicon

- A **`Lexicon`** is a container that stores a collection of words.
- The `Lexicon` is designed to answer the following question efficiently:

  Given a word, is it contained in the `Lexicon`?

- The `Lexicon` does *not* support access by index. You can't, for example, ask what the 137th English word is.
- However, it *does* support questions of the form "does this word exist?" or "do any words have this as a prefix?"

Tautonyms

- A ***tautonym*** is a word formed by repeating the same string twice.
  - For example: murmur, couscous, papa, etc.
- What English words are tautonyms?

Some Aa

http://upload.wikimedia.org/wikipedia/commons/f/f1/Aa_large.jpg

One Bulbul

More than One Caracara

http://www.greglasley.net/images/CO/Crested-Caracara-F3.jpg

Introducing the Dikdik

And a Music Recommendation

Time-Out for Announcements!

MLK Weekend

- We have this Monday off for MLK weekend. Our next class will be this upcoming Wednesday.
- The LaIR will be open on Monday night as usual, but will be closed on Sunday.
- Some suggested reading / listening / watching recommendations:
  - “The Autobiography of Malcolm X,” as told to Alex Haley.
  - “The Ballot or the Bullet” by Malcolm X.
  - “Between the World and Me” by Ta-Nehisi Coates.
  - “The Case for Reparations” by Ta-Nehisi Coates.
  - “Debate at Cambridge Union,” James Baldwin and William F. Buckley, Jr.
  - “Do Artifacts Have Politics?” by Langdon Winner.
  - “Letter from Birmingham City Jail” by Martin Luther King, Jr.
  - “Letter from a Region in my Mind” by James Baldwin.
  - “Notes on an Imagined Plaque” by The Memory Palace.
  - “The Other America” by Martin Luther King, Jr.

Assignment 2

- Assignment 2 (Fun with Collections) goes out today. It's due next Friday at 10:30AM.
  - Determine the language a piece of text is written in.
  - Explore the impact of sea level rise.
- We've provided a suggested timetable for completing this assignment at the top of the assignment description. Aim to stick to this timeline; you've got plenty of time to complete things if you start early.
- ***You must complete this assignment individually***. Working in pairs is not permitted on this assignment.

YEAH Hours

- We'll be holding review sessions for most of our assignments to help give you an overview of what's involved.
- We lovingly call these ***YEAH Hours*** (***Y***our ***E***arly ***A***ssignment ***H***elp).
- YEAH Hours for Assignment 2 will be today at 4PM Pacific over Zoom ***using this link*** (also available on EdStem).
- Can't make it? No worries! They'll be recorded.

Come to Black LaIR for Assignment Help!

Black LaIR is open to ****everyone**** for **conceptual** and **debugging help** on **CS106B Assignments**. We hope to see you there!

What:

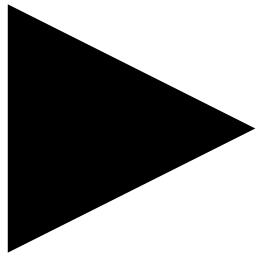
Virtual, one-on-one **conceptual** and **debugging help** sessions for CS106A and CS106B held through **QueueStatus** and **Zoom**.

When:

- **Tuesdays:** 5 - 8 PM PST
- **Thursdays:** 5 - 8 PM PST
- **Saturdays:** 12 - 3 PM PST

CS106B QueueStatus: **https://queuestatus.com/queues/1149**

Visit **https://blackincs.stanford.edu/black-lair** to learn more, and reach out to **ajarno@stanford.edu** if you have any questions or concerns!



Set

Set

- The `Set` represents an unordered collection of distinct elements.
- Elements can be added and removed. Duplicates aren’t allowed.

Set

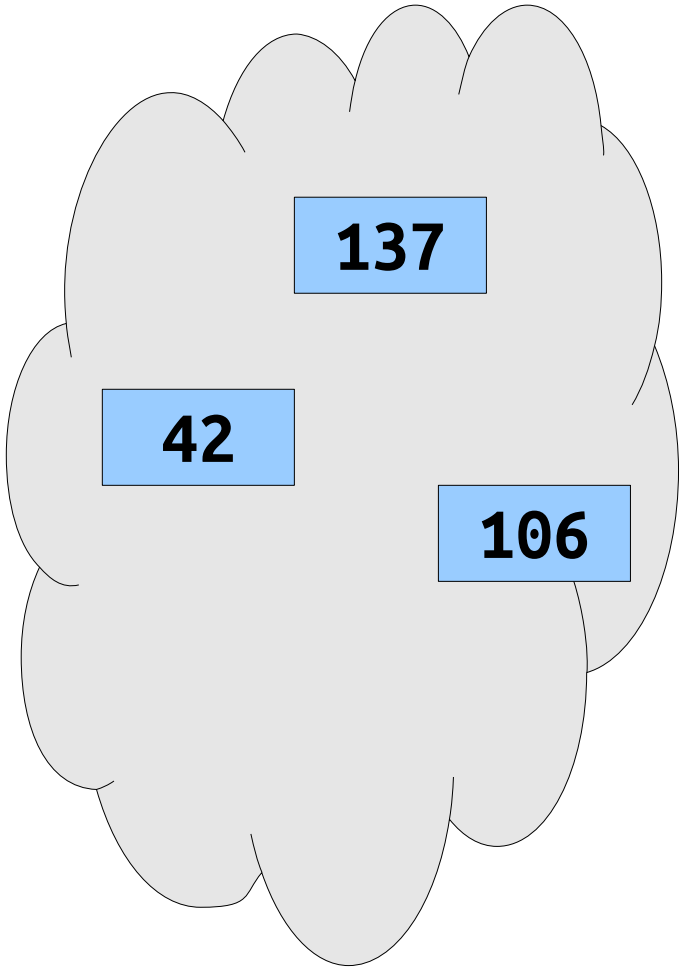
- The **Set** represents an unordered collection of distinct elements.
- Elements can be added and removed. Duplicates aren't allowed.

```
Set<int> values = {137, 106, 42};
```

Set

- The **Set** represents an unordered collection of distinct elements.
- Elements can be added and removed. Duplicates aren't allowed.

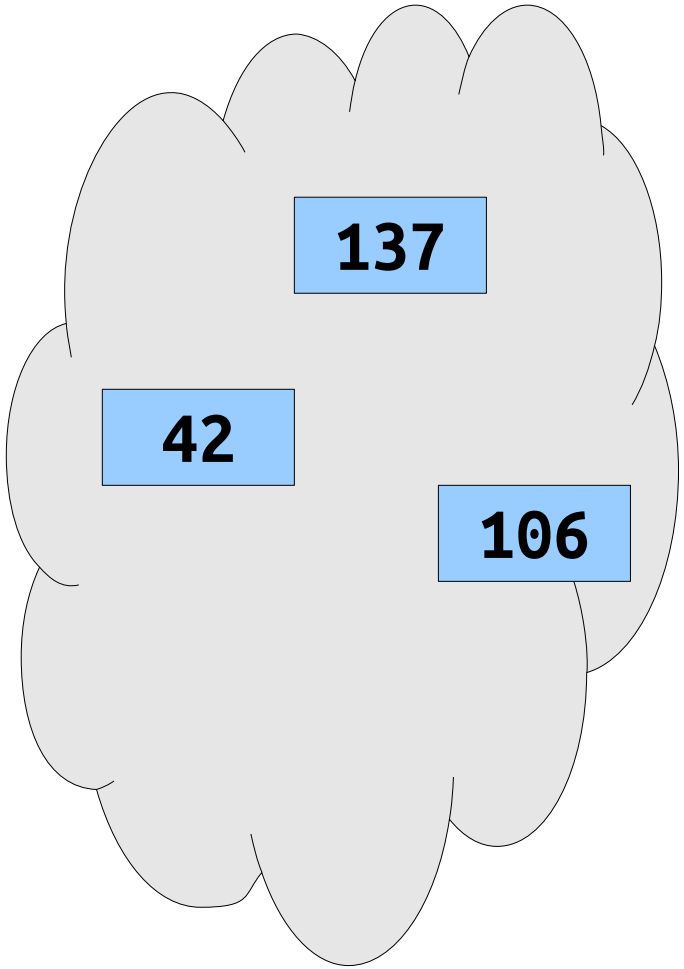
```
Set<int> values = {137, 106, 42};
```



Set

- The **Set** represents an unordered collection of distinct elements.
- Elements can be added and removed. Duplicates aren't allowed.

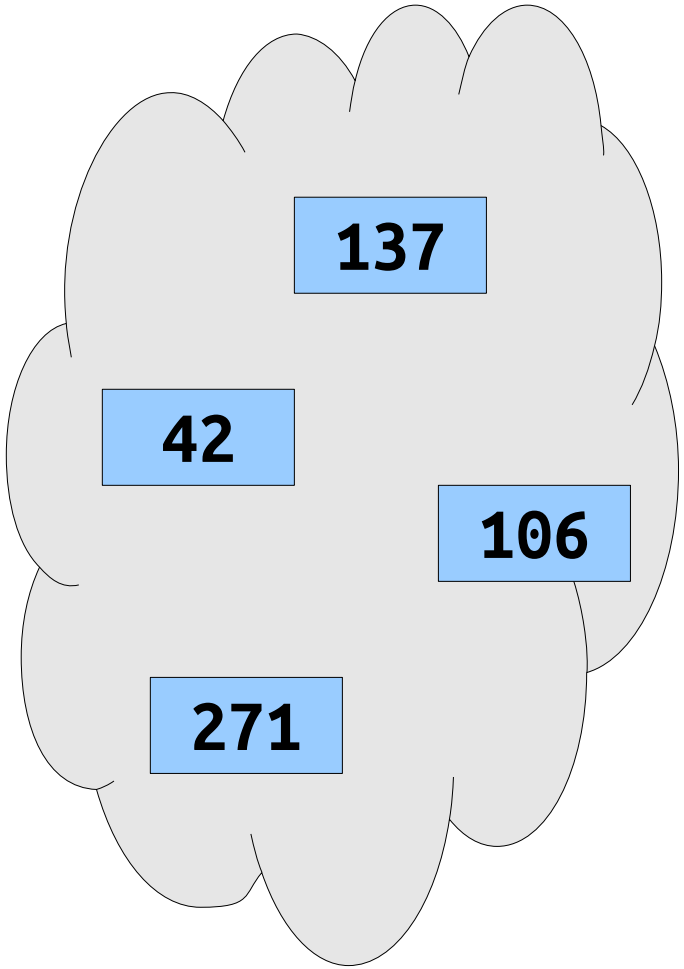
```
Set<int> values = {137, 106, 42};
values += 271;
```



Set

- The **Set** represents an unordered collection of distinct elements.
- Elements can be added and removed. Duplicates aren't allowed.

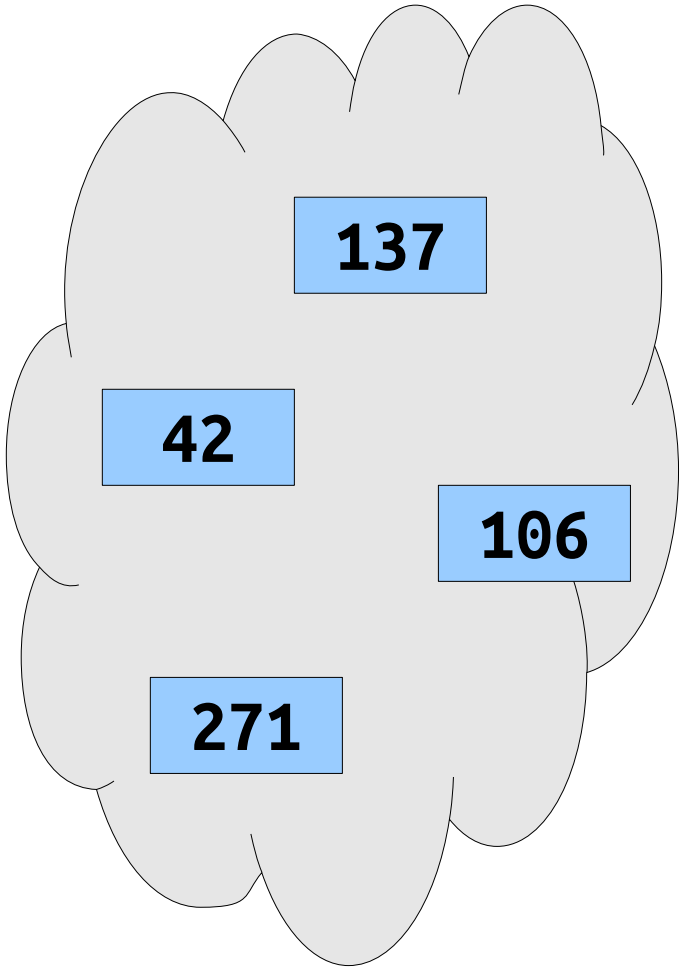
```
Set<int> values = {137, 106, 42};
values += 271;
```



Set

- The **Set** represents an unordered collection of distinct elements.
- Elements can be added and removed. Duplicates aren't allowed.

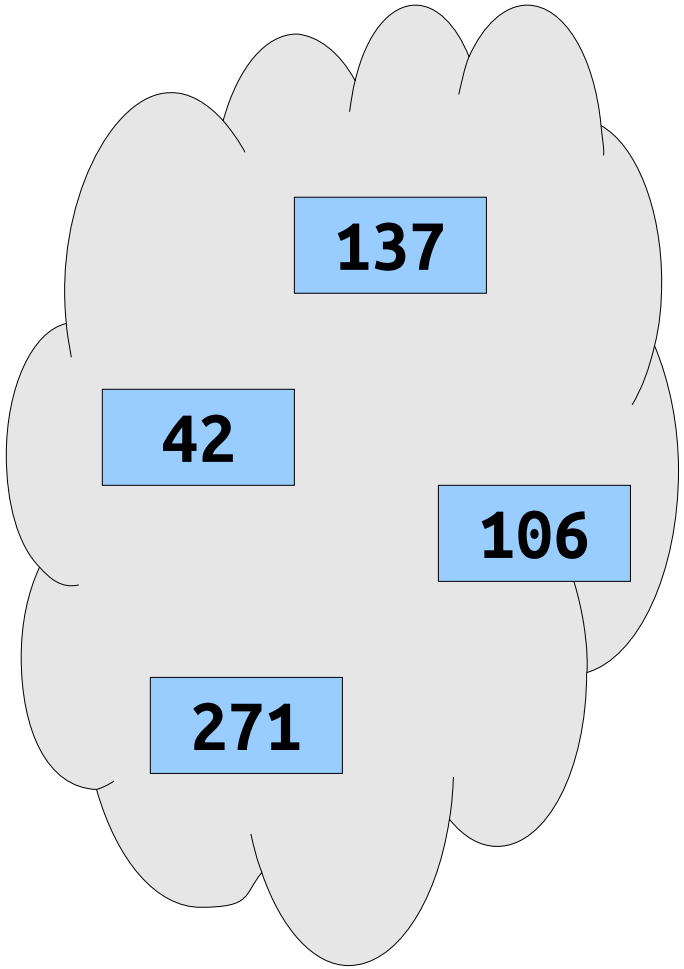
```
Set<int> values = {137, 106, 42};

values += 271;
values += 271;
```



Set

- The **Set** represents an unordered collection of distinct elements.
- Elements can be added and removed. Duplicates aren't allowed.

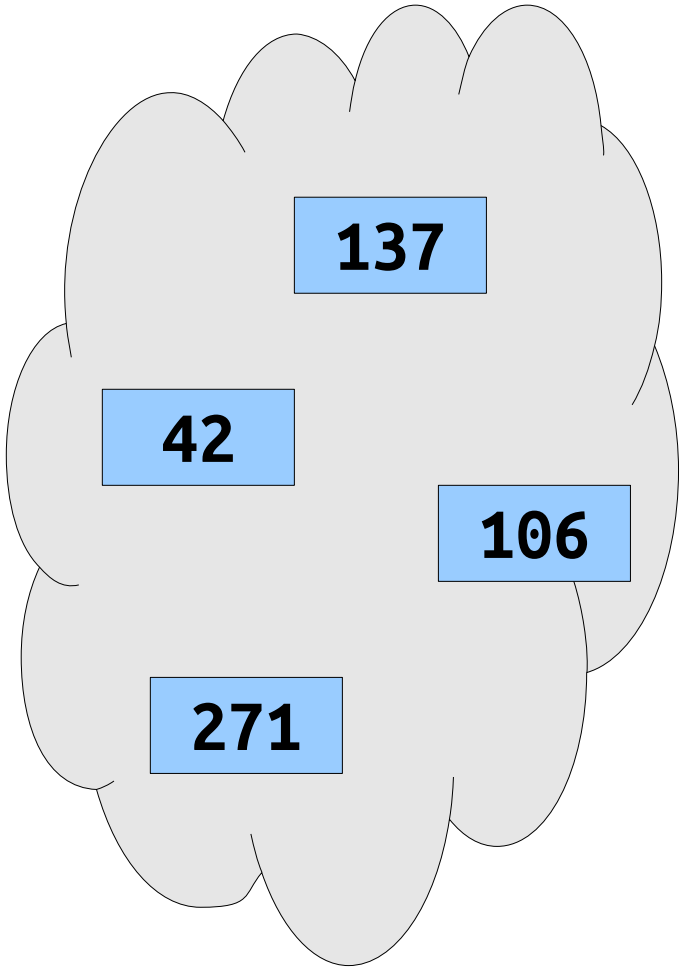
```
Set<int> values = {137, 106, 42};

values += 271;
values += 271; // Has no effect
```



Set

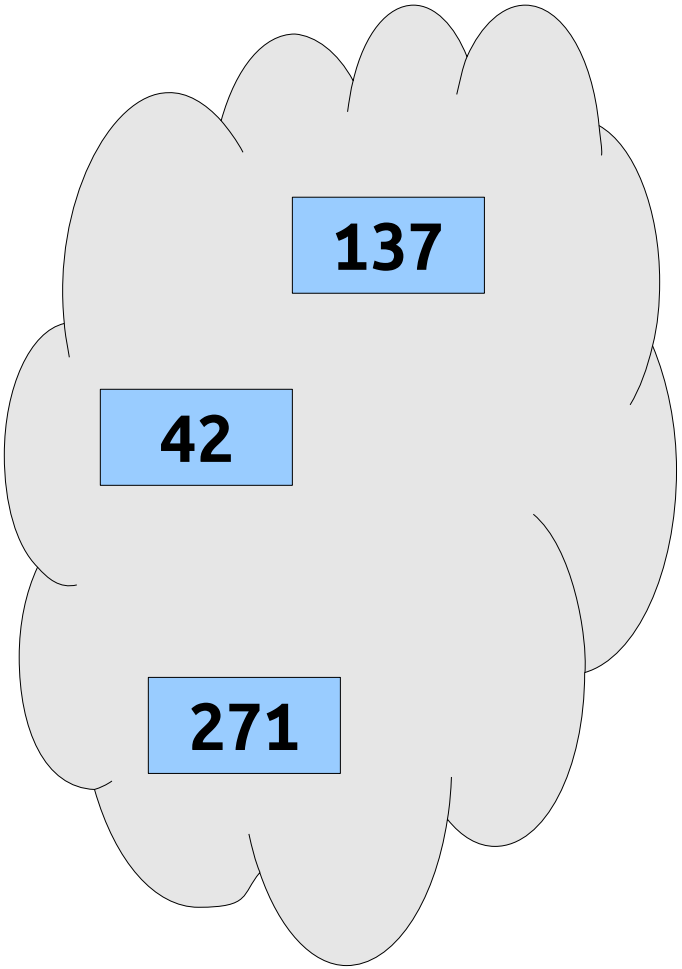
- The **Set** represents an unordered collection of distinct elements.
- Elements can be added and removed. Duplicates aren't allowed.

```
Set<int> values = {137, 106, 42};

values += 271;
values += 271; // Has no effect

values -= 106;
```



Set

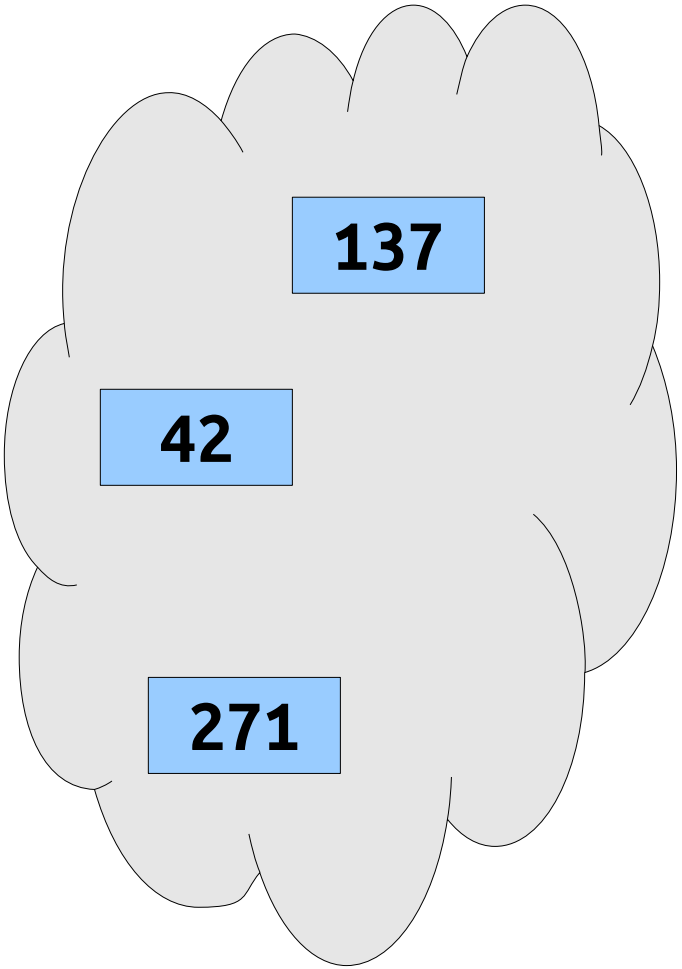
- The **Set** represents an unordered collection of distinct elements.
- Elements can be added and removed. Duplicates aren't allowed.




```
Set<int> values = {137, 106, 42};

values += 271;
values += 271; // Has no effect

values -= 106;
```

Set

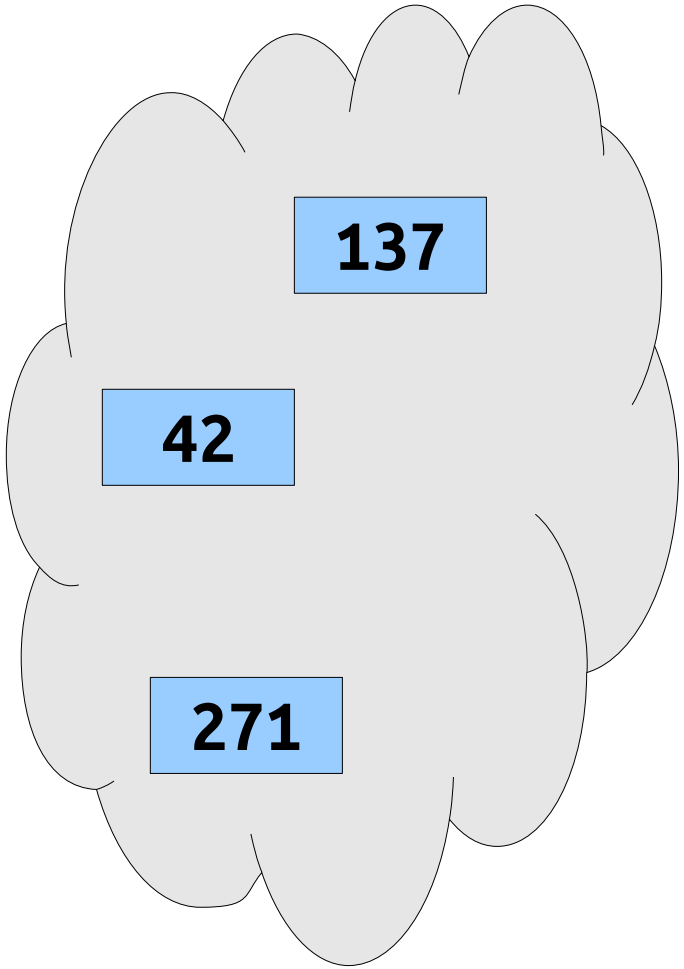
- The **Set** represents an unordered collection of distinct elements.
- Elements can be added and removed. Duplicates aren't allowed.




```
Set<int> values = {137, 106, 42};

values += 271;
values += 271; // Has no effect

values -= 106;
values -= 103;
```

