

Containers

Part Two

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



One Bulbul



More than One Caracara



Introducing the Dikdik



And a Music Recommendation



Set

Set

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

Set

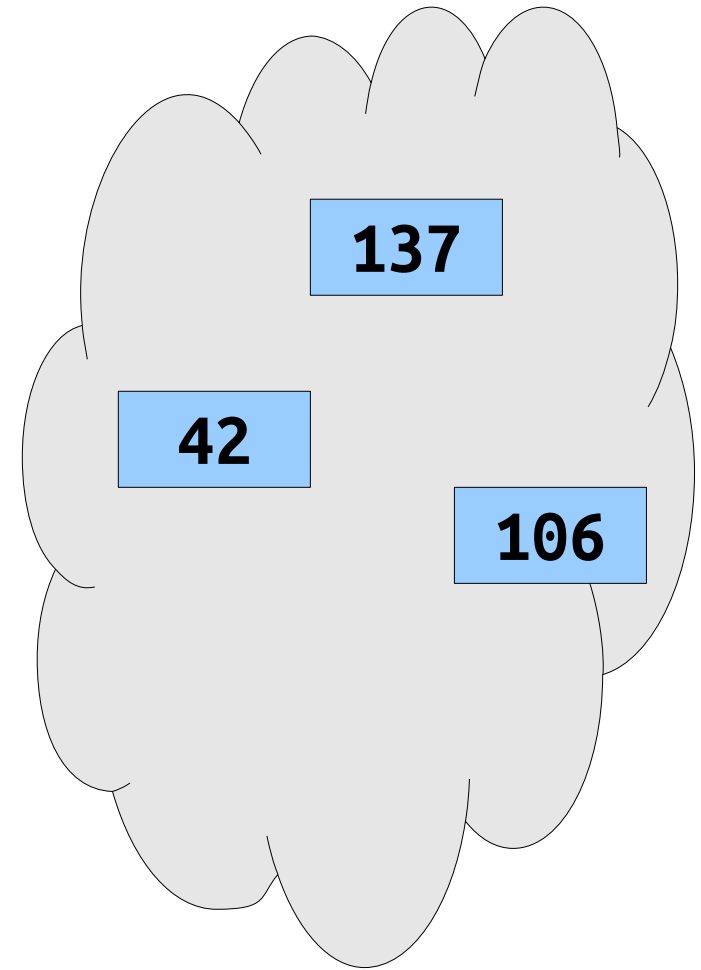
- The **Set** represents an unordered collection of distinct elements.
- Elements can be added and removed. Duplicates aren't allowed.

```
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.

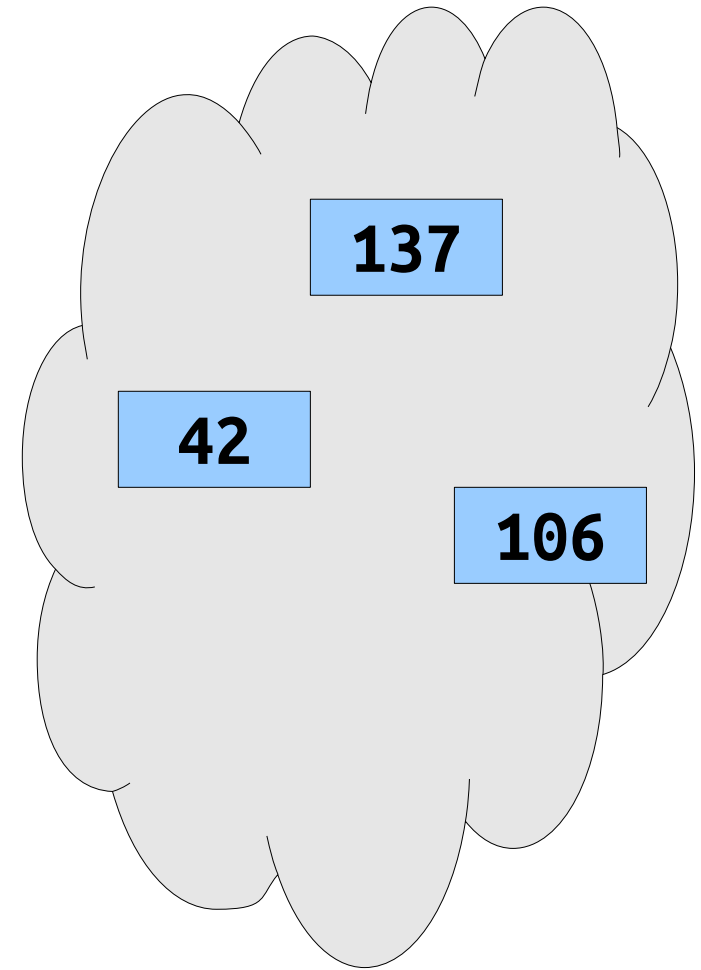
```
Set<int> values = {137, 106, 42};
```



Set

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

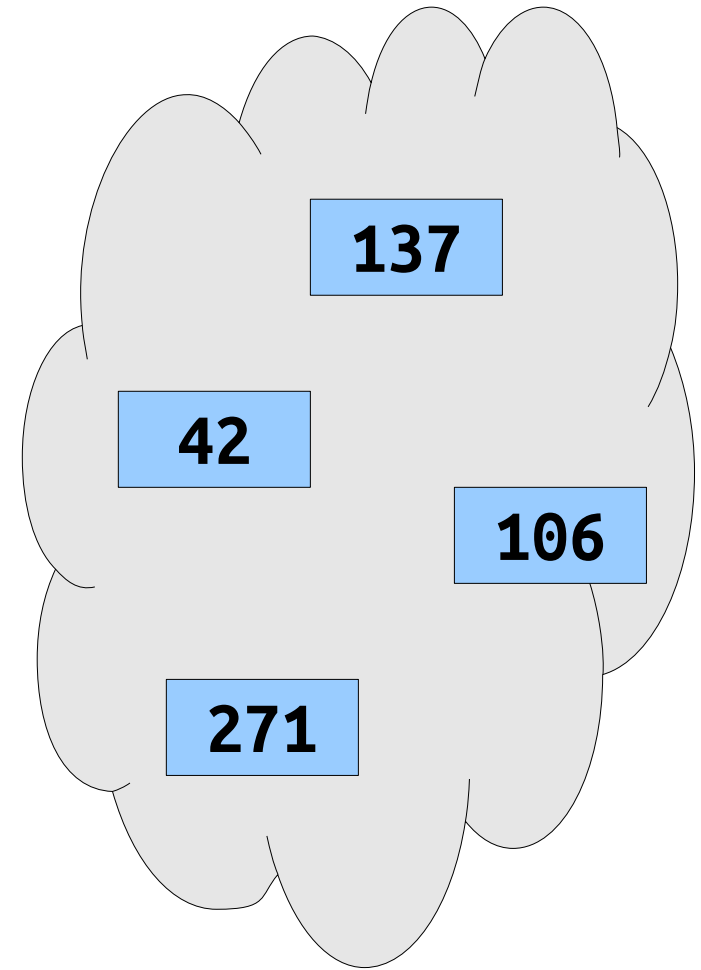
```
Set<int> values = {137, 106, 42};  
values += 271;
```



Set

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