Set

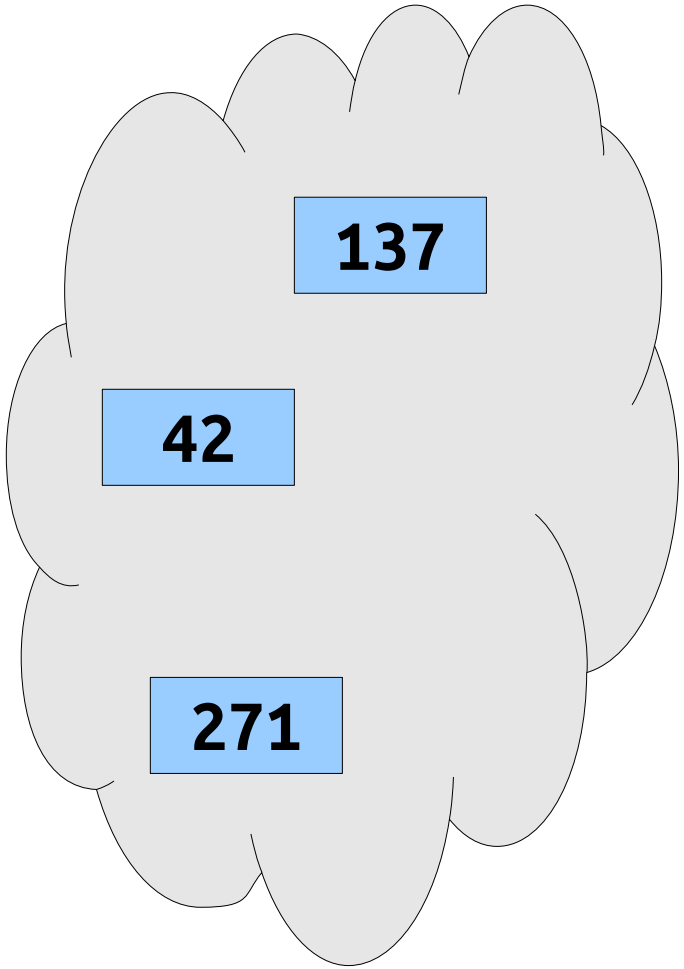
- The **Set** represents an unordered collection of distinct elements.
- Elements can be added and removed. Duplicates aren't allowed.




```
Set<int> values = {137, 106, 42};

values += 271;
values += 271; // Has no effect

values -= 106;
values -= 103; // Has no effect
```

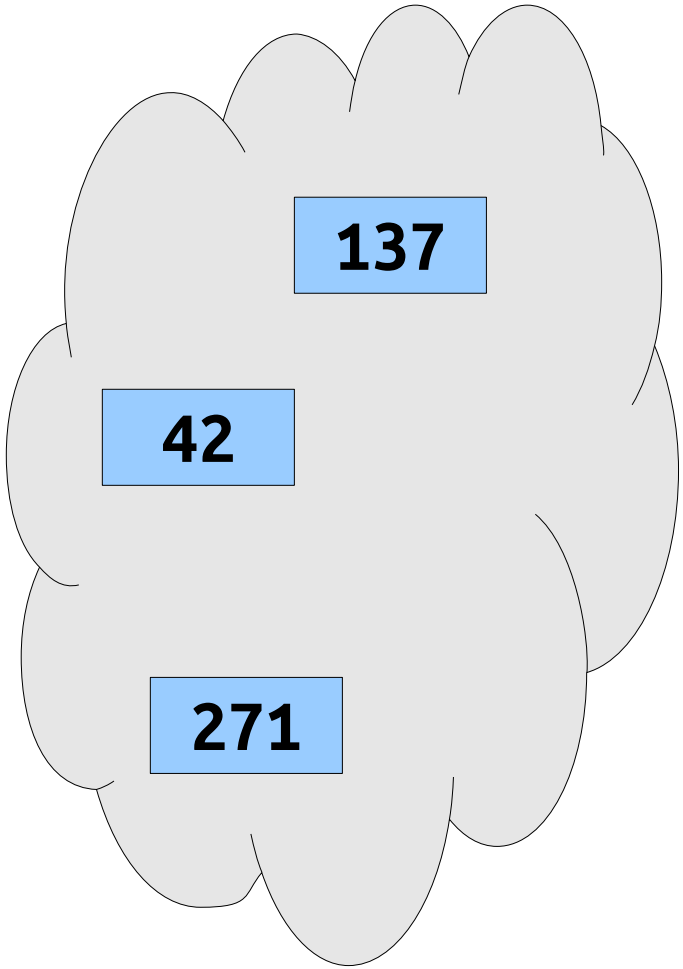
Set

- The `Set` represents an unordered collection of distinct elements.
- Elements can be added and removed. Duplicates aren't allowed.
- You may find it helpful to interpret += as "ensure this item is there" and -= as "ensure this item isn't there."



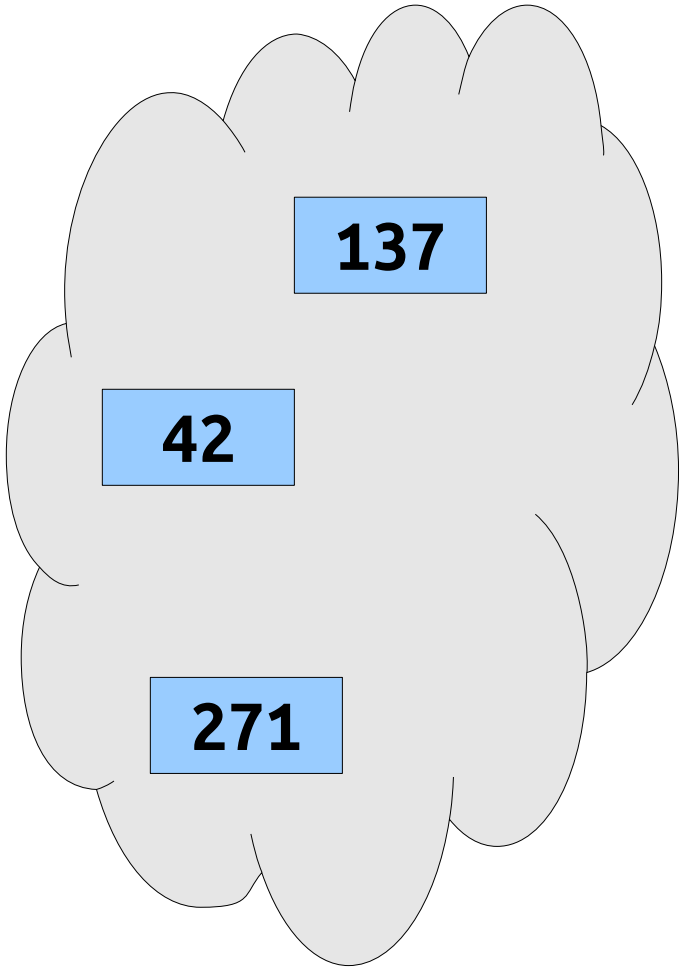
Set

- Sets make it easy to check if you've seen something before.



Set

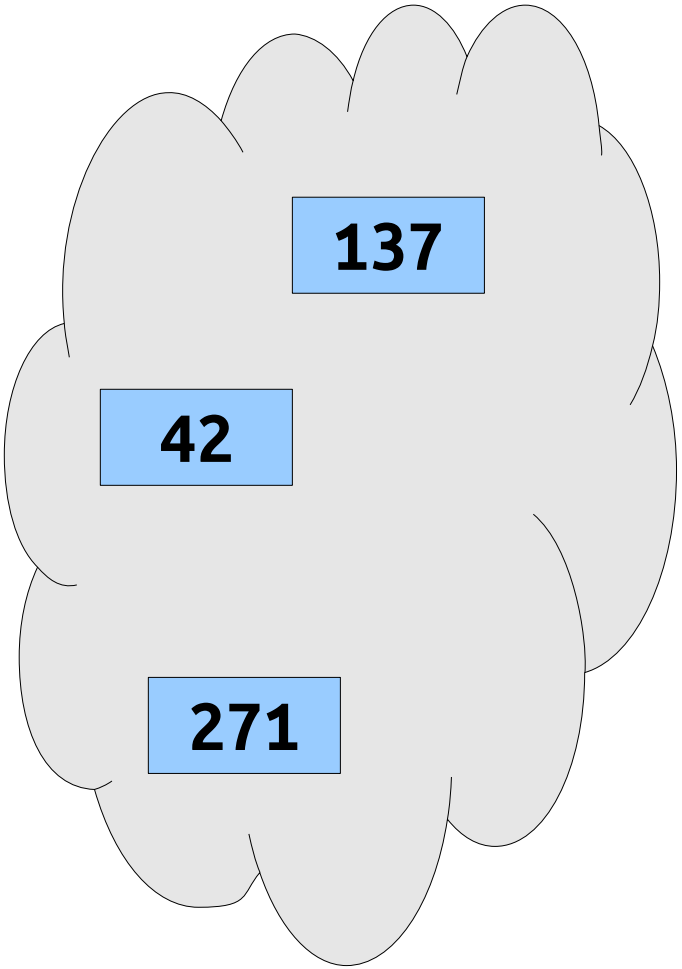
- Sets make it easy to check if you've seen something before.




```
if (values.contains(137)) {
    cout << "<(^_^)>" << endl;
}
```

Set

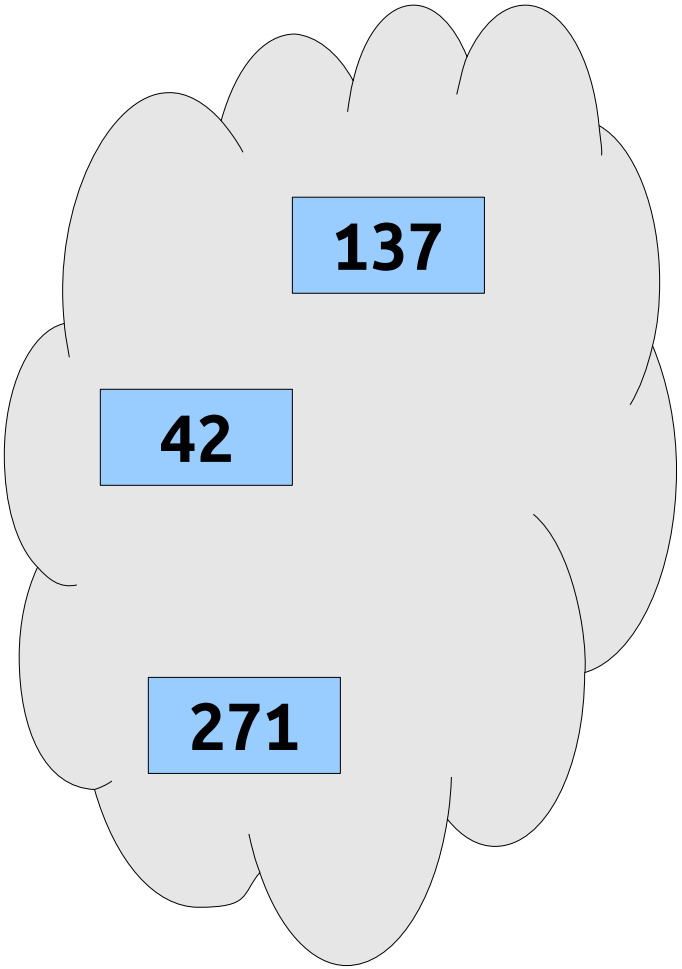
- Sets make it easy to check if you've seen something before.
- You can loop over the contents of a set with a range-based `for` loop.




```
if (values.contains(137)) {
    cout << "<(^_^)>" << endl;
}
```

Set

- Sets make it easy to check if you've seen something before.
- You can loop over the contents of a set with a range-based **for** loop.




```
if (values.contains(137)) {
    cout << "<(^_^)>" << endl;
}

for (int value: values) {
    cout << value << endl;
}
```

Why Sets?

- Imagine you're maintaining a shopping list as a `Vector`.

Why Sets?

- Imagine you're maintaining a shopping list as a `Vector`.

```
Vector<string> toBuy;
```

Why Sets?

- Imagine you're maintaining a shopping list as a `Vector`.

```
Vector<string> toBuy;
toBuy += "Scotch bonnet";
```

Why Sets?

- Imagine you're maintaining a shopping list as a `Vector`.


```
Vector<string> toBuy;
toBuy += "Scotch bonnet";
```

Why Sets?

- Imagine you're maintaining a shopping list as a `Vector`.


```
Vector<string> toBuy;
toBuy += "Scotch bonnet";
toBuy += "Tomatoes";
```

Why Sets?

- Imagine you're maintaining a shopping list as a `Vector`.

0	Scotch Bonnet
1	Tomatoes

```
Vector<string> toBuy;
toBuy += "Scotch bonnet";
toBuy += "Tomatoes";
```

Why Sets?

- Imagine you're maintaining a shopping list as a `Vector`.

0	Scotch Bonnet
1	Tomatoes

```
Vector<string> toBuy;
toBuy += "Scotch bonnet";
toBuy += "Tomatoes";
toBuy += "Rice";
```

Why Sets?

- Imagine you're maintaining a shopping list as a `Vector`.

0	Scotch Bonnet
1	Tomatoes
2	Rice

```
Vector<string> toBuy;
toBuy += "Scotch bonnet";
toBuy += "Tomatoes";
toBuy += "Rice";
```

Why Sets?

- Imagine you're maintaining a shopping list as a `Vector`.

0	Scotch Bonnet
1	Tomatoes
2	Rice

```
Vector<string> toBuy;
toBuy += "Scotch bonnet";
toBuy += "Tomatoes";
toBuy += "Rice";
toBuy += "Curry powder";
```

Why Sets?

- Imagine you're maintaining a shopping list as a `Vector`.

0	Scotch Bonnet
1	Tomatoes
2	Rice
3	Curry Powder

```
Vector<string> toBuy;
toBuy += "Scotch bonnet";
toBuy += "Tomatoes";
toBuy += "Rice";
toBuy += "Curry powder";
```

Why Sets?

- Imagine you're maintaining a shopping list as a `Vector`.
- If you add the same item twice, it shows up twice.

0	Scotch Bonnet
1	Tomatoes
2	Rice
3	Curry Powder

```
Vector<string> toBuy;
toBuy += "Scotch bonnet";
toBuy += "Tomatoes";
toBuy += "Rice";
toBuy += "Curry powder";
```

Why Sets?

- Imagine you're maintaining a shopping list as a `Vector`.
- If you add the same item twice, it shows up twice.

0	Scotch Bonnet
1	Tomatoes
2	Rice
3	Curry Powder

```
Vector<string> toBuy;
toBuy += "Scotch bonnet";
toBuy += "Tomatoes";
toBuy += "Rice";
toBuy += "Curry powder";
toBuy += "Rice";
```

Why Sets?

- Imagine you're maintaining a shopping list as a `Vector`.
- If you add the same item twice, it shows up twice.

0	Scotch Bonnet
1	Tomatoes
2	Rice
3	Curry Powder
4	Rice

```
Vector<string> toBuy;
toBuy += "Scotch bonnet";
toBuy += "Tomatoes";
toBuy += "Rice";
toBuy += "Curry powder";
toBuy += "Rice";
```

Why Sets?

- Imagine you're maintaining a shopping list as a `Vector`.
- If you add the same item twice, it shows up twice.

0	Scotch Bonnet
1	Tomatoes
2	Rice
3	Curry Powder
4	Rice

```
Vector<string> toBuy;
toBuy += "Scotch bonnet";
toBuy += "Tomatoes";
toBuy += "Rice";
toBuy += "Curry powder";
toBuy += "Rice"; // Oops...
```

Why Sets?

- Imagine you're maintaining a shopping list as a `Vector`.
- If you add the same item twice, it shows up twice.
- Each item has a numeric position, which you don't care about but need to know to remove things.

0	Scotch Bonnet
1	Tomatoes
2	Rice
3	Curry Powder
4	Rice

```
Vector<string> toBuy;
toBuy += "Scotch bonnet";
toBuy += "Tomatoes";
toBuy += "Rice";
toBuy += "Curry powder";
toBuy += "Rice"; // Oops...
```

Why Sets?

- Imagine you're maintaining a shopping list as a `Vector`.
- If you add the same item twice, it shows up twice.
- Each item has a numeric position, which you don't care about but need to know to remove things.

0	Scotch Bonnet
1	Tomatoes
2	Rice
3	Curry Powder
4	Rice

```
Vector<string> toBuy;
toBuy += "Scotch bonnet";
toBuy += "Tomatoes";
toBuy += "Rice";
toBuy += "Curry powder";
toBuy += "Rice"; // Oops...

toBuy.remove(1); // Not "remove
                 // tomatoes."
```

Why Sets?

- Imagine you're maintaining a shopping list as a `Vector`.
- If you add the same item twice, it shows up twice.
- Each item has a numeric position, which you don't care about but need to know to remove things.

0	Scotch Bonnet

2	Rice
3	Curry Powder
4	Rice

```
Vector<string> toBuy;
toBuy += "Scotch bonnet";
toBuy += "Tomatoes";
toBuy += "Rice";
toBuy += "Curry powder";
toBuy += "Rice"; // Oops...

toBuy.remove(1); // Not "remove
                 // tomatoes."
```

Why Sets?

- Imagine you're maintaining a shopping list as a `Vector`.
- If you add the same item twice, it shows up twice.
- Each item has a numeric position, which you don't care about but need to know to remove things.
- When an item gets removed, other items have their positions shifted, which takes extra time.

0	Scotch Bonnet

2	Rice
3	Curry Powder
4	Rice

```
Vector<string> toBuy;
toBuy += "Scotch bonnet";
toBuy += "Tomatoes";
toBuy += "Rice";
toBuy += "Curry powder";
toBuy += "Rice"; // Oops...

toBuy.remove(1); // Not "remove
                 // tomatoes."
```

Why Sets?

- Imagine you're maintaining a shopping list as a `Vector`.
- If you add the same item twice, it shows up twice.
- Each item has a numeric position, which you don't care about but need to know to remove things.
- When an item gets removed, other items have their positions shifted, which takes extra time.

0	Scotch Bonnet
1	Rice
2	Curry Powder
3	Rice

```
Vector<string> toBuy;
toBuy += "Scotch bonnet";
toBuy += "Tomatoes";
toBuy += "Rice";
toBuy += "Curry powder";
toBuy += "Rice"; // Oops...

toBuy.remove(1); // Not "remove
                 // tomatoes."
```

Why Sets?

- A shopping list is a great place to use a `Set`.
- Sets ignore duplicates, so adding an existing item has no effect.
- There's no notion of "the first item on the list," which matches how you use a shopping list.
- Removing items is easy; no indices need to be adjusted.

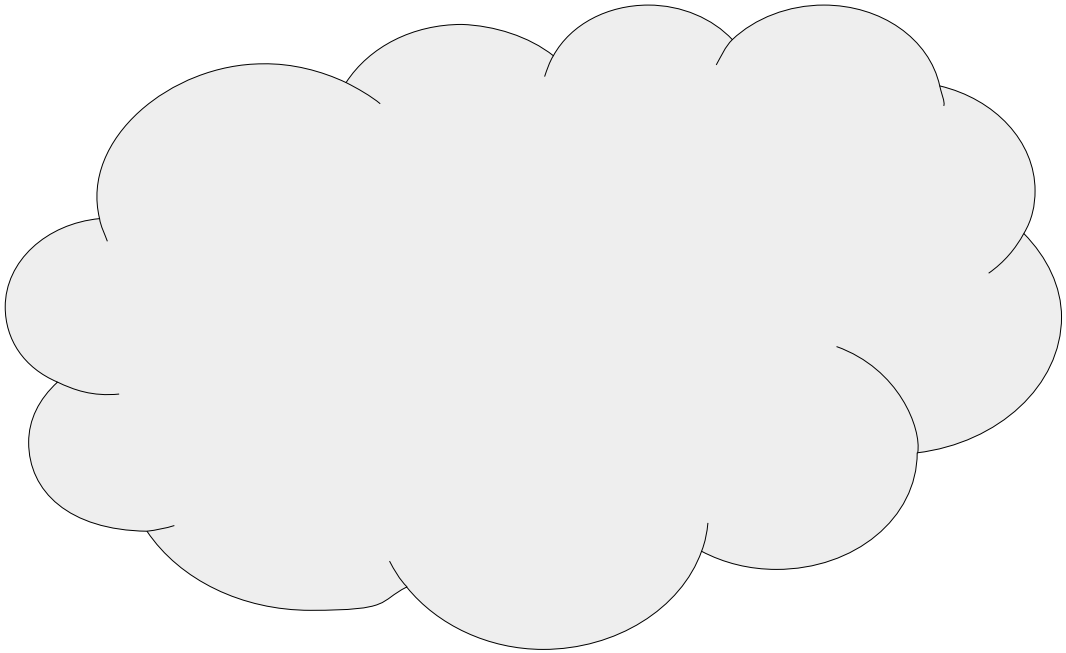
Why Sets?

- A shopping list is a great place to use a `Set`.
- Sets ignore duplicates, so adding an existing item has no effect.
- There's no notion of "the first item on the list," which matches how you use a shopping list.
- Removing items is easy; no indices need to be adjusted.

```
Set<string> toBuy;
```

Why Sets?

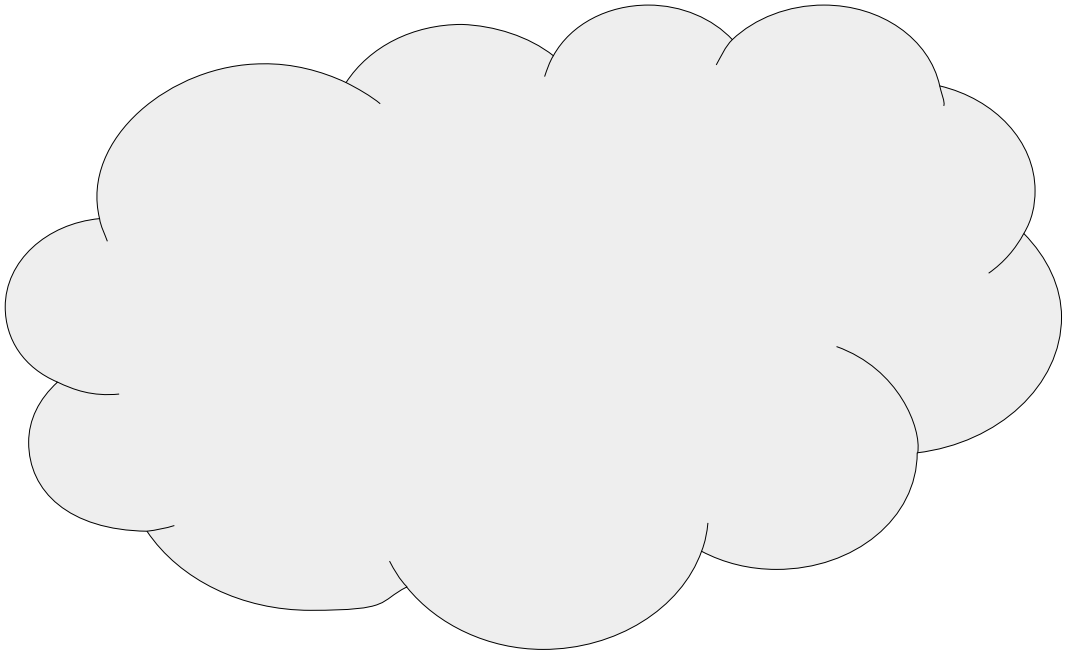
- A shopping list is a great place to use a `Set`.
- Sets ignore duplicates, so adding an existing item has no effect.
- There's no notion of "the first item on the list," which matches how you use a shopping list.
- Removing items is easy; no indices need to be adjusted.



```
Set<string> toBuy;
```

Why Sets?

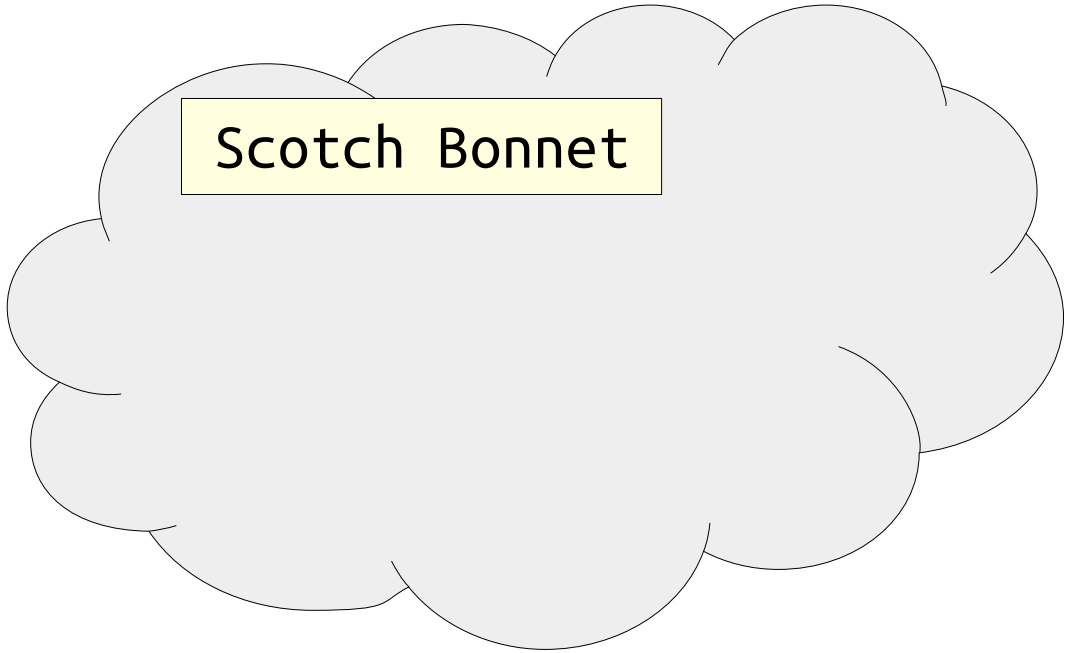
- A shopping list is a great place to use a `Set`.
- Sets ignore duplicates, so adding an existing item has no effect.
- There's no notion of "the first item on the list," which matches how you use a shopping list.
- Removing items is easy; no indices need to be adjusted.



```
Set<string> toBuy;
toBuy += "Scotch bonnet";
```

Why Sets?

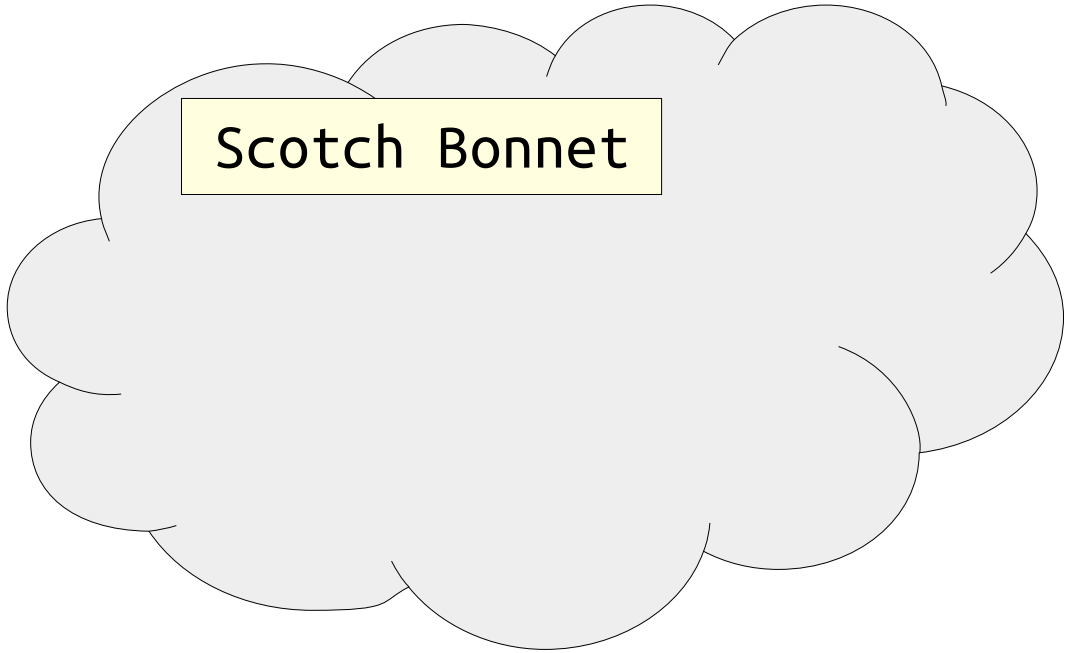
- A shopping list is a great place to use a `Set`.
- Sets ignore duplicates, so adding an existing item has no effect.
- There's no notion of "the first item on the list," which matches how you use a shopping list.
- Removing items is easy; no indices need to be adjusted.




```
Set<string> toBuy;
toBuy += "Scotch bonnet";
```

Why Sets?

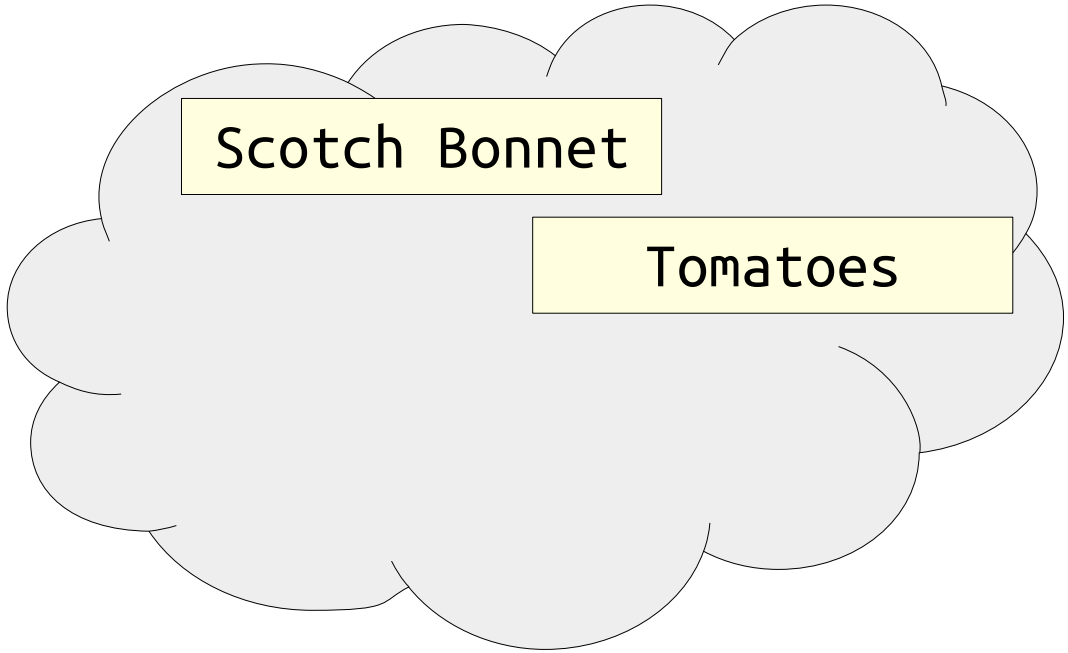
- A shopping list is a great place to use a `Set`.
- Sets ignore duplicates, so adding an existing item has no effect.
- There's no notion of "the first item on the list," which matches how you use a shopping list.
- Removing items is easy; no indices need to be adjusted.




```
Set<string> toBuy;
toBuy += "Scotch bonnet";
toBuy += "Tomatoes";
```

Why Sets?

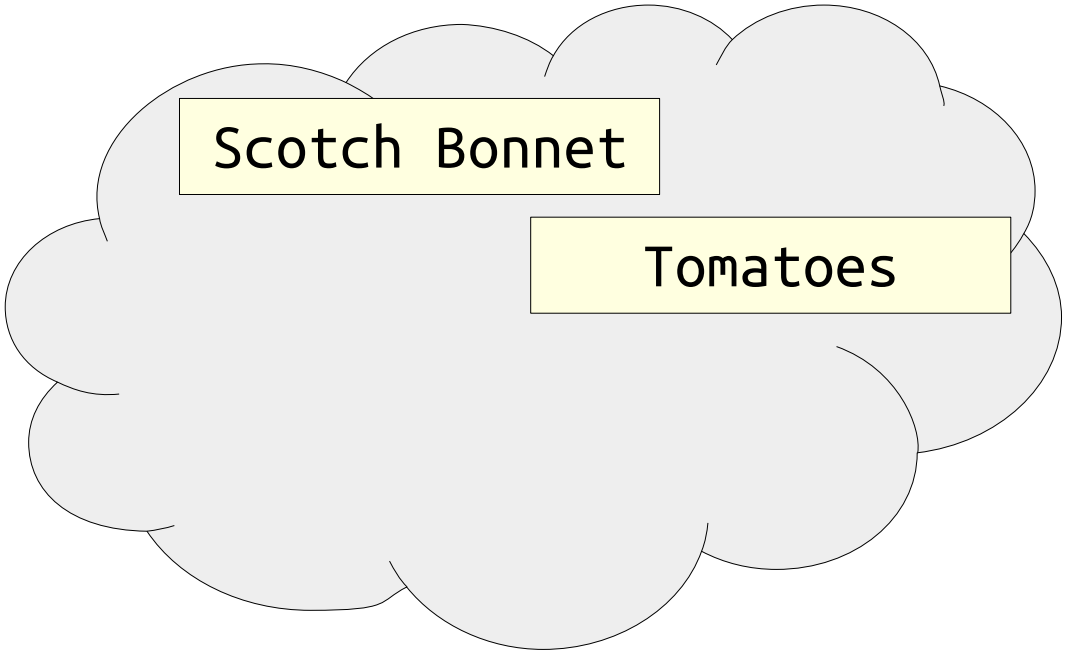
- A shopping list is a great place to use a `Set`.
- Sets ignore duplicates, so adding an existing item has no effect.
- There's no notion of "the first item on the list," which matches how you use a shopping list.
- Removing items is easy; no indices need to be adjusted.




```
Set<string> toBuy;
toBuy += "Scotch bonnet";
toBuy += "Tomatoes";
```

Why Sets?

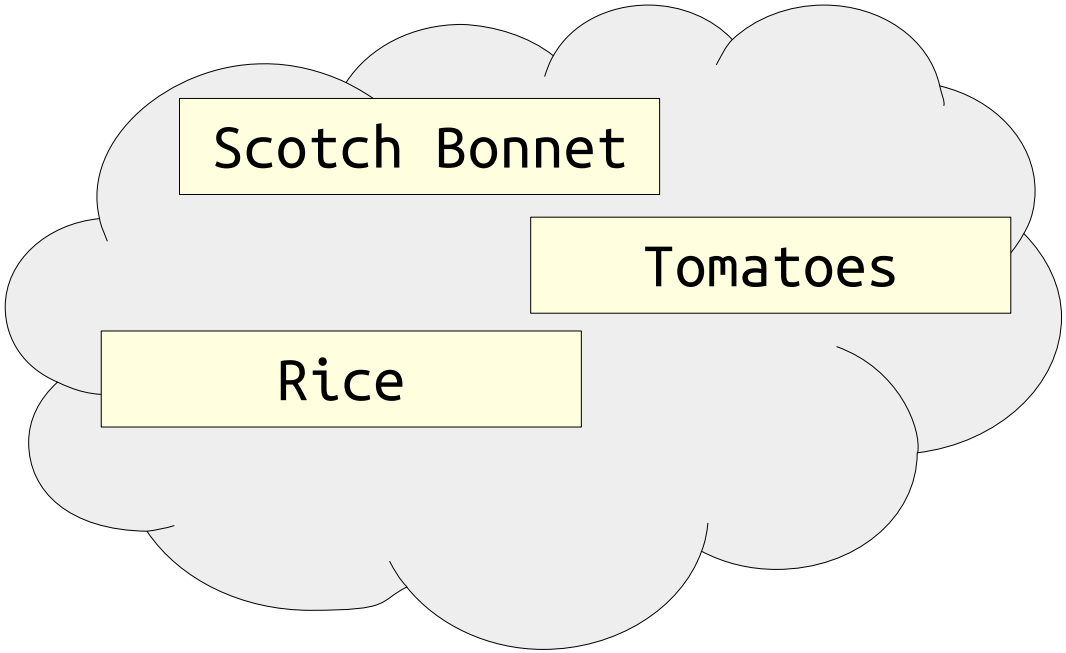
- A shopping list is a great place to use a `Set`.
- Sets ignore duplicates, so adding an existing item has no effect.
- There's no notion of "the first item on the list," which matches how you use a shopping list.
- Removing items is easy; no indices need to be adjusted.




```
Set<string> toBuy;
toBuy += "Scotch bonnet";
toBuy += "Tomatoes";
toBuy += "Rice";
```

Why Sets?

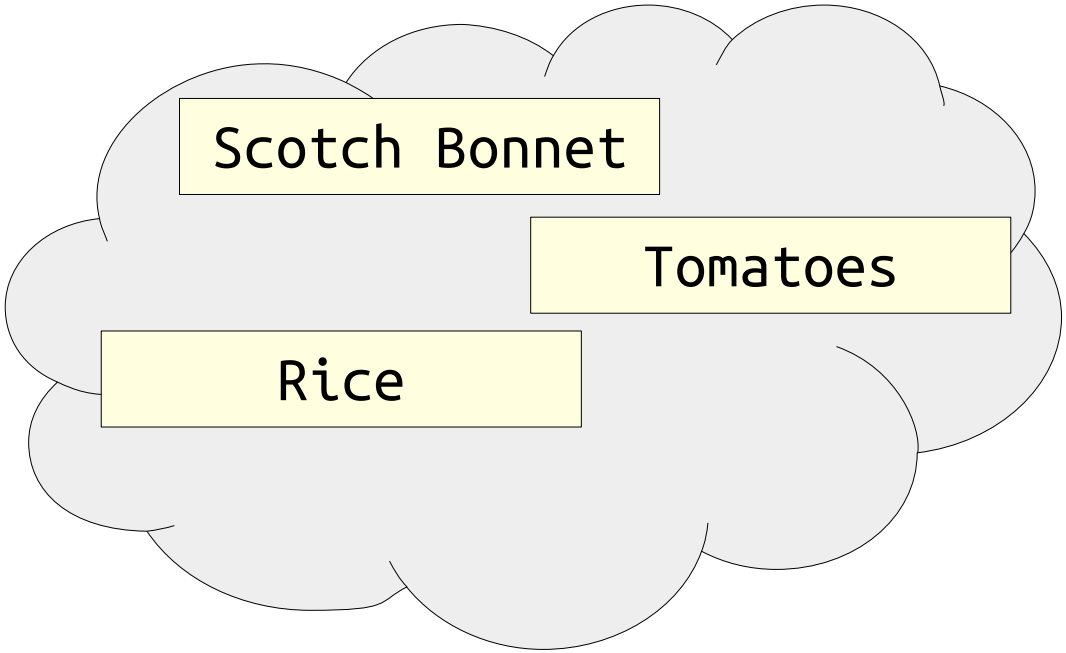
- A shopping list is a great place to use a `Set`.
- Sets ignore duplicates, so adding an existing item has no effect.
- There's no notion of "the first item on the list," which matches how you use a shopping list.
- Removing items is easy; no indices need to be adjusted.




```
Set<string> toBuy;
toBuy += "Scotch bonnet";
toBuy += "Tomatoes";
toBuy += "Rice";
```

Why Sets?

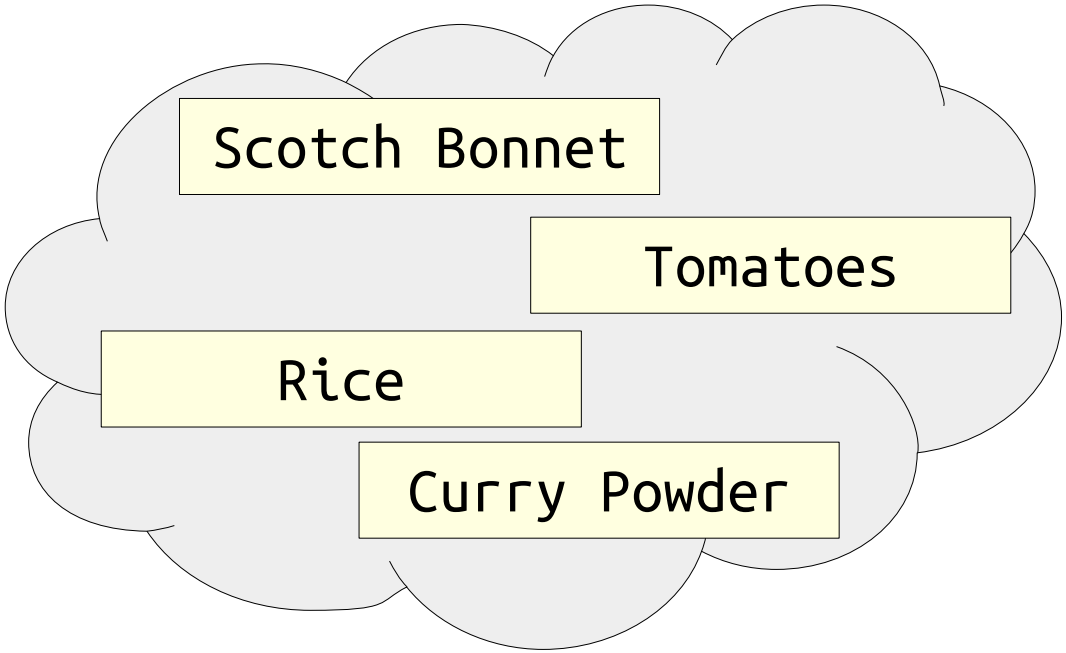
- A shopping list is a great place to use a `Set`.
- Sets ignore duplicates, so adding an existing item has no effect.
- There's no notion of "the first item on the list," which matches how you use a shopping list.
- Removing items is easy; no indices need to be adjusted.




```
Set<string> toBuy;
toBuy += "Scotch bonnet";
toBuy += "Tomatoes";
toBuy += "Rice";
toBuy += "Curry powder";
```

Why Sets?

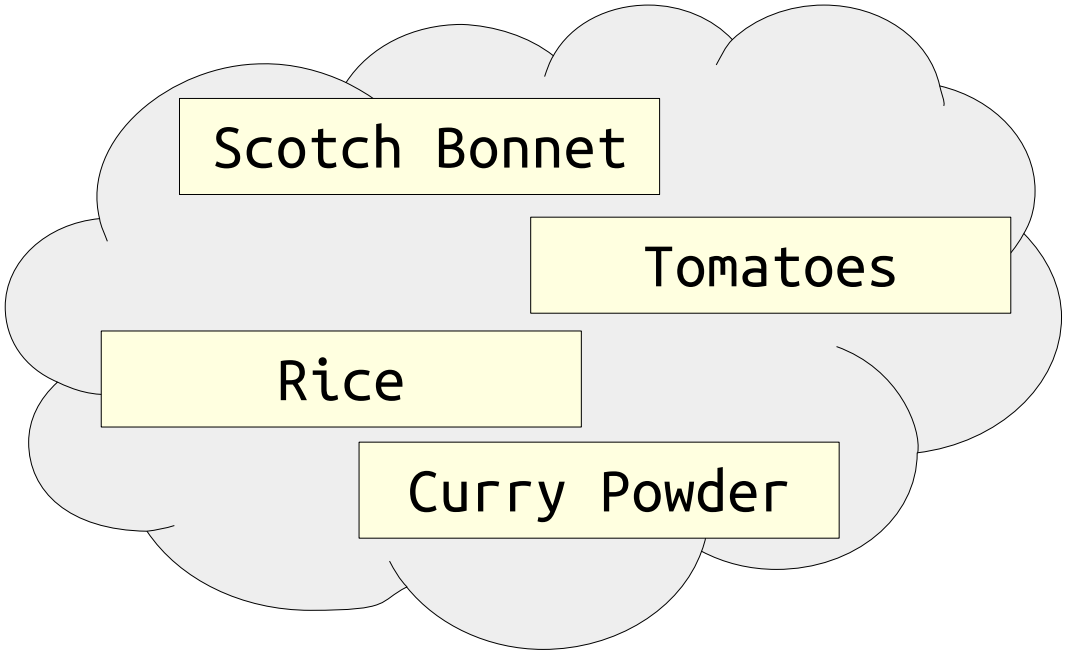
- A shopping list is a great place to use a `Set`.
- Sets ignore duplicates, so adding an existing item has no effect.
- There's no notion of "the first item on the list," which matches how you use a shopping list.
- Removing items is easy; no indices need to be adjusted.




```
Set<string> toBuy;
toBuy += "Scotch bonnet";
toBuy += "Tomatoes";
toBuy += "Rice";
toBuy += "Curry powder";
```

Why Sets?

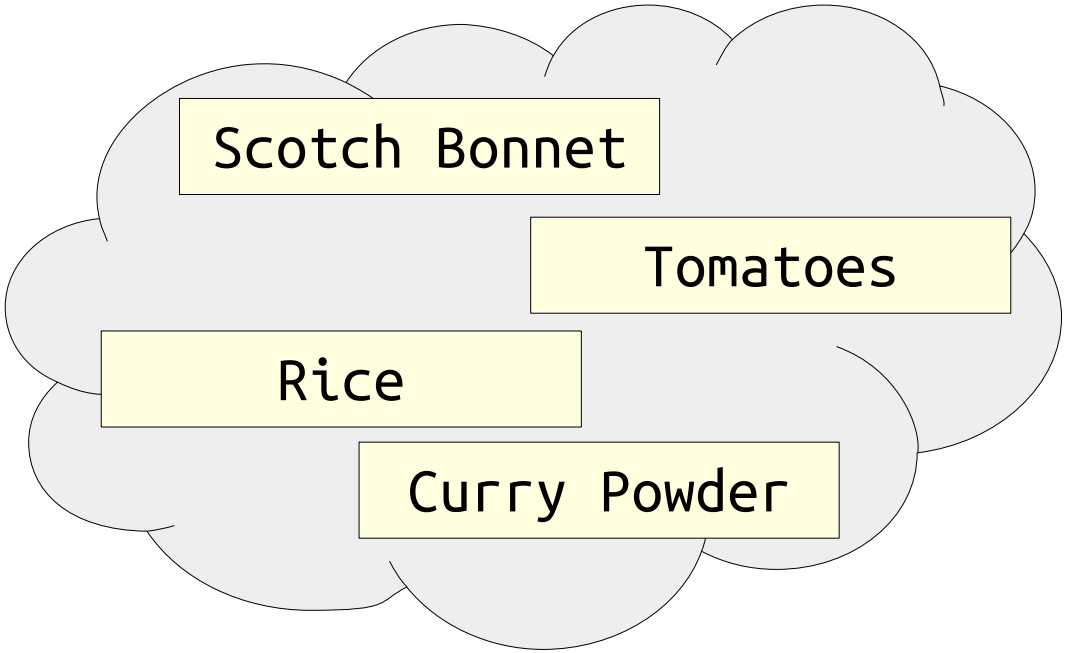
- A shopping list is a great place to use a `Set`.
- Sets ignore duplicates, so adding an existing item has no effect.
- There's no notion of "the first item on the list," which matches how you use a shopping list.
- Removing items is easy; no indices need to be adjusted.




```
Set<string> toBuy;
toBuy += "Scotch bonnet";
toBuy += "Tomatoes";
toBuy += "Rice";
toBuy += "Curry powder";
toBuy += "Rice";
```

Why Sets?