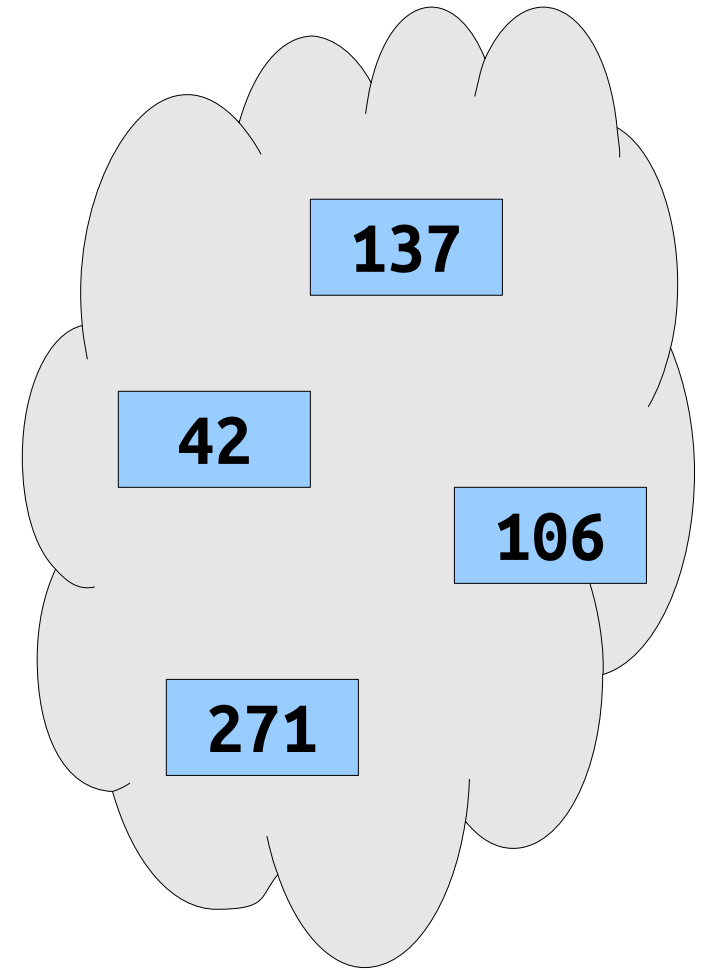
```
Set<int> values = {137, 106, 42};  
values += 271;
```



Set

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

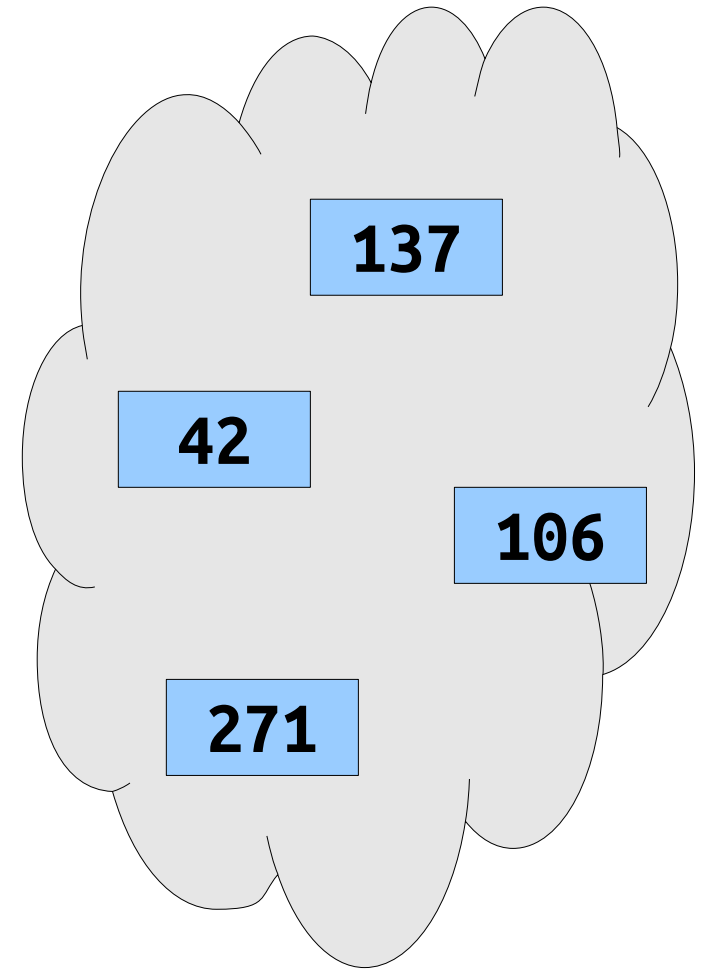
```
Set<int> values = {137, 106, 42};  
values += 271;  
values += 271;
```



Set

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

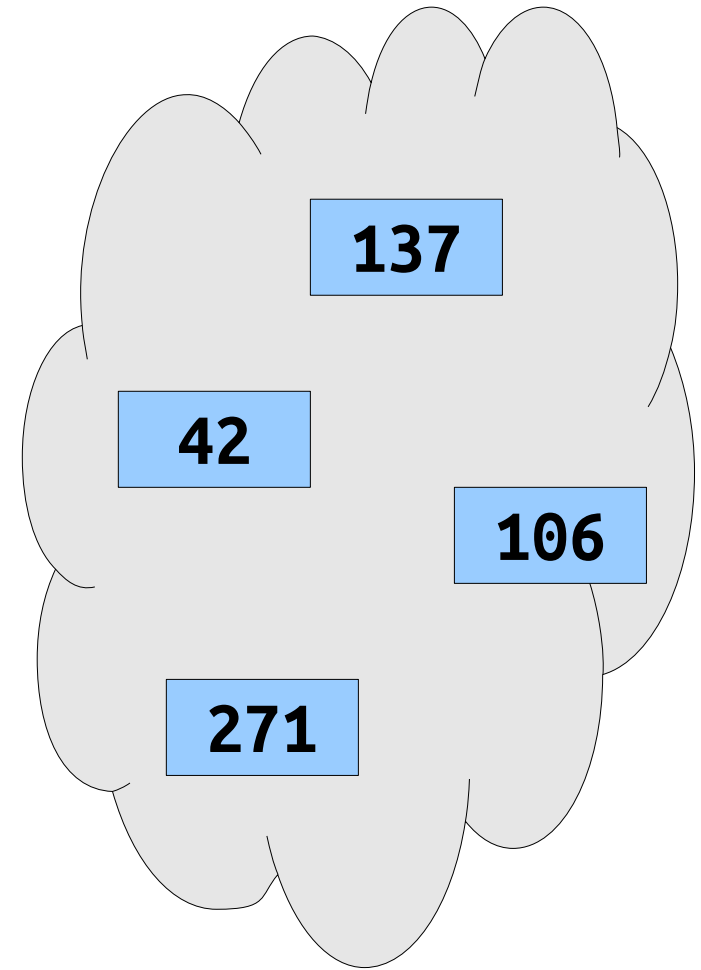