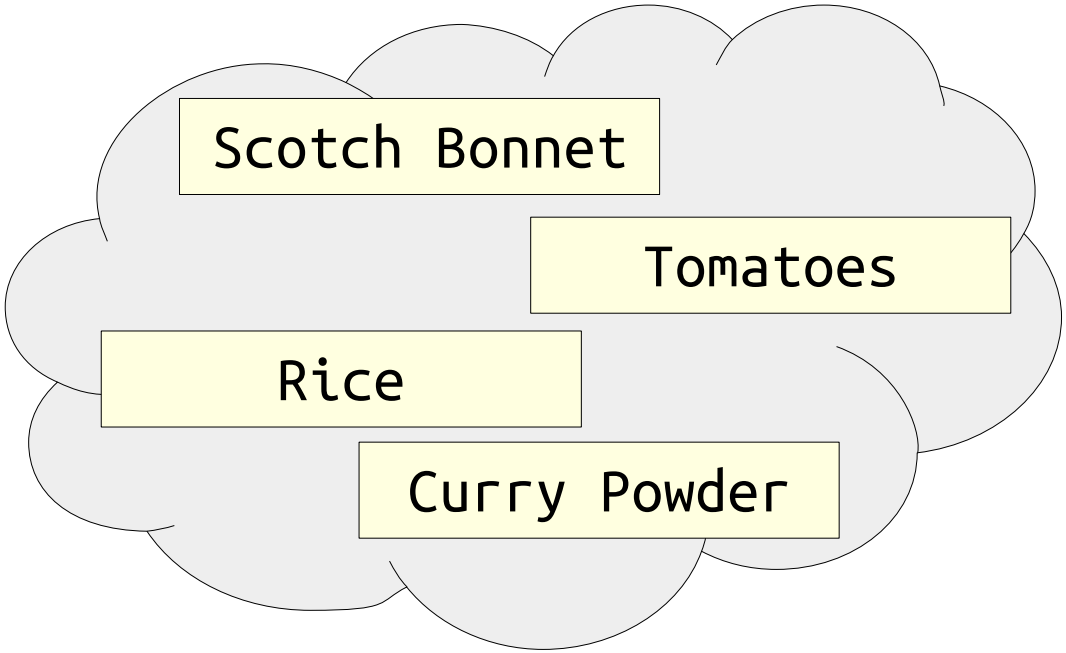
- A shopping list is a great place to use a `Set`.
- Sets ignore duplicates, so adding an existing item has no effect.
- There's no notion of "the first item on the list," which matches how you use a shopping list.
- Removing items is easy; no indices need to be adjusted.




```
Set<string> toBuy;
toBuy += "Scotch bonnet";
toBuy += "Tomatoes";
toBuy += "Rice";
toBuy += "Curry powder";
toBuy += "Rice"; // Okay!
```

Why Sets?

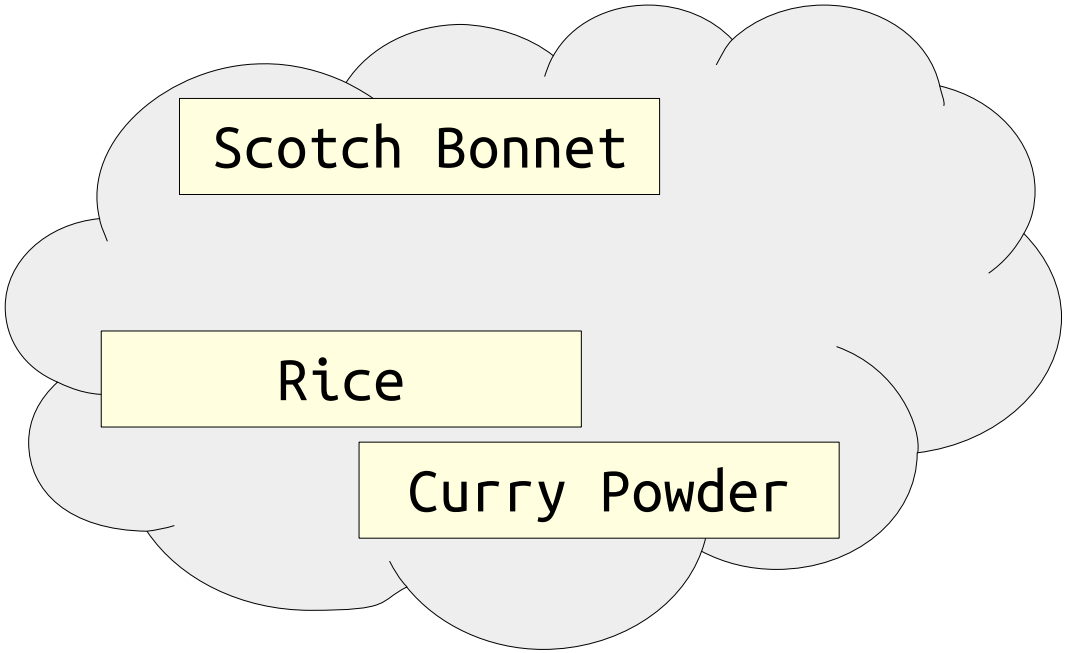
- A shopping list is a great place to use a `Set`.
- Sets ignore duplicates, so adding an existing item has no effect.
- There's no notion of "the first item on the list," which matches how you use a shopping list.
- Removing items is easy; no indices need to be adjusted.




```
Set<string> toBuy;
toBuy += "Scotch bonnet";
toBuy += "Tomatoes";
toBuy += "Rice";
toBuy += "Curry powder";
toBuy += "Rice"; // Okay!

toBuy -= "Tomatoes"; // Clearer!
```

Why Sets?

- A shopping list is a great place to use a `Set`.
- Sets ignore duplicates, so adding an existing item has no effect.
- There's no notion of "the first item on the list," which matches how you use a shopping list.
- Removing items is easy; no indices need to be adjusted.




```
Set<string> toBuy;
toBuy += "Scotch bonnet";
toBuy += "Tomatoes";
toBuy += "Rice";
toBuy += "Curry powder";
toBuy += "Rice"; // Okay!

toBuy -= "Tomatoes"; // Clearer!
```

Operations on `Sets`

- You can add a value to a `Set` by writing

set `+=` ***value***`;`

- You can remove a value from a `Set` by writing

set `-=` ***value***`;`

- You can check if a value exists in a `Set` by writing

set`.contains(`***value***`)`

- Many more operations are available (union, intersection, difference, subset, etc.). Check the Stanford C++ Library Reference guide for details!

Map

Map

- The `Map` class represents a set of key/value pairs.
  - It's analogous to `dict` in Python, to `Map` in Java, and to objects (used as key/value stores) in JavaScript.
- Each key is associated with a value.

Map

- The `Map` class represents a set of key/value pairs.
  - It's analogous to `dict` in Python, to `Map` in Java, and to objects (used as key/value stores) in JavaScript.
- Each key is associated with a value.

```
Map<string, int> heights;
```

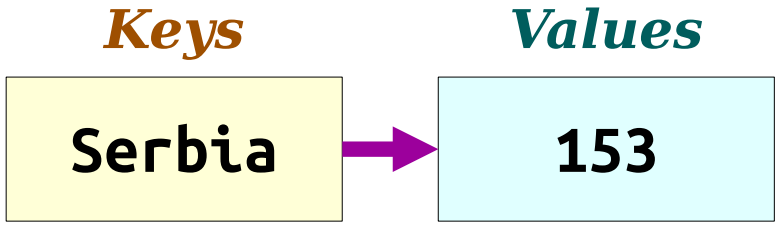
Map

- The `Map` class represents a set of key/value pairs.
  - It's analogous to `dict` in Python, to `Map` in Java, and to objects (used as key/value stores) in JavaScript.
- Each key is associated with a value.

```
Map<string, int> heights;

heights["Serbia"] = 153;
```

Map

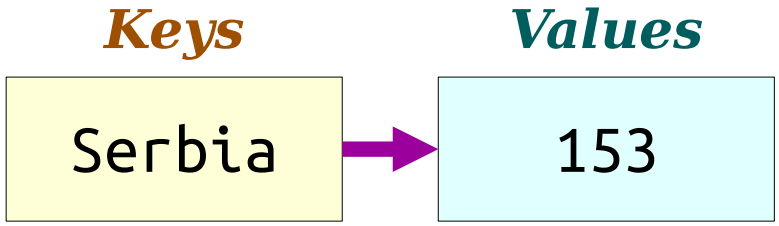
- The `Map` class represents a set of key/value pairs.
  - It's analogous to `dict` in Python, to `Map` in Java, and to objects (used as key/value stores) in JavaScript.
- Each key is associated with a value.




```
Map<string, int> heights;

heights["Serbia"] = 153;
```

Map

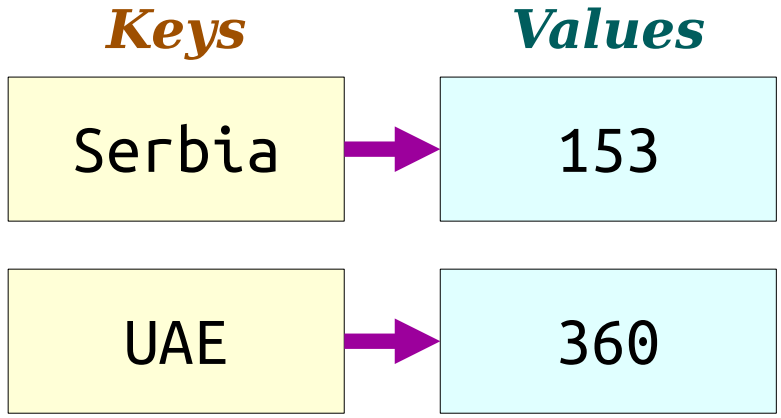
- The `Map` class represents a set of key/value pairs.
  - It's analogous to `dict` in Python, to `Map` in Java, and to objects (used as key/value stores) in JavaScript.
- Each key is associated with a value.




```
Map<string, int> heights;

heights["Serbia"] = 153;
heights["UAE"] = 360;
```

Map

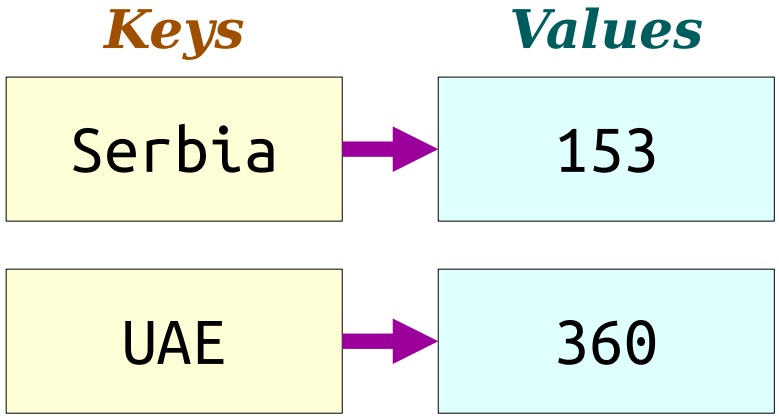
- The `Map` class represents a set of key/value pairs.
  - It's analogous to `dict` in Python, to `Map` in Java, and to objects (used as key/value stores) in JavaScript.
- Each key is associated with a value.




```
Map<string, int> heights;

heights["Serbia"] = 153;
heights["UAE"] = 360;
```

Map

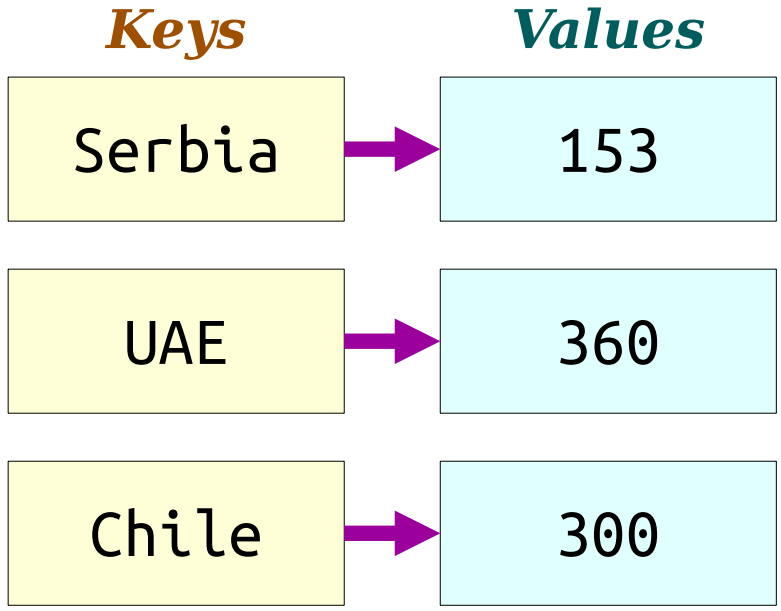
- The `Map` class represents a set of key/value pairs.
  - It's analogous to `dict` in Python, to `Map` in Java, and to objects (used as key/value stores) in JavaScript.
- Each key is associated with a value.




```
Map<string, int> heights;

heights["Serbia"] = 153;
heights["UAE"] = 360;
heights["Chile"] = 300;
```

Map

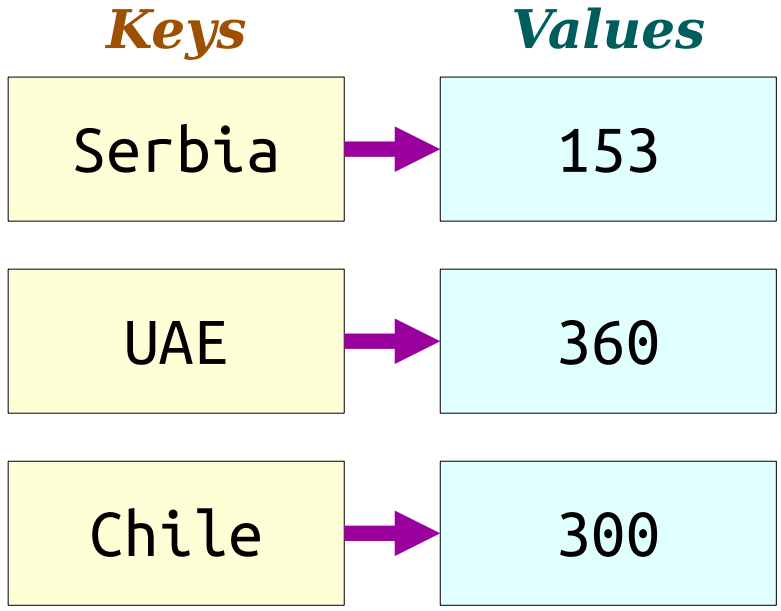
- The `Map` class represents a set of key/value pairs.
  - It's analogous to `dict` in Python, to `Map` in Java, and to objects (used as key/value stores) in JavaScript.
- Each key is associated with a value.




```
Map<string, int> heights;

heights["Serbia"] = 153;
heights["UAE"] = 360;
heights["Chile"] = 300;
```

Map

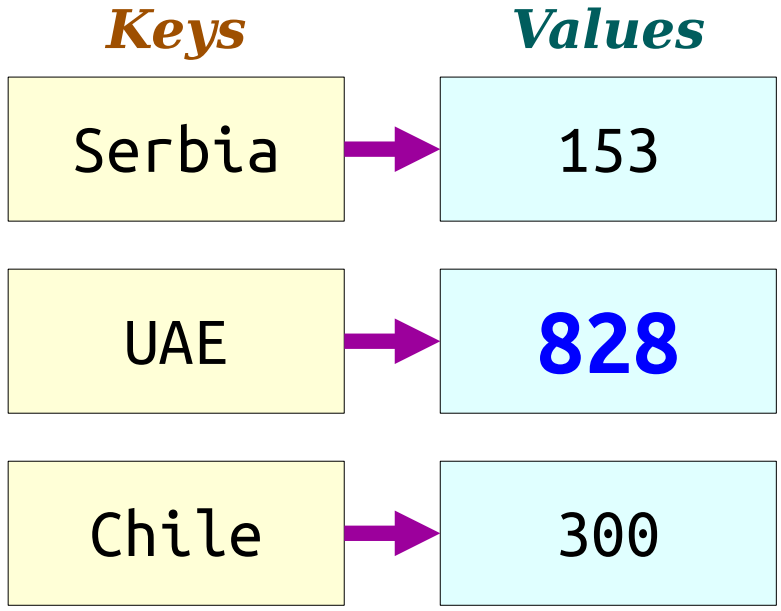
- The `Map` class represents a set of key/value pairs.
  - It's analogous to `dict` in Python, to `Map` in Java, and to objects (used as key/value stores) in JavaScript.
- Each key is associated with a value.




```
Map<string, int> heights;

heights["Serbia"] = 153;
heights["UAE"] = 360;
heights["Chile"] = 300;
heights["UAE"] = 828;
```

Map

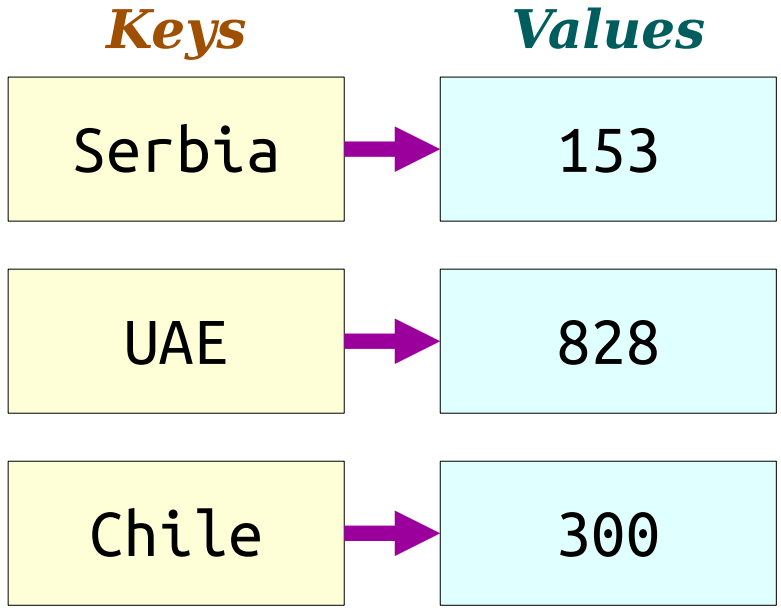
- The `Map` class represents a set of key/value pairs.
  - It's analogous to `dict` in Python, to `Map` in Java, and to objects (used as key/value stores) in JavaScript.
- Each key is associated with a value.




```
Map<string, int> heights;

heights["Serbia"] = 153;
heights["UAE"] = 360;
heights["Chile"] = 300;
heights["UAE"] = 828;
```

Map

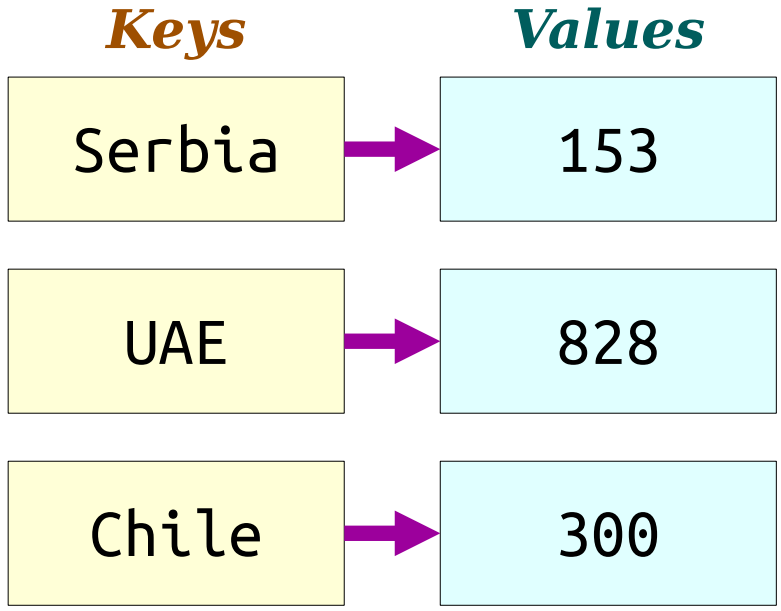
- The `Map` class represents a set of key/value pairs.
  - It's analogous to `dict` in Python, to `Map` in Java, and to objects (used as key/value stores) in JavaScript.
- Each key is associated with a value.




```
Map<string, int> heights;

heights["Serbia"] = 153;
heights["UAE"] = 360;
heights["Chile"] = 300;
heights["UAE"] = 828;
```

Map

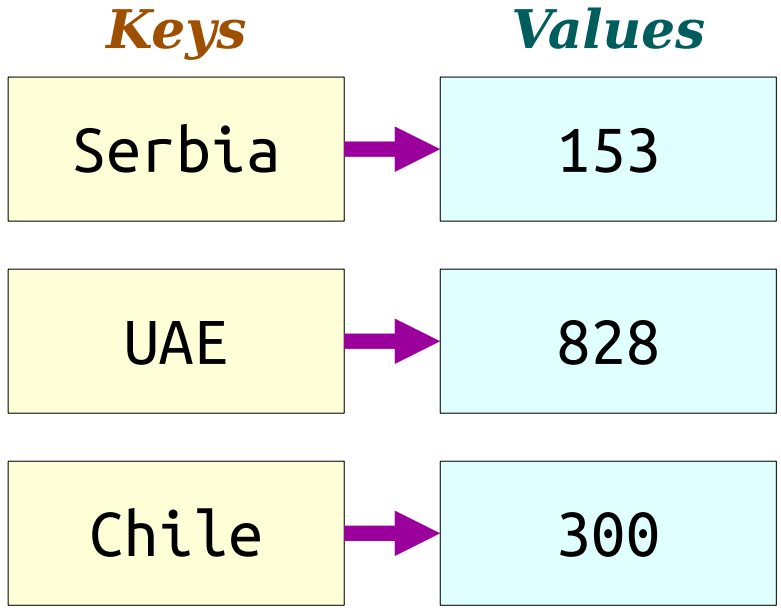
- The `Map` class represents a set of key/value pairs.
  - It's analogous to `dict` in Python, to `Map` in Java, and to objects (used as key/value stores) in JavaScript.
- Each key is associated with a value.
- Given a key, we can look up the associated value.




```
Map<string, int> heights;

heights["Serbia"] = 153;
heights["UAE"] = 360;
heights["Chile"] = 300;
heights["UAE"] = 828;
```

Map

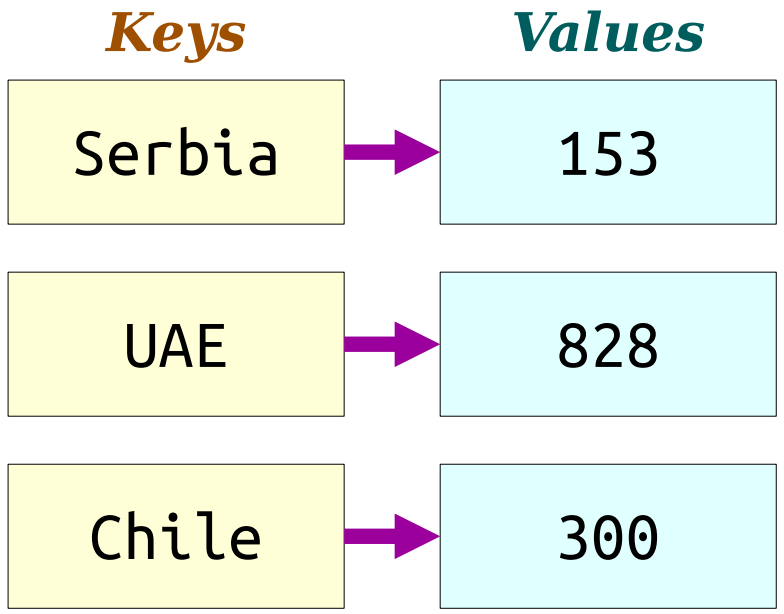
- The `Map` class represents a set of key/value pairs.
  - It's analogous to `dict` in Python, to `Map` in Java, and to objects (used as key/value stores) in JavaScript.
- Each key is associated with a value.
- Given a key, we can look up the associated value.




```
Map<string, int> heights;

heights["Serbia"] = 153;
heights["UAE"] = 360;
heights["Chile"] = 300;
heights["UAE"] = 828;

cout << heights["Chile"] << endl;
```

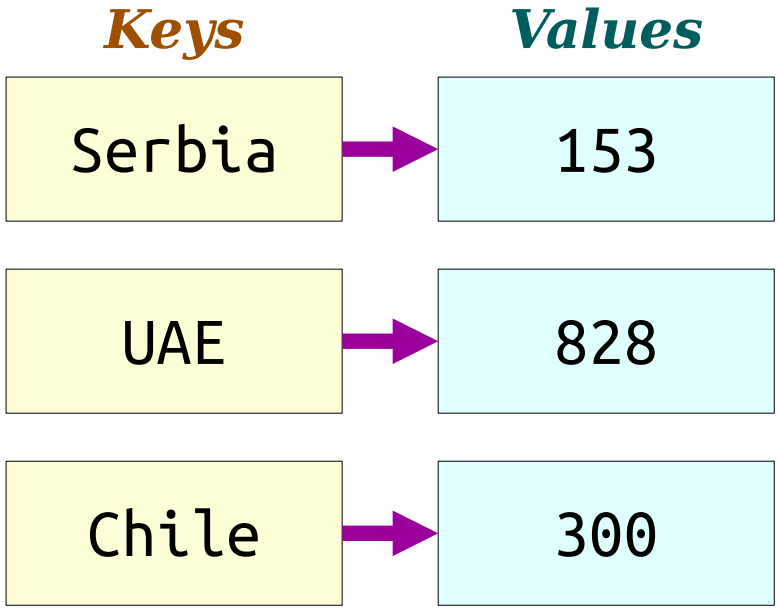
Map

- We can loop over the keys in a map with a range-based for loop.



Map

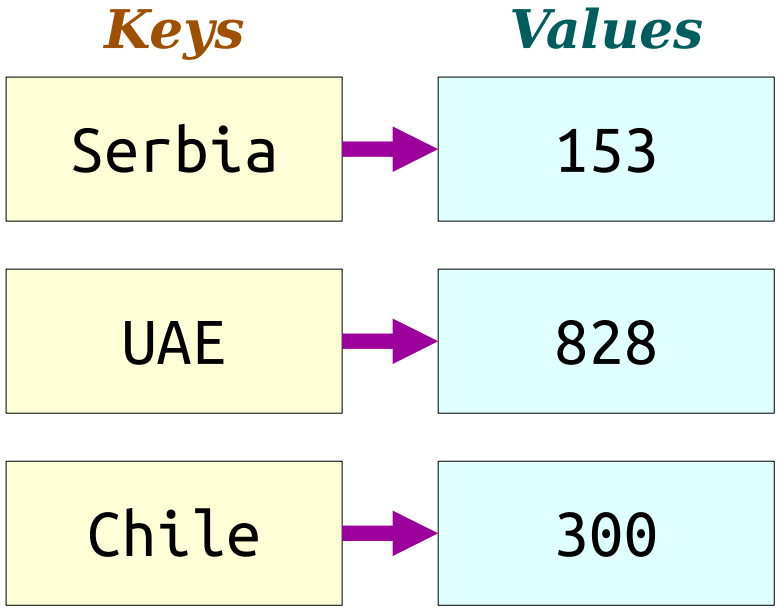
- We can loop over the keys in a map with a range-based for loop.




```
for (string key: heights) {
    cout << heights[key] << endl;
}
```

Map

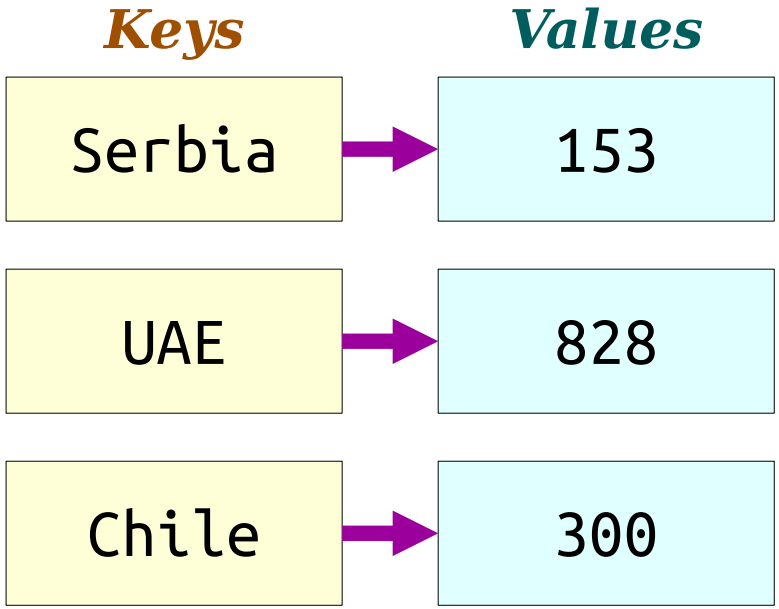
- We can loop over the keys in a map with a range-based for loop.
- We can check whether a key is present in the map.




```
for (string key: heights) {
    cout << heights[key] << endl;
}
```

Map

- We can loop over the keys in a map with a range-based for loop.
- We can check whether a key is present in the map.




```
for (string key: heights) {
    cout << heights[key] << endl;
}

if (heights.containsKey("Mali") {
    cout << "BCEAO" << endl;
}
```

`Map` Autoinsertion

Map Autoinsertion

```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```

Map Autoinsertion

```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```

Map Autoinsertion

```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```

Map Autoinsertion

```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```

Map Autoinsertion

```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```

Map Autoinsertion

```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```

Map Autoinsertion

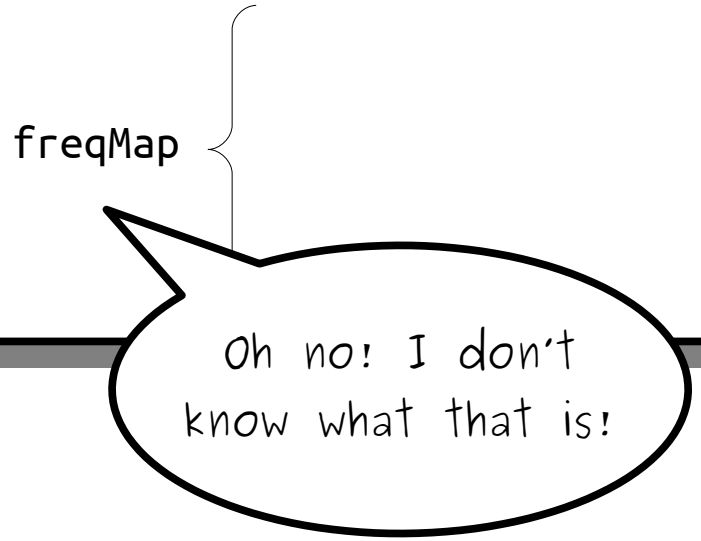
```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```

Map Autoinsertion

```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```

Map Autoinsertion

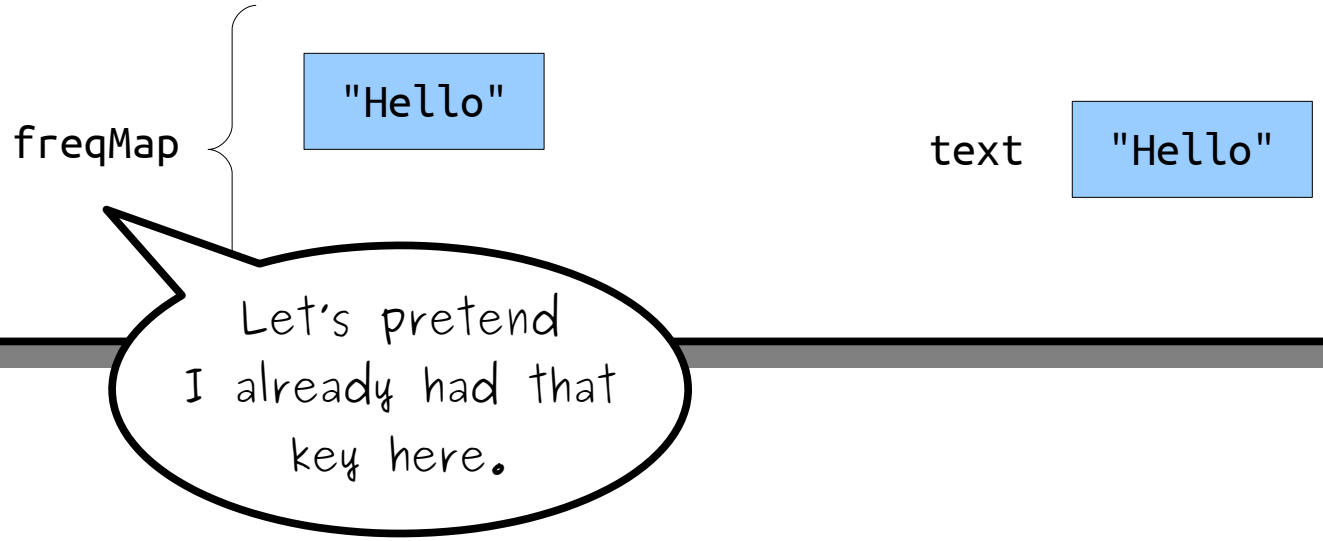
```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```



text "Hello"

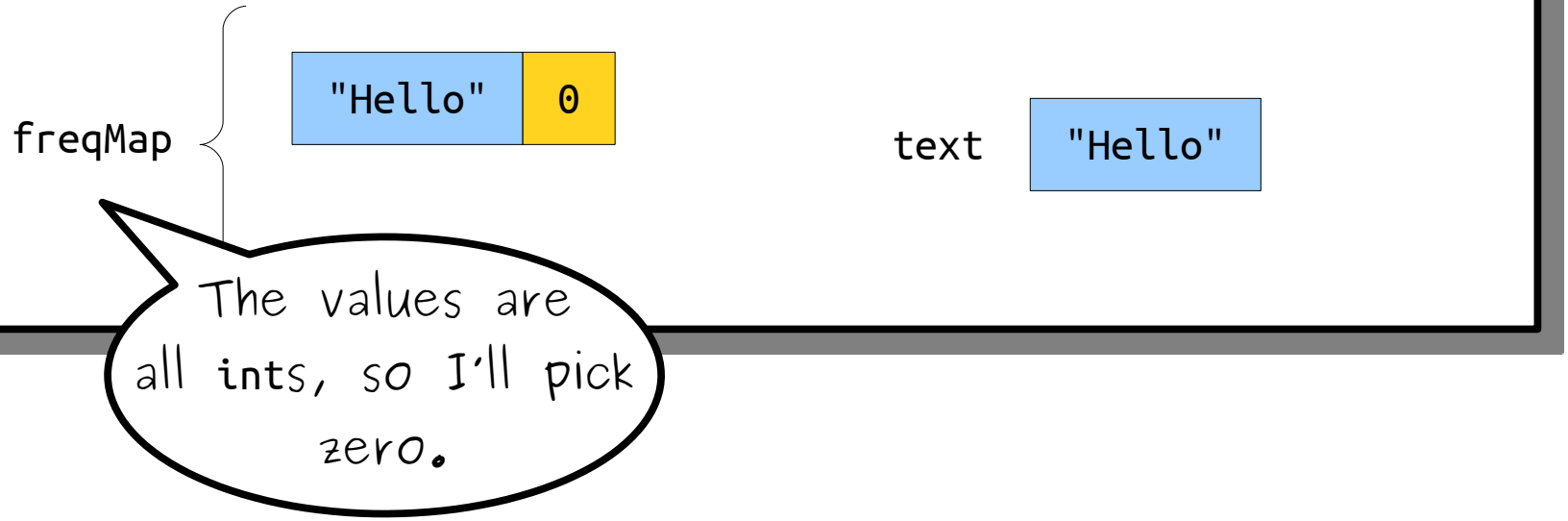
Map Autoinsertion

```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```



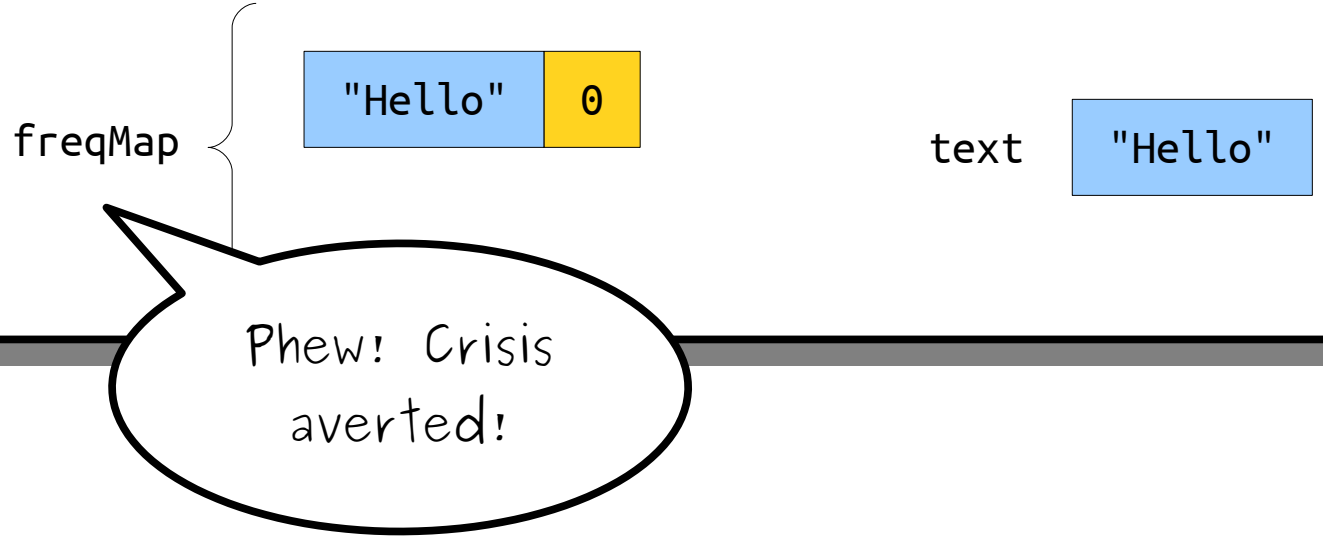
Map Autoinsertion

```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```



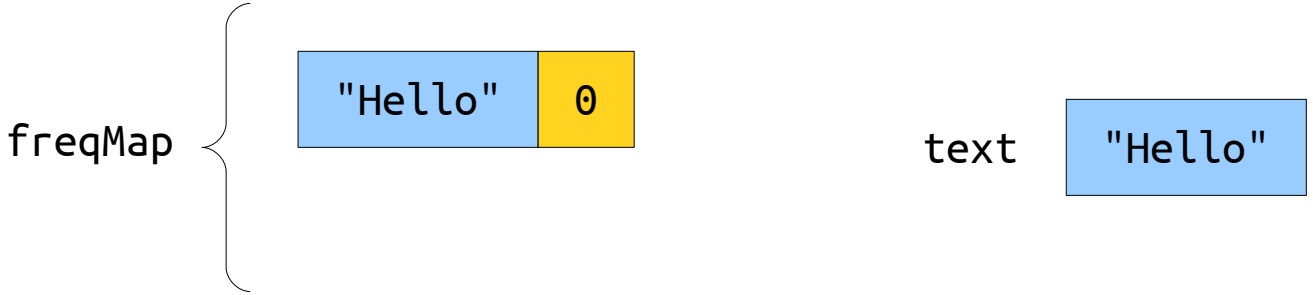
Map Autoinsertion

```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```



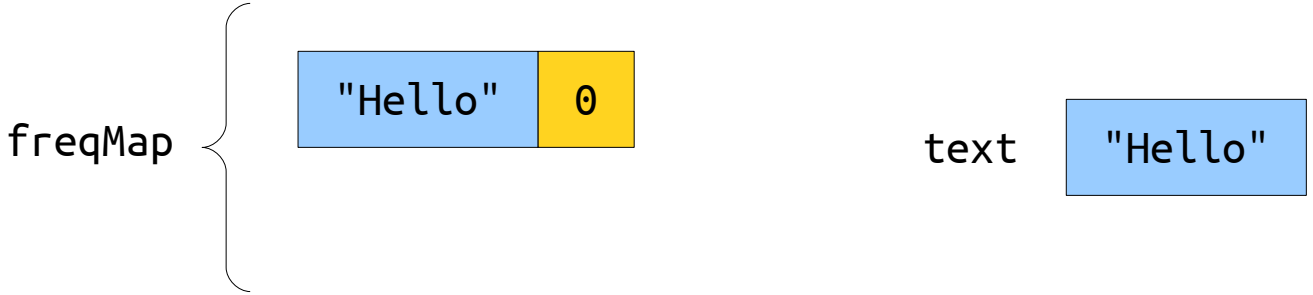
Map Autoinsertion

```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```



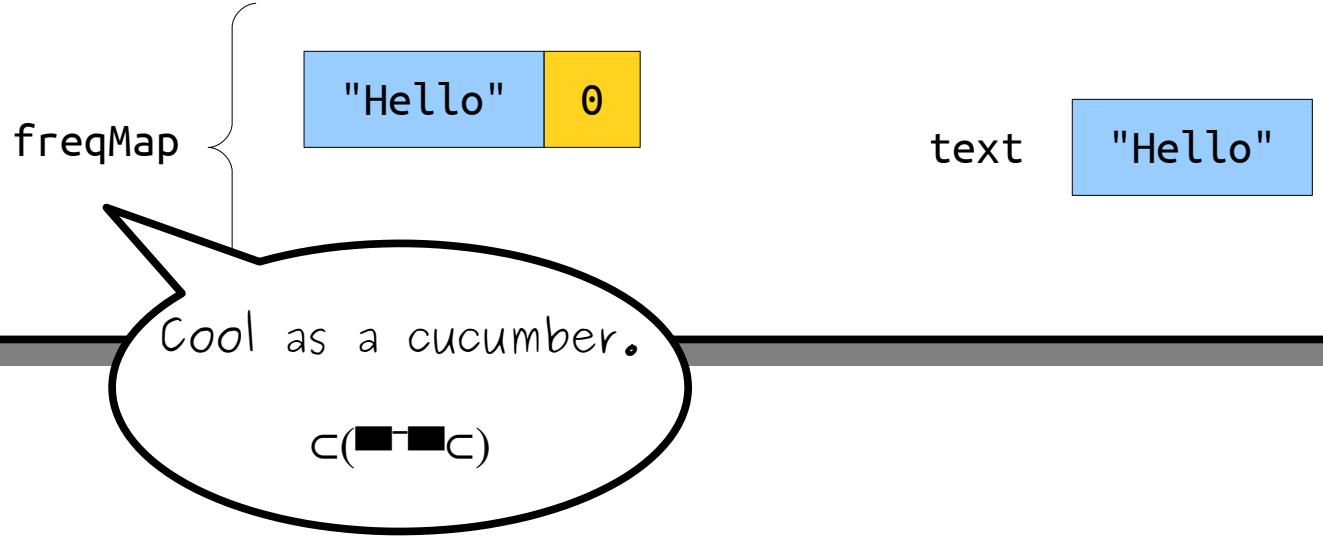
Map Autoinsertion

```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```



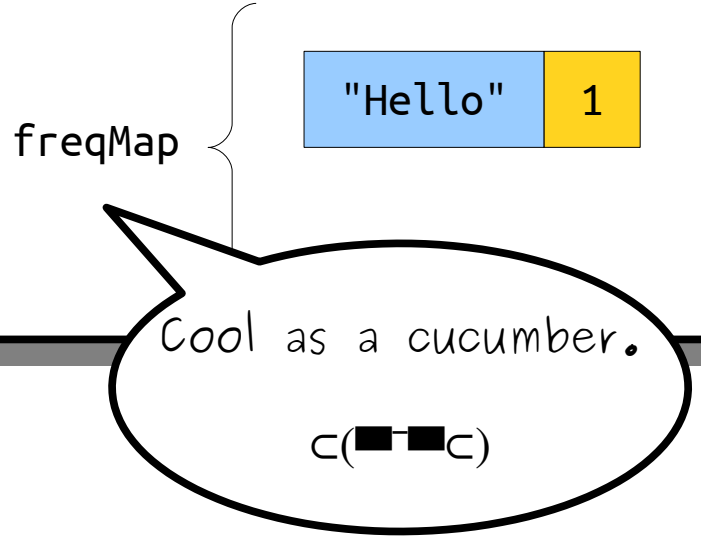
Map Autoinsertion

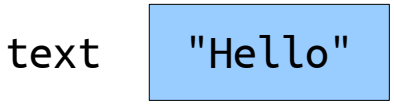
```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```



Map Autoinsertion

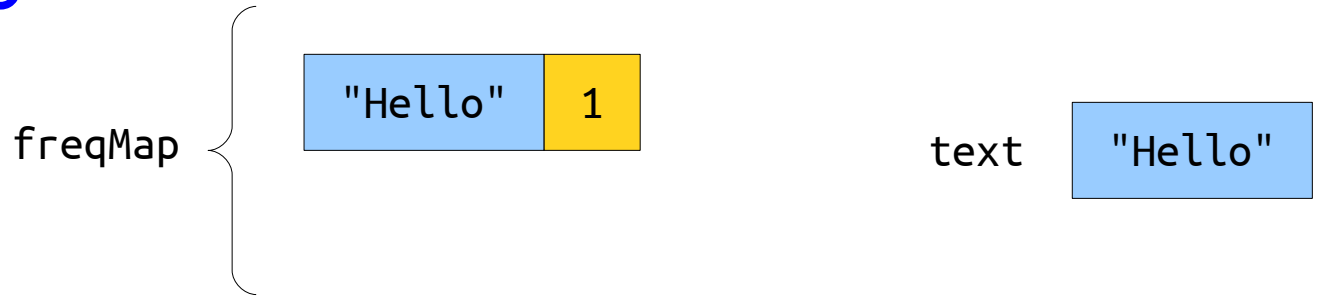
```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```





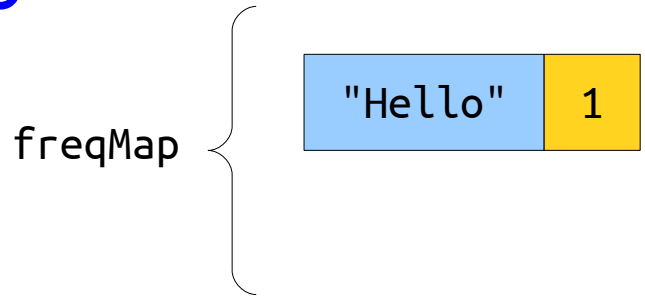
Map Autoinsertion

```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```



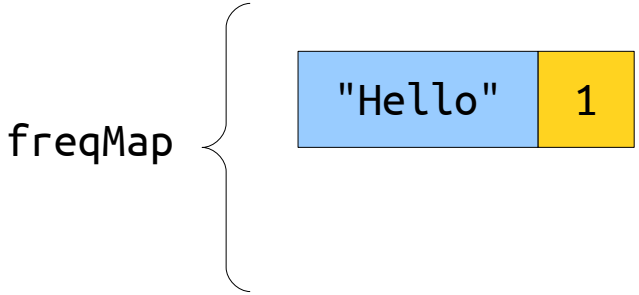
Map Autoinsertion

```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```



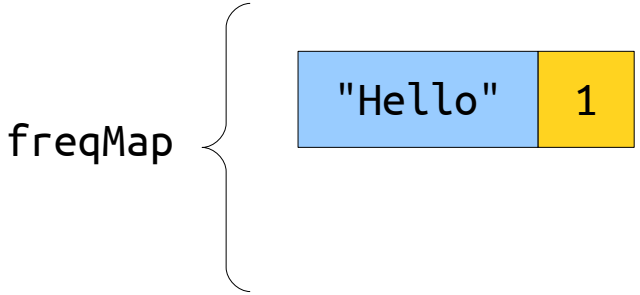
Map Autoinsertion

```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```



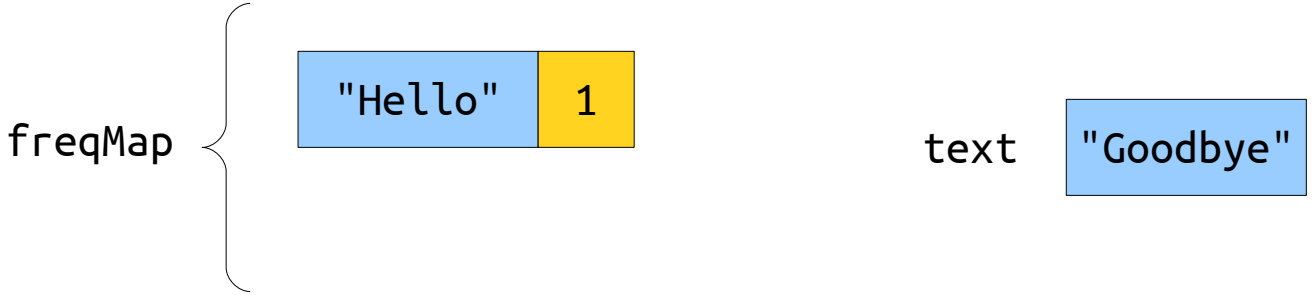
Map Autoinsertion

```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```



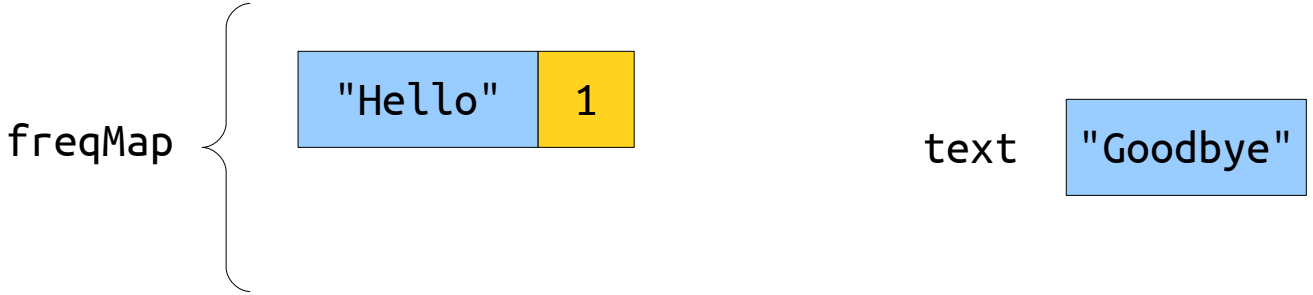
Map Autoinsertion

```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```



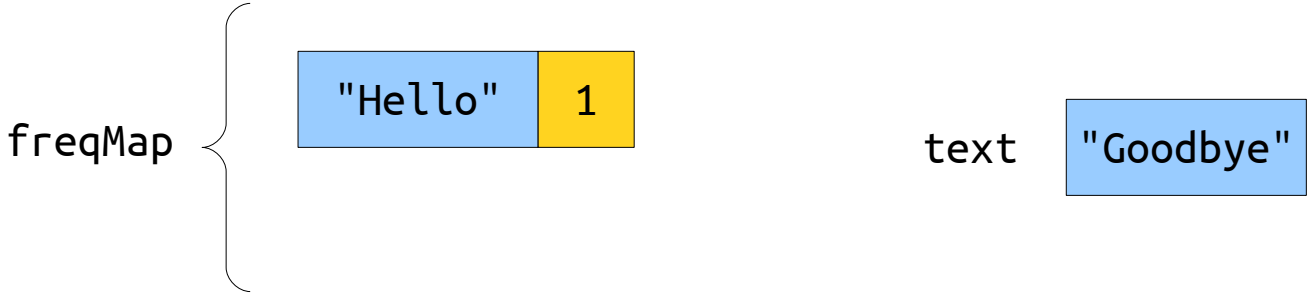
Map Autoinsertion

```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```



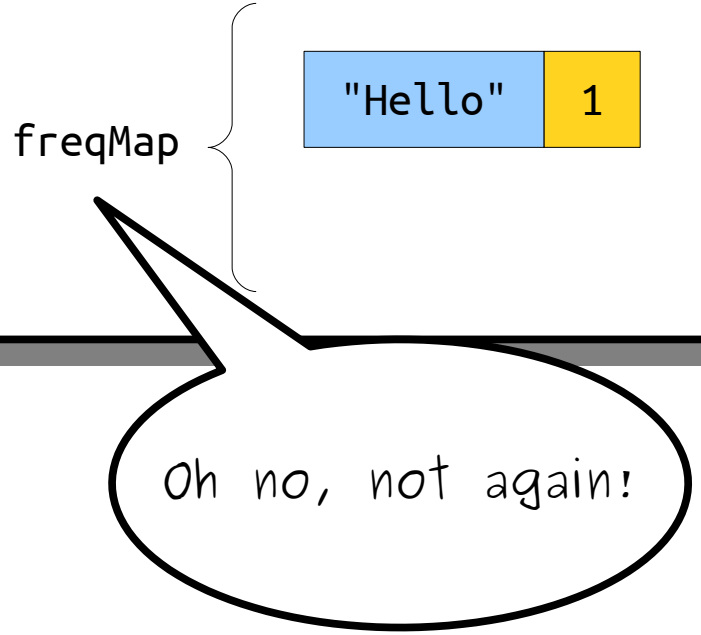
Map Autoinsertion

```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```



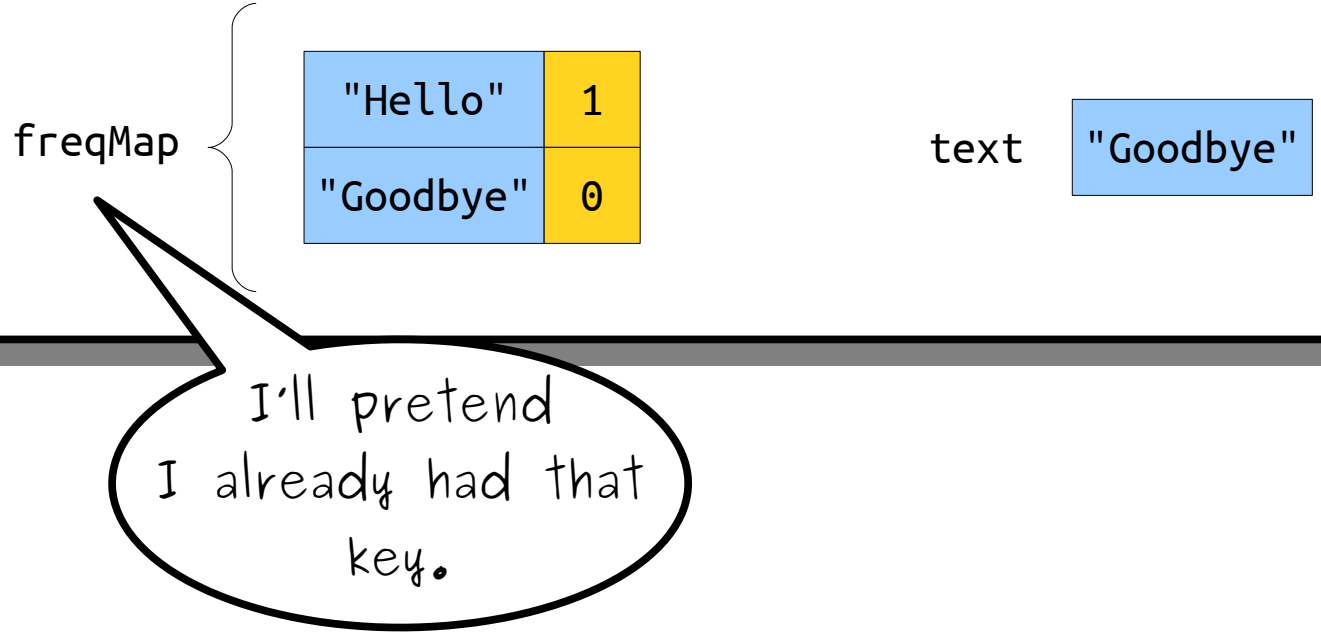
Map Autoinsertion

```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```



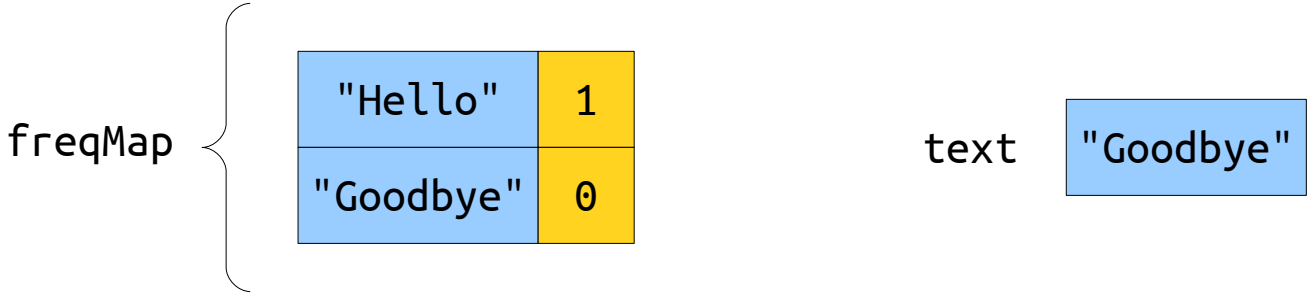
Map Autoinsertion

```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```



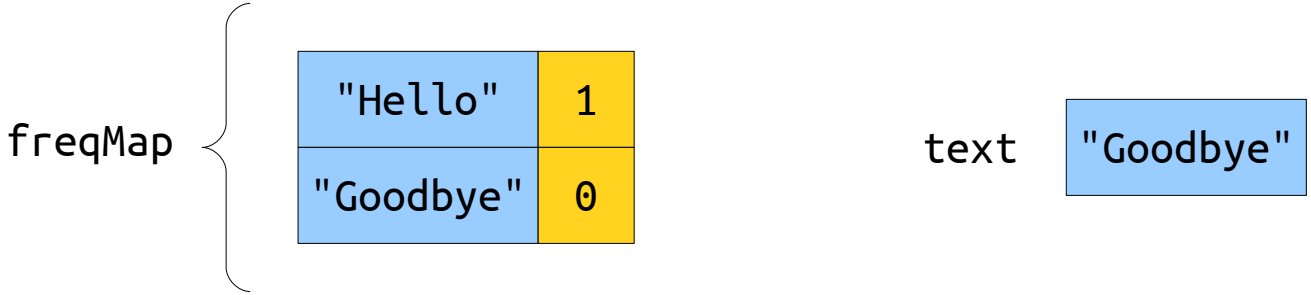
Map Autoinsertion

```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```



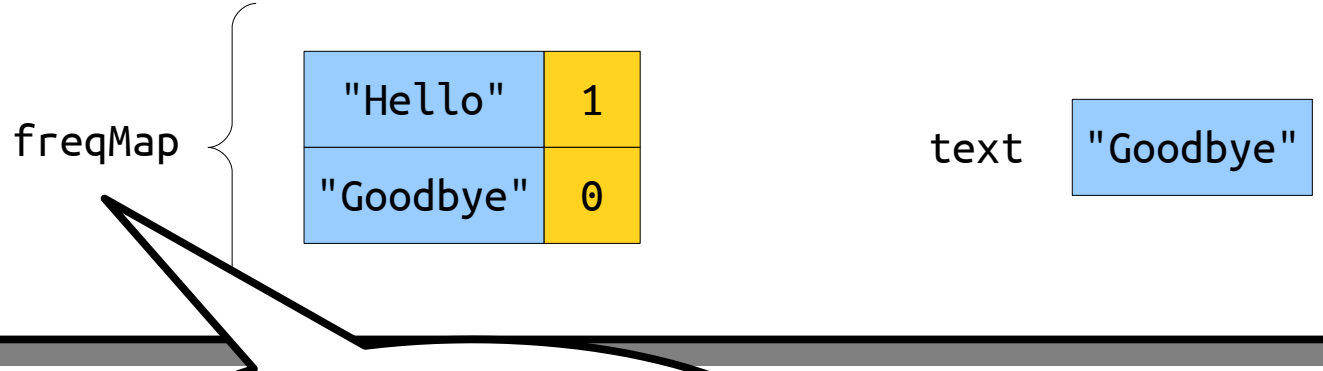
Map Autoinsertion

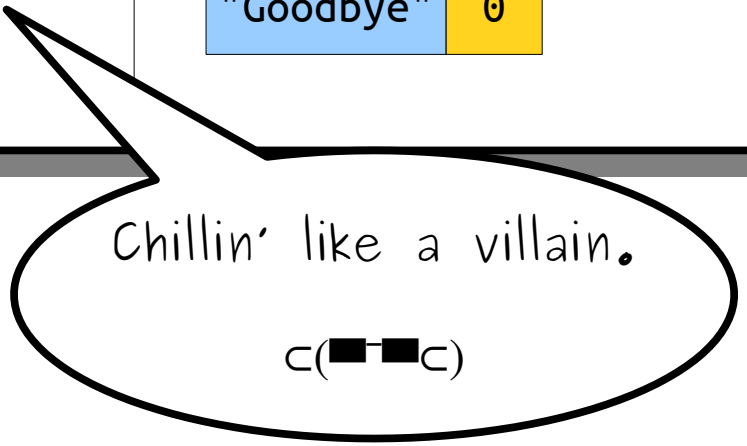
```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```



Map Autoinsertion

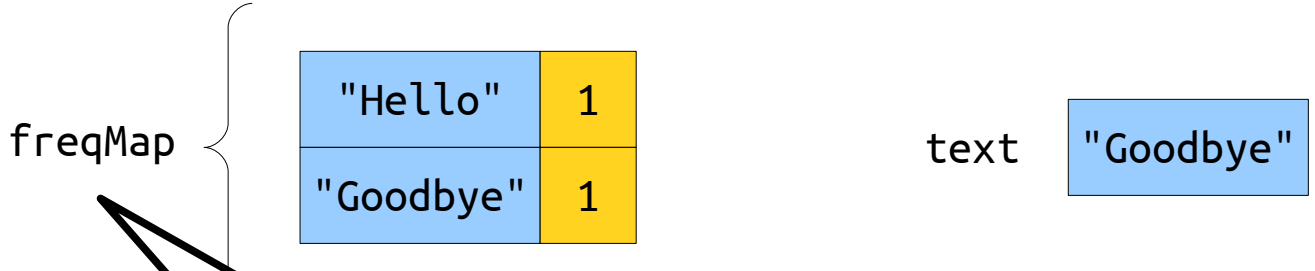
```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```

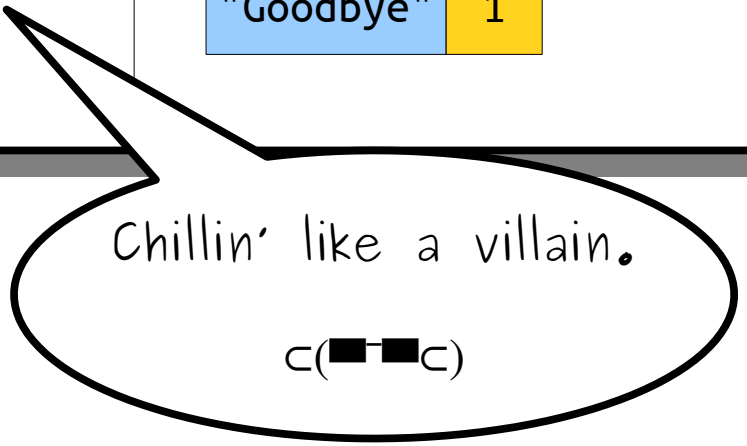




Map Autoinsertion

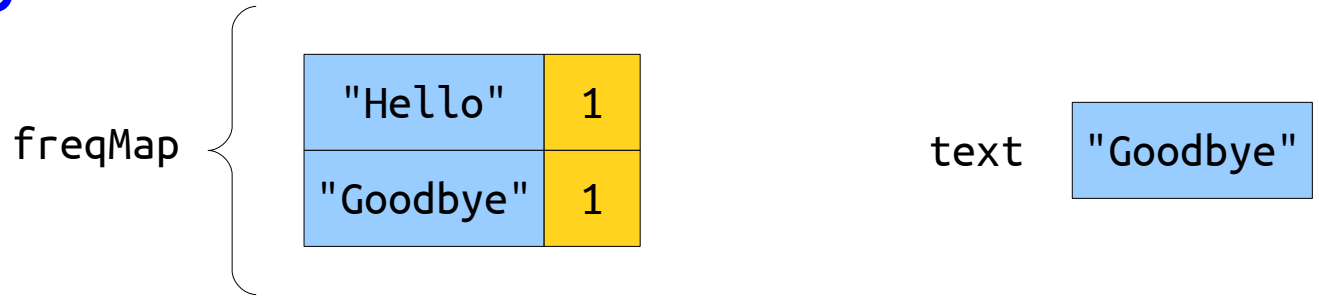
```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```





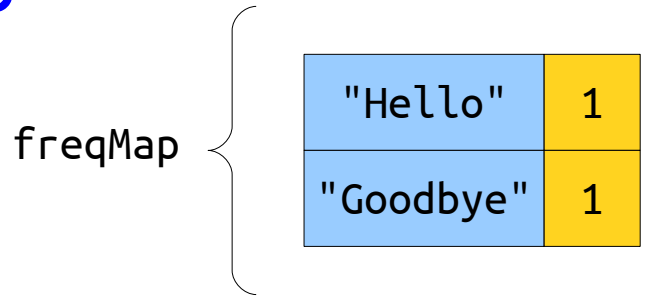
Map Autoinsertion

```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```



Map Autoinsertion

```
Map<string, int> freqMap;
while (true) {
    string text = getLine("Enter some text: ");
    cout << "Times seen: " << freqMap[text] << endl;
    freqMap[text]++;
}
```



Map Autoinsertion

- If you look up something in a `Map` using square brackets,
  - if the key already exists, its associated value is returned; and
  - if the key doesn't exist, it's added in with a "sensible default" value, and that value is then returned.
- This can take some getting used to, but it's surprisingly convenient.

Type	***Default***
`int`	`0`
`double`	`0.0`
`bool`	`false`
`string`	`""`
Any Container	Empty container of that type
`char`	*`(it's complicated)`*

Sorting by First Letters

Map Autoinsertion

```
Lexicon english("EnglishWords.txt");

Map<char, Lexicon> wordsByFirstLetter;
for (string word: english) {
   wordsByFirstLetter[word[0]] += word;
}
```

Map Autoinsertion

```
Lexicon english("EnglishWords.txt");

Map<char, Lexicon> wordsByFirstLetter;
for (string word: english) {
   wordsByFirstLetter[word[0]] += word;
}
```

Map Autoinsertion

```
Lexicon english("EnglishWords.txt");

Map<char, Lexicon> wordsByFirstLetter;
for (string word: english) {
   wordsByFirstLetter[word[0]] += word;
}
```

wordsByFirstLetter

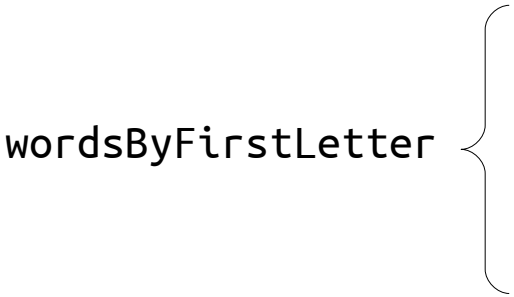
Map Autoinsertion

```
Lexicon english("EnglishWords.txt");

Map<char, Lexicon> wordsByFirstLetter;
for (string word: english) {
    wordsByFirstLetter[word[0]] += word;
}
```

wordsByFirstLetter

Map Autoinsertion

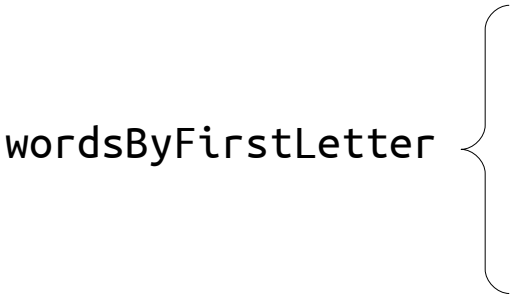
```
Lexicon english("EnglishWords.txt");

Map<char, Lexicon> wordsByFirstLetter;
for (string word: english) {
    wordsByFirstLetter[word[0]] += word;
}
```

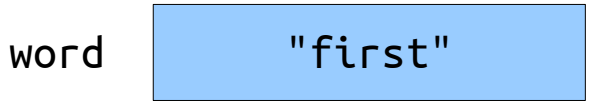


word "first"

Map Autoinsertion

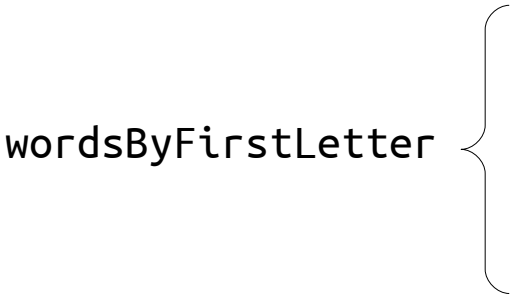
```
Lexicon english("EnglishWords.txt");

Map<char, Lexicon> wordsByFirstLetter;
for (string word: english) {
   wordsByFirstLetter[word[0]] += word;
}
```

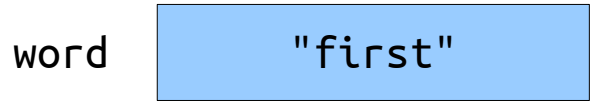




Map Autoinsertion

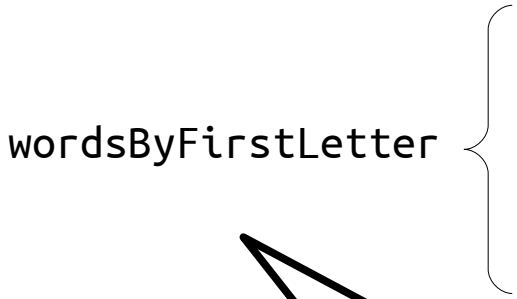
```
Lexicon english("EnglishWords.txt");

Map<char, Lexicon> wordsByFirstLetter;
for (string word: english) {
   wordsByFirstLetter[word[0]] += word;
}
```

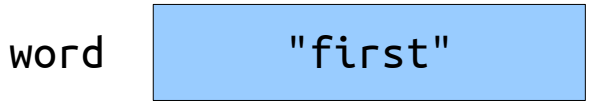




Map Autoinsertion

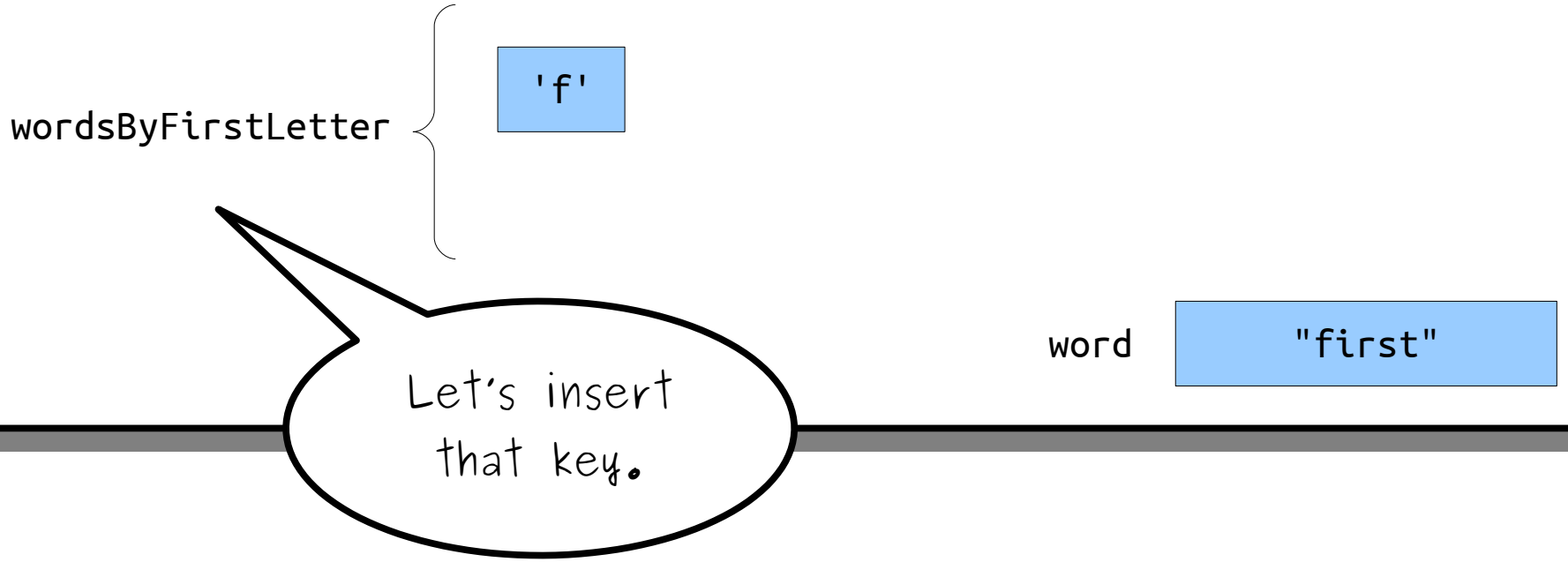
```
Lexicon english("EnglishWords.txt");

Map<char, Lexicon> wordsByFirstLetter;
for (string word: english) {
   wordsByFirstLetter[word[0]] += word;
}
```





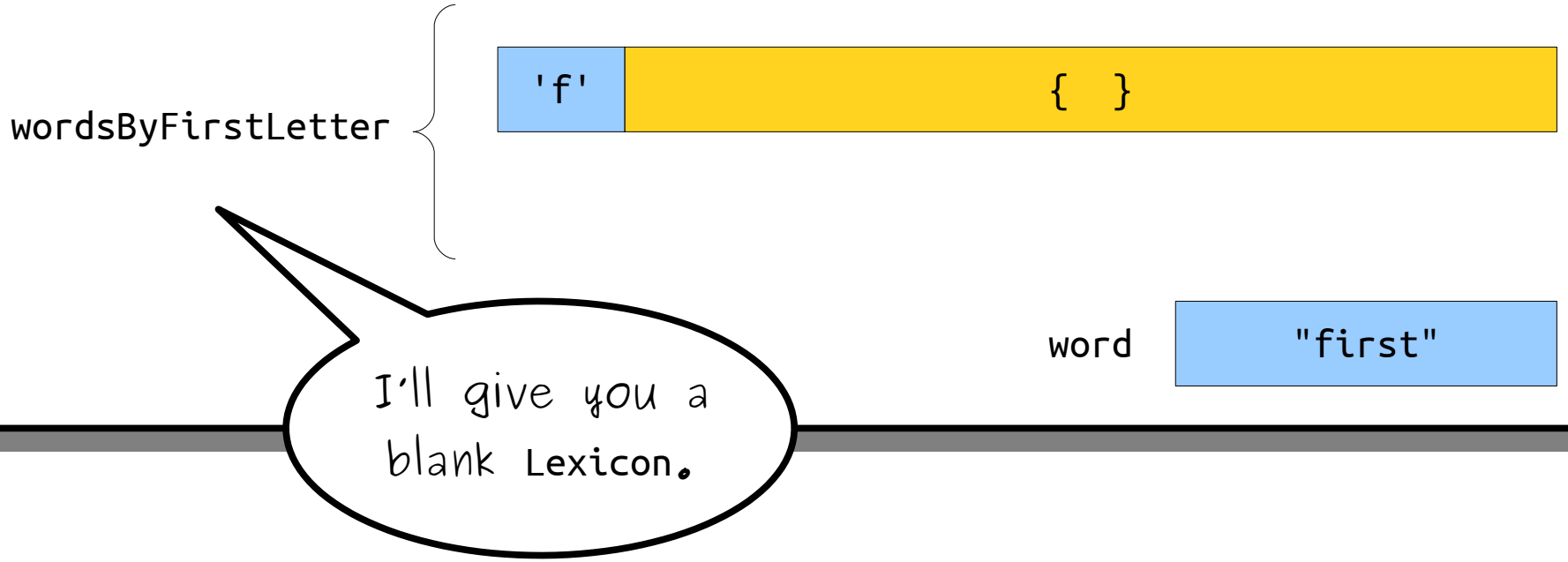
Map Autoinsertion

```
Lexicon english("EnglishWords.txt");

Map<char, Lexicon> wordsByFirstLetter;
for (string word: english) {
   wordsByFirstLetter[word[0]] += word;
}
```



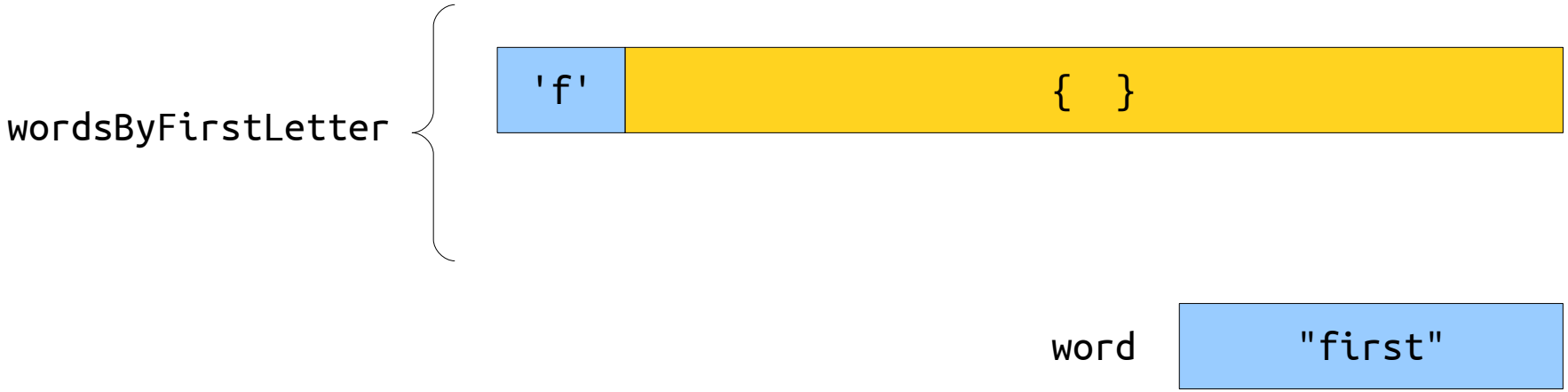
Map Autoinsertion

```
Lexicon english("EnglishWords.txt");

Map<char, Lexicon> wordsByFirstLetter;
for (string word: english) {
   wordsByFirstLetter[word[0]] += word;
}
```



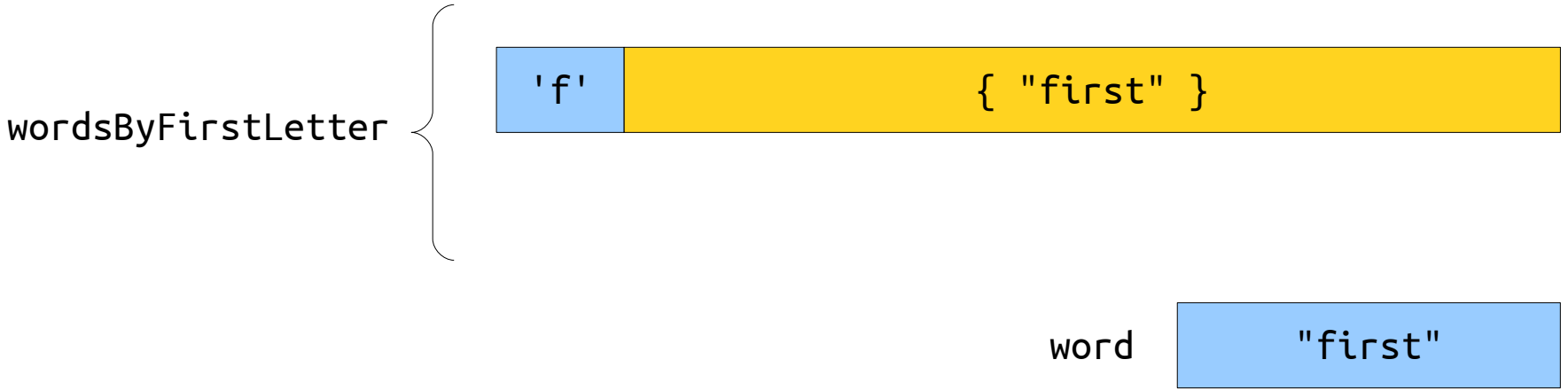
Map Autoinsertion

```
Lexicon english("EnglishWords.txt");

Map<char, Lexicon> wordsByFirstLetter;
for (string word: english) {
   wordsByFirstLetter[word[0]] += word;
}
```



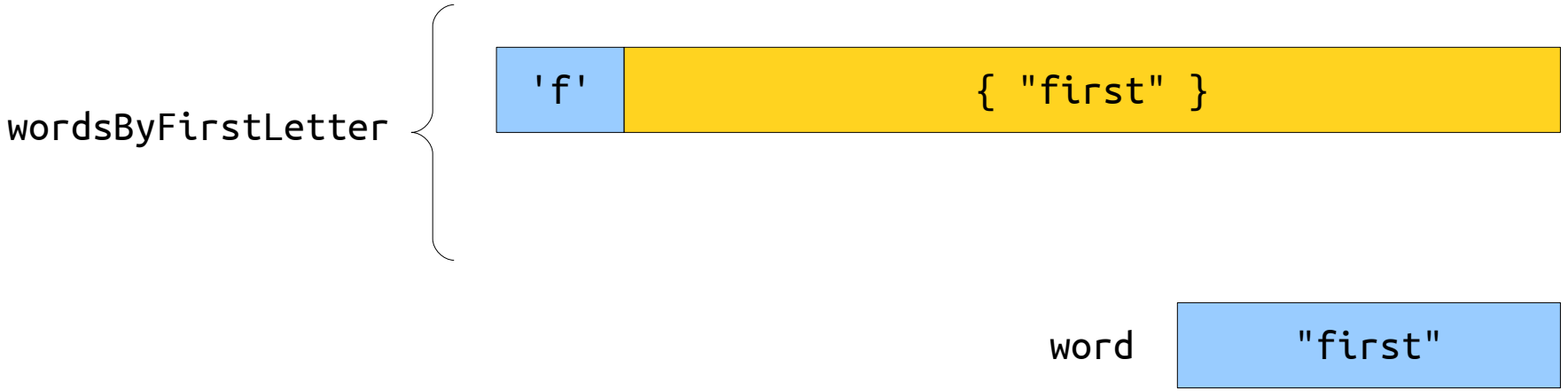
Map Autoinsertion

```
Lexicon english("EnglishWords.txt");

Map<char, Lexicon> wordsByFirstLetter;
for (string word: english) {
   wordsByFirstLetter[word[0]] += word;
}
```



Map Autoinsertion

```
Lexicon english("EnglishWords.txt");

Map<char, Lexicon> wordsByFirstLetter;
for (string word: english) {
    wordsByFirstLetter[word[0]] += word;
}
```



Map Autoinsertion

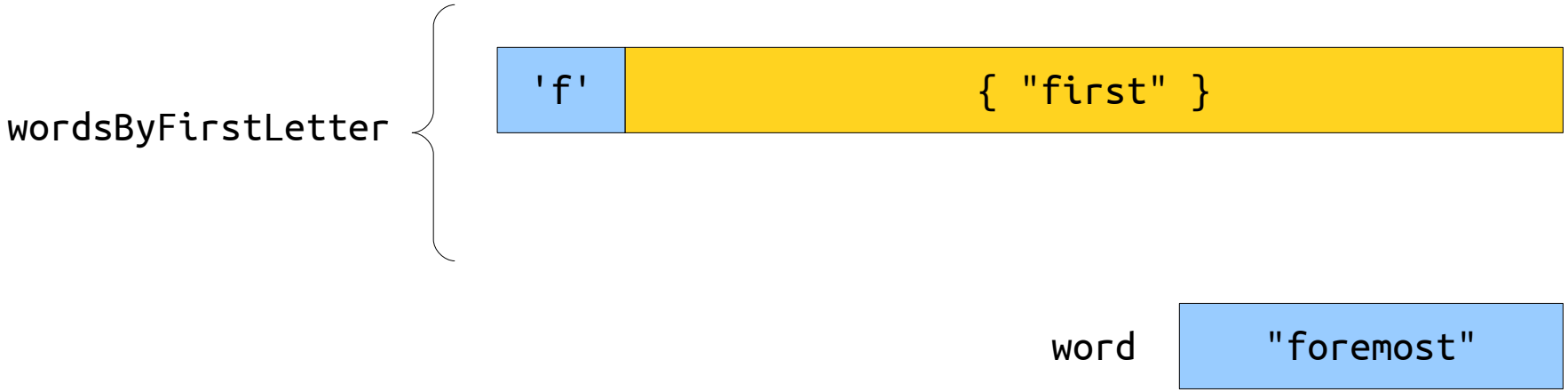
```
Lexicon english("EnglishWords.txt");

Map<char, Lexicon> wordsByFirstLetter;
for (string word: english) {
    wordsByFirstLetter[word[0]] += word;
}
```

Map Autoinsertion

```
Lexicon english("EnglishWords.txt");

Map<char, Lexicon> wordsByFirstLetter;
for (string word: english) {
    wordsByFirstLetter[word[0]] += word;
}
```

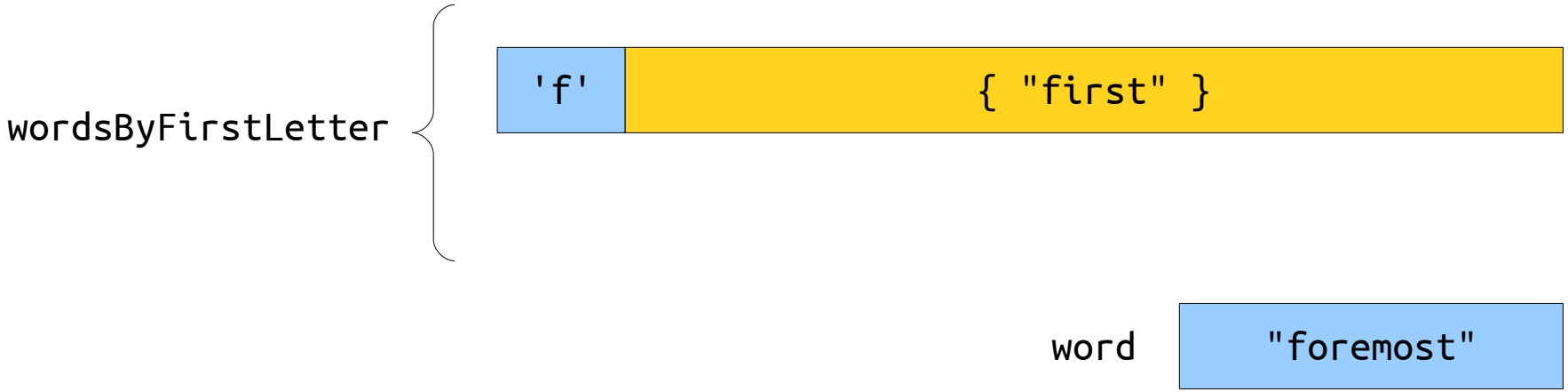
Map Autoinsertion

```
Lexicon english("EnglishWords.txt");

Map<char, Lexicon> wordsByFirstLetter;
for (string word: english) {
    wordsByFirstLetter[word[0]] += word;
}
```



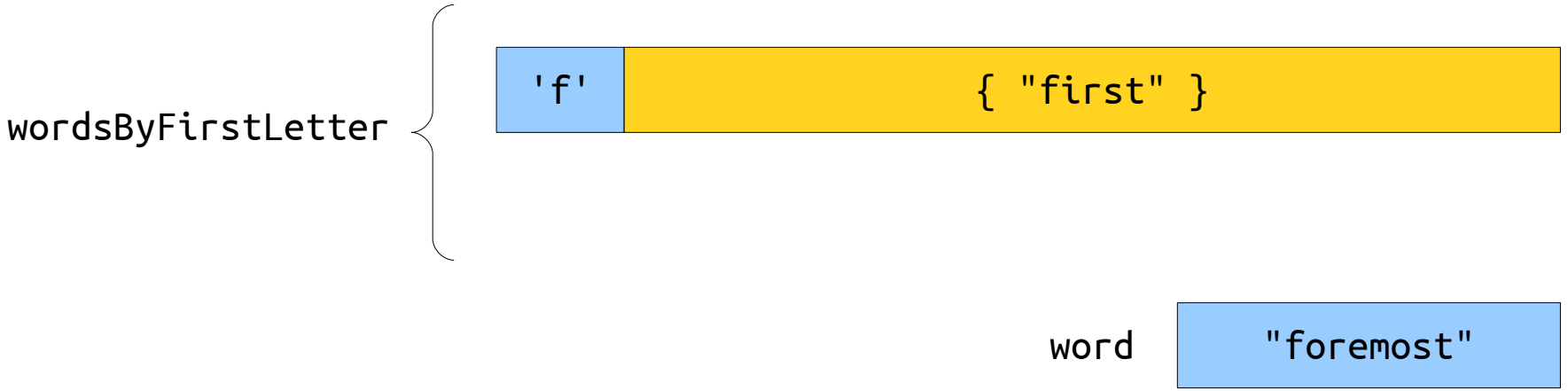
Map Autoinsertion

```
Lexicon english("EnglishWords.txt");

Map<char, Lexicon> wordsByFirstLetter;
for (string word: english) {
   wordsByFirstLetter[word[0]] += word;
}
```



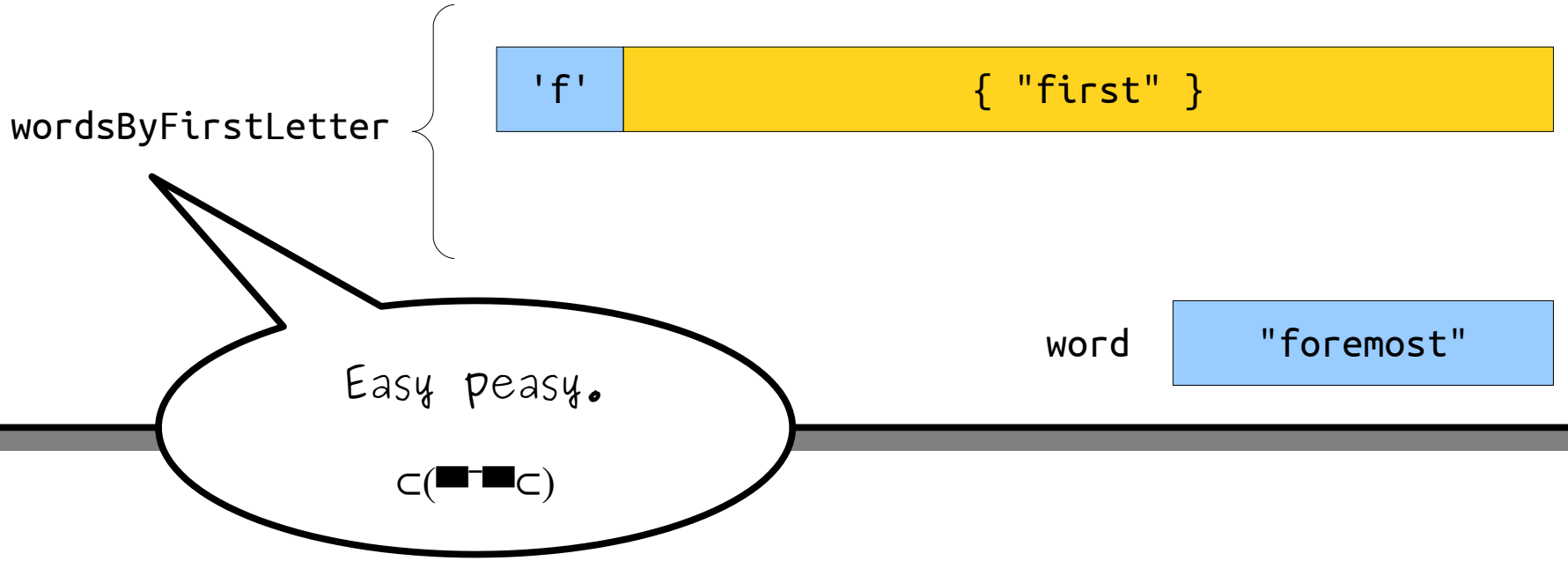
Map Autoinsertion

```
Lexicon english("EnglishWords.txt");

Map<char, Lexicon> wordsByFirstLetter;
for (string word: english) {
   wordsByFirstLetter[word[0]] += word;
}
```



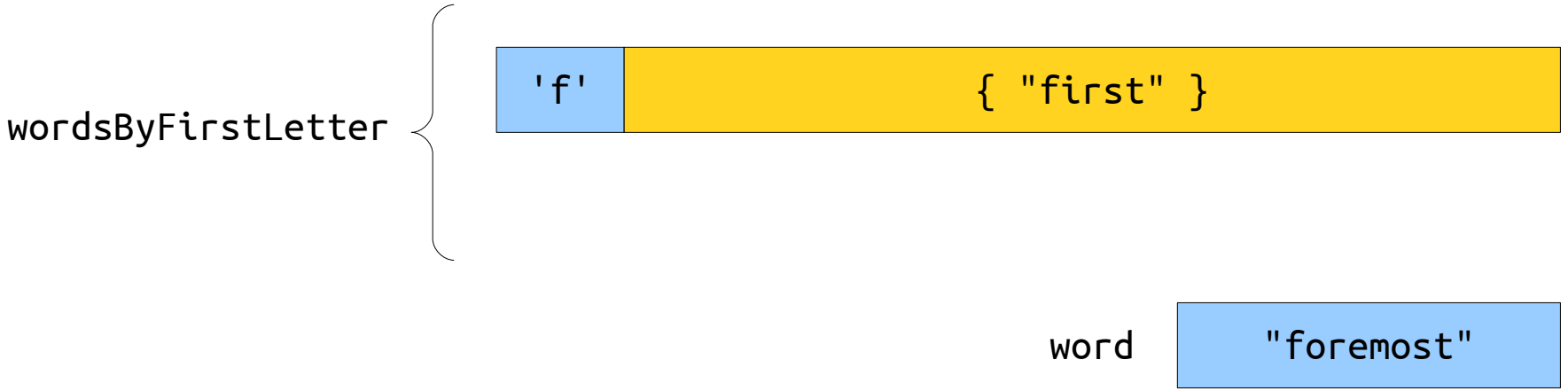
Map Autoinsertion

```
Lexicon english("EnglishWords.txt");

Map<char, Lexicon> wordsByFirstLetter;
for (string word: english) {
   wordsByFirstLetter[word[0]] += word;
}
```



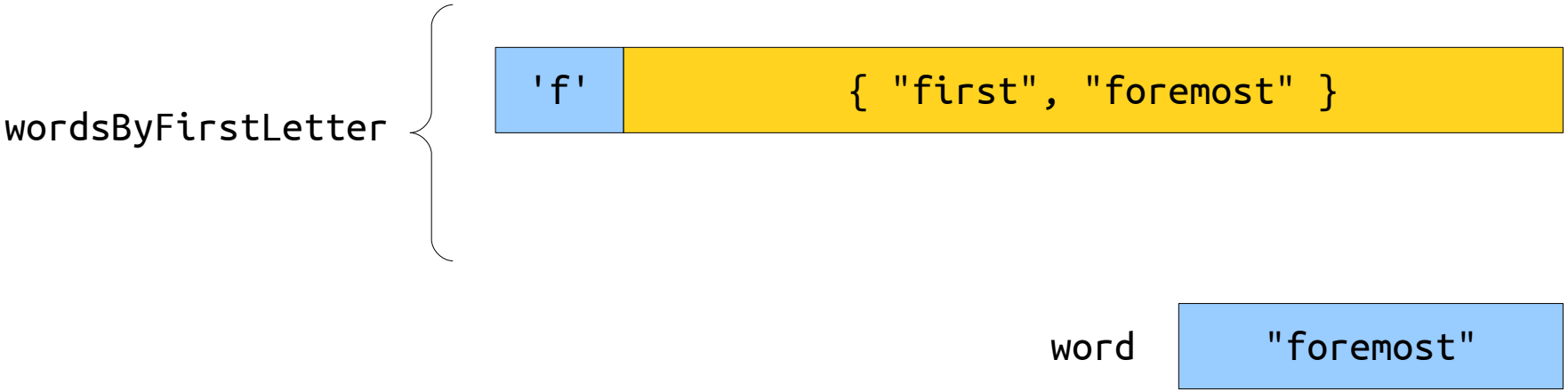
Map Autoinsertion

```
Lexicon english("EnglishWords.txt");

Map<char, Lexicon> wordsByFirstLetter;
for (string word: english) {
   wordsByFirstLetter[word[0]] += word;
}
```



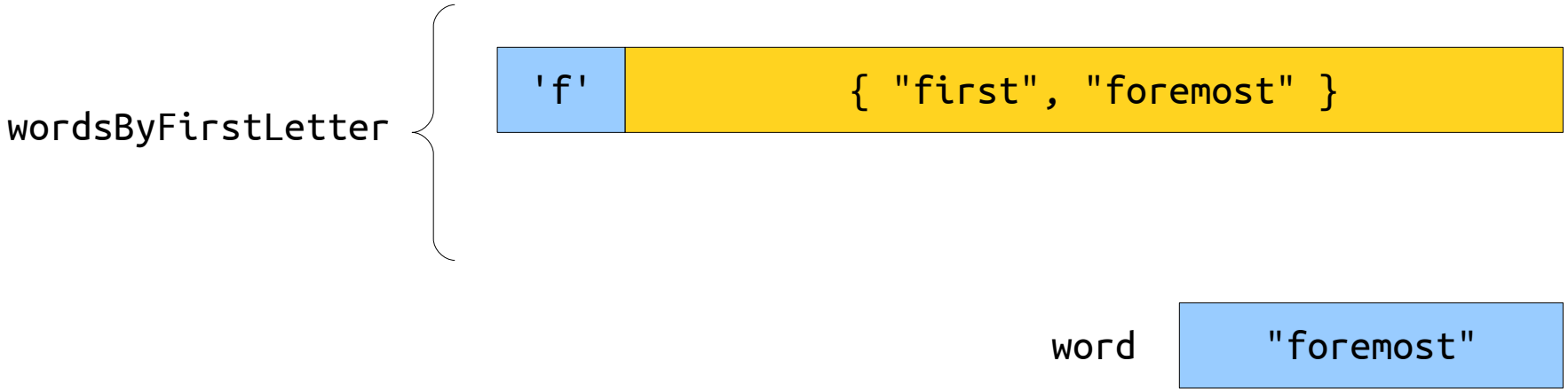
Map Autoinsertion

```
Lexicon english("EnglishWords.txt");

Map<char, Lexicon> wordsByFirstLetter;
for (string word: english) {
   wordsByFirstLetter[word[0]] += word;
}
```



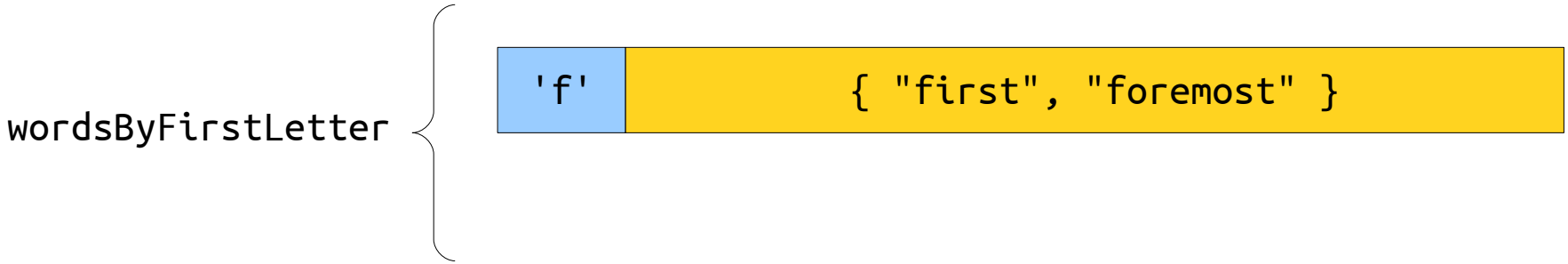
Map Autoinsertion

```
Lexicon english("EnglishWords.txt");

Map<char, Lexicon> wordsByFirstLetter;
for (string word: english) {
    wordsByFirstLetter[word[0]] += word;
}
```



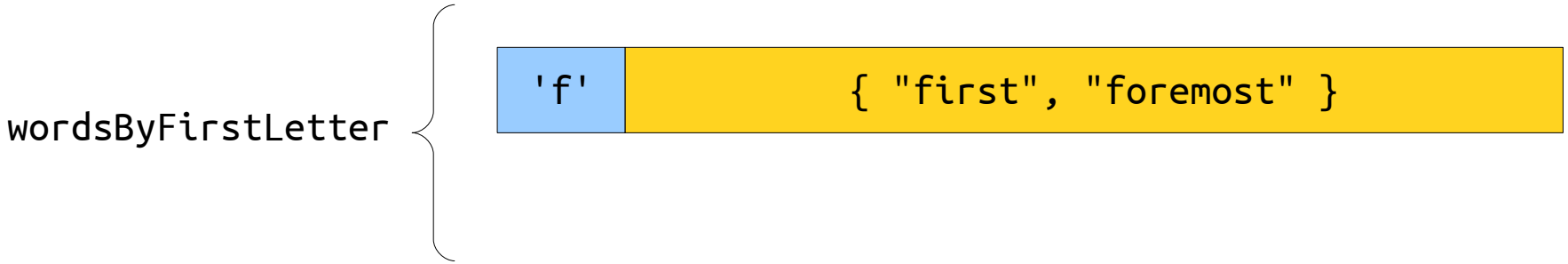
Map Autoinsertion

```
Lexicon english("EnglishWords.txt");

Map<char, Lexicon> wordsByFirstLetter;
for (string word: english) {
    wordsByFirstLetter[word[0]] += word;
}
```



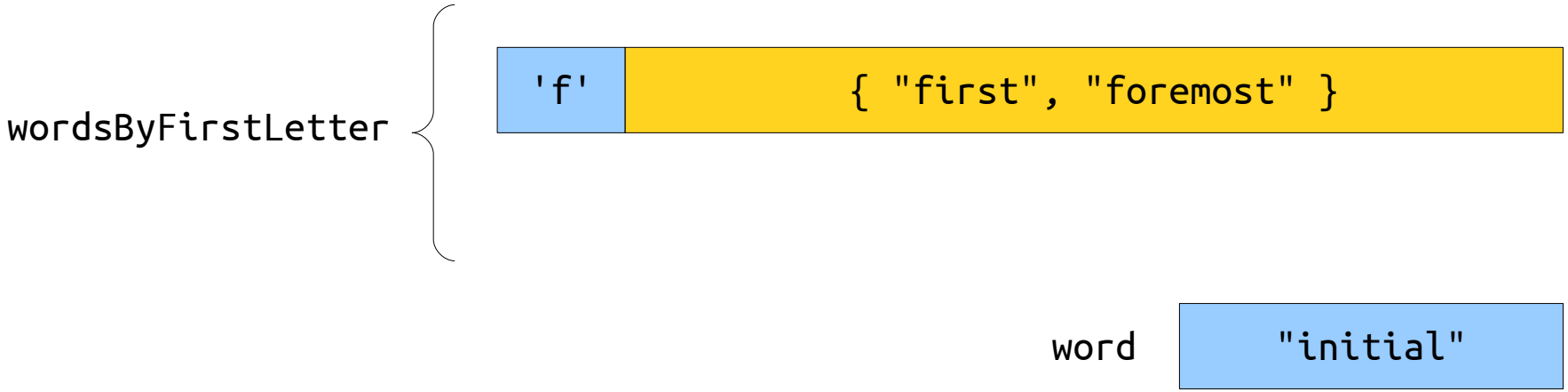
Map Autoinsertion

```
Lexicon english("EnglishWords.txt");

Map<char, Lexicon> wordsByFirstLetter;
for (string word: english) {
    wordsByFirstLetter[word[0]] += word;
}
```



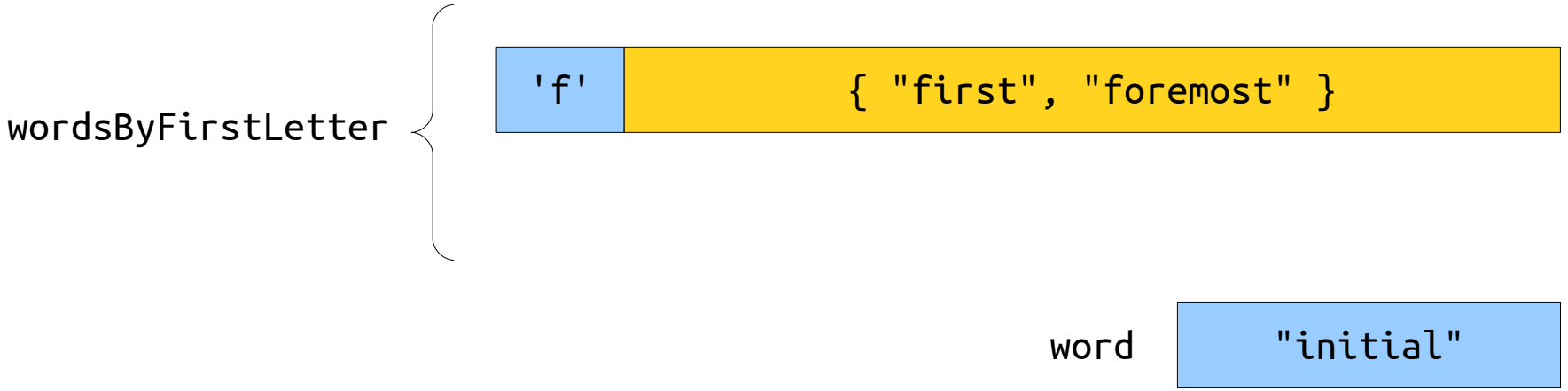
Map Autoinsertion

```
Lexicon english("EnglishWords.txt");

Map<char, Lexicon> wordsByFirstLetter;
for (string word: english) {
    wordsByFirstLetter[word[0]] += word;
}
```



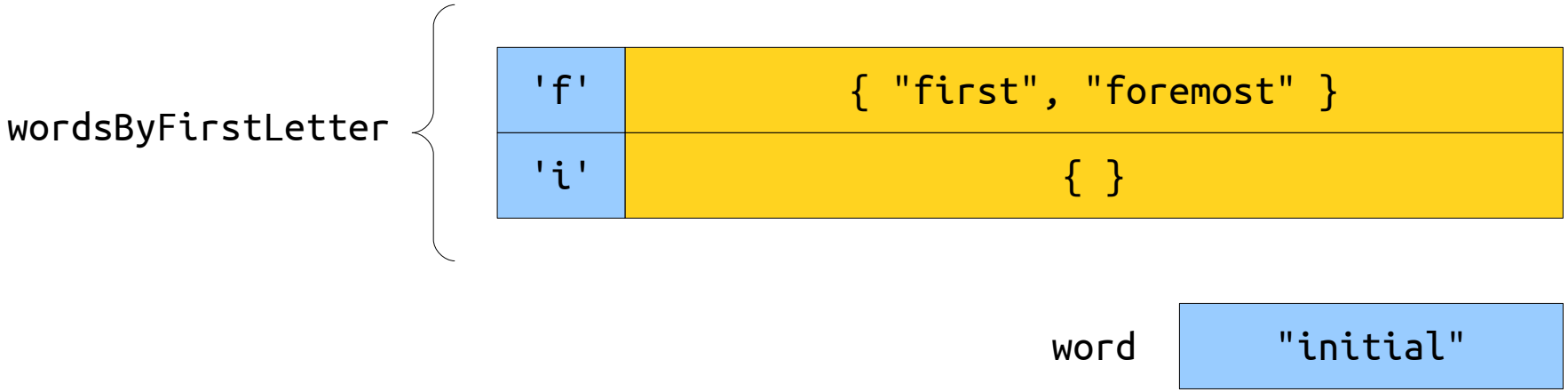
Map Autoinsertion

```
Lexicon english("EnglishWords.txt");

Map<char, Lexicon> wordsByFirstLetter;
for (string word: english) {
   wordsByFirstLetter[word[0]] += word;
}
```



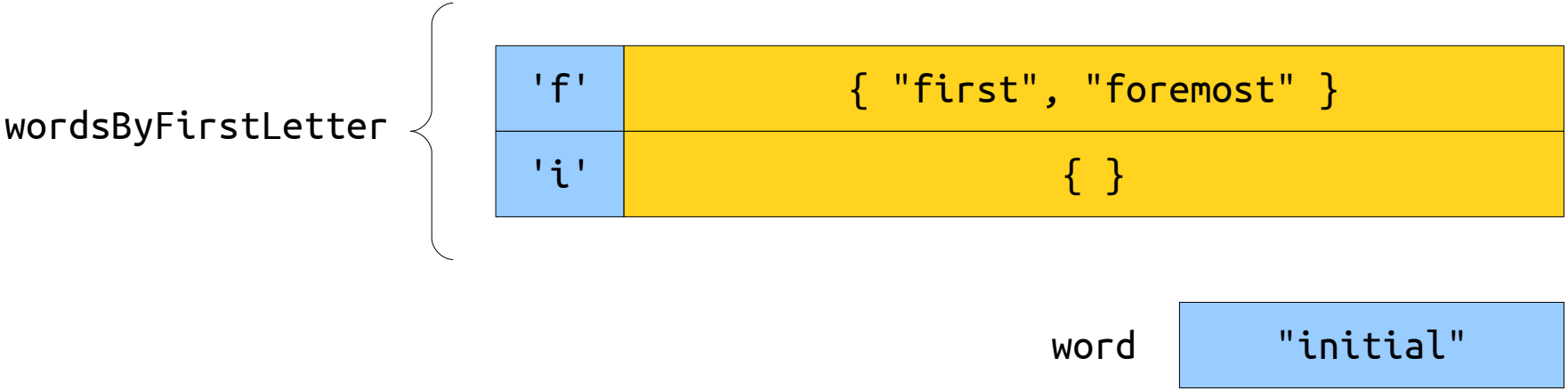
Map Autoinsertion

```
Lexicon english("EnglishWords.txt");

Map<char, Lexicon> wordsByFirstLetter;
for (string word: english) {
   wordsByFirstLetter[word[0]] += word;
}
```



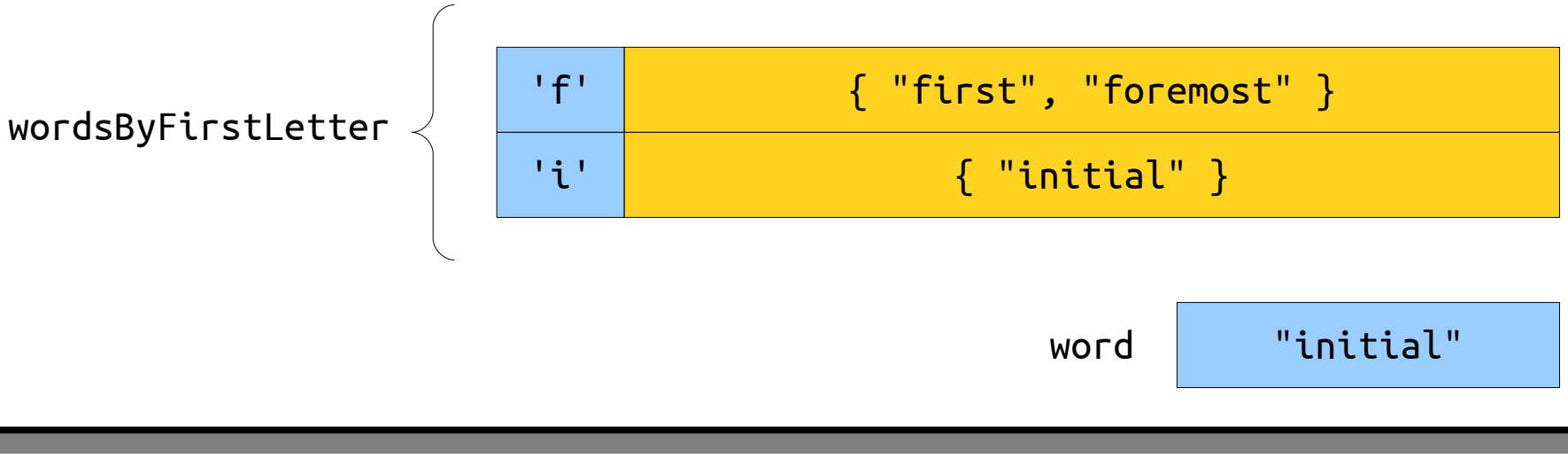
Map Autoinsertion

```
Lexicon english("EnglishWords.txt");

Map<char, Lexicon> wordsByFirstLetter;
for (string word: english) {
   wordsByFirstLetter[word[0]] += word;
}
```



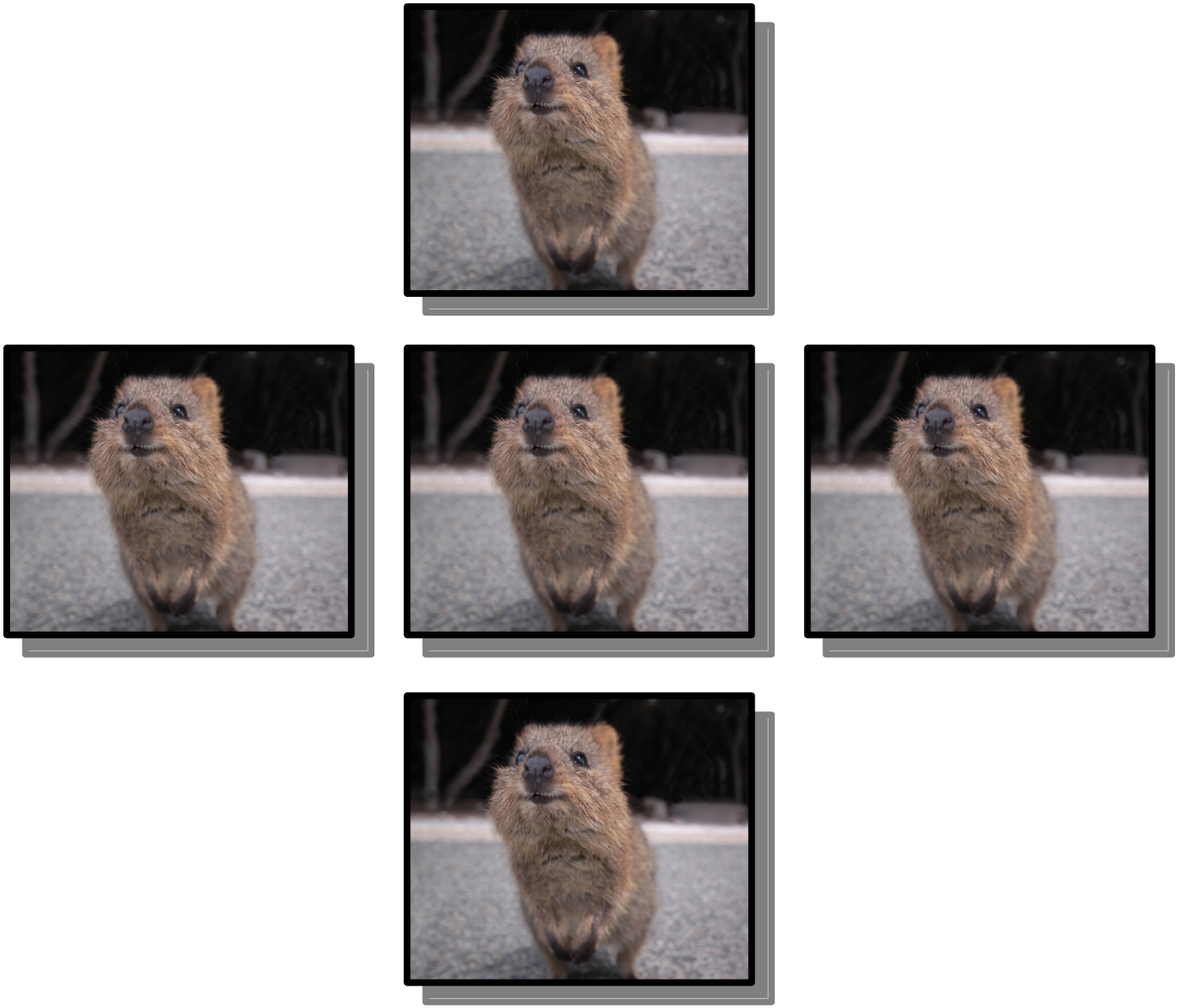
Map Autoinsertion

```
Lexicon english("EnglishWords.txt");

Map<char, Lexicon> wordsByFirstLetter;
for (string word: english) {
   wordsByFirstLetter[word[0]] += word;
}
```

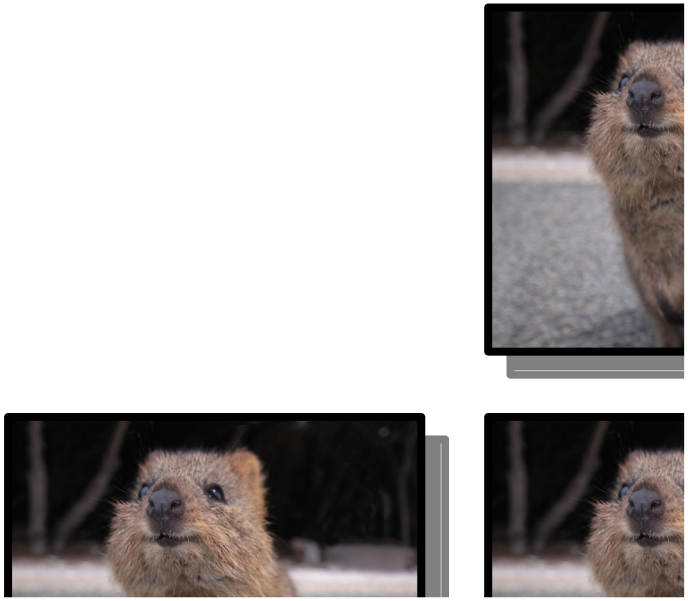


Quokka

Quokka Quincunx



Quarter Quokka Quincunx



Your Action Items

- ***Read Chapter 5.***
  - It's all about container types, and it'll fill in any remaining gaps from this week.
- ***Start Assignment 2.***
  - Make slow and steady progress here, if you can. Aim to finish Rosetta Stone and make some initial progress into Rising Tides before we return on Wednesday.

Next Time

- ***Thinking Recursively***
  - How can you best solve problems using recursion?
  - What techniques are necessary to do so?
  - And what problems yield easily to a recursive solution?