```
Set<int> values = {137, 106, 42};  
values += 271;  
values += 271; // Has no effect
```



Set

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

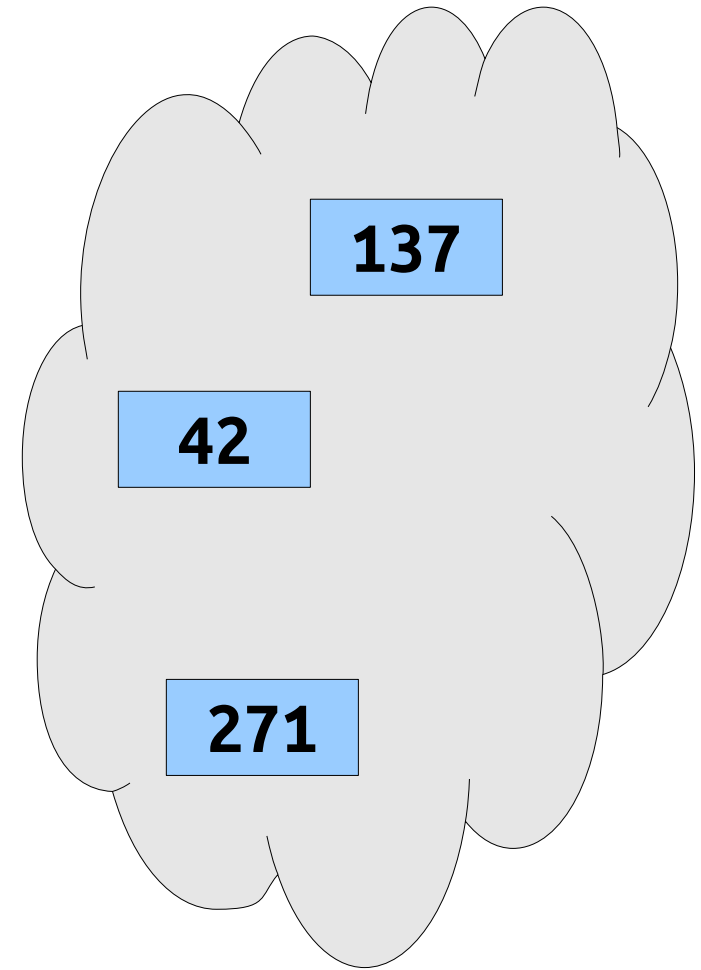
```
Set<int> values = {137, 106, 42};  
values += 271;  
values += 271; // Has no effect  
values -= 106;
```



Set

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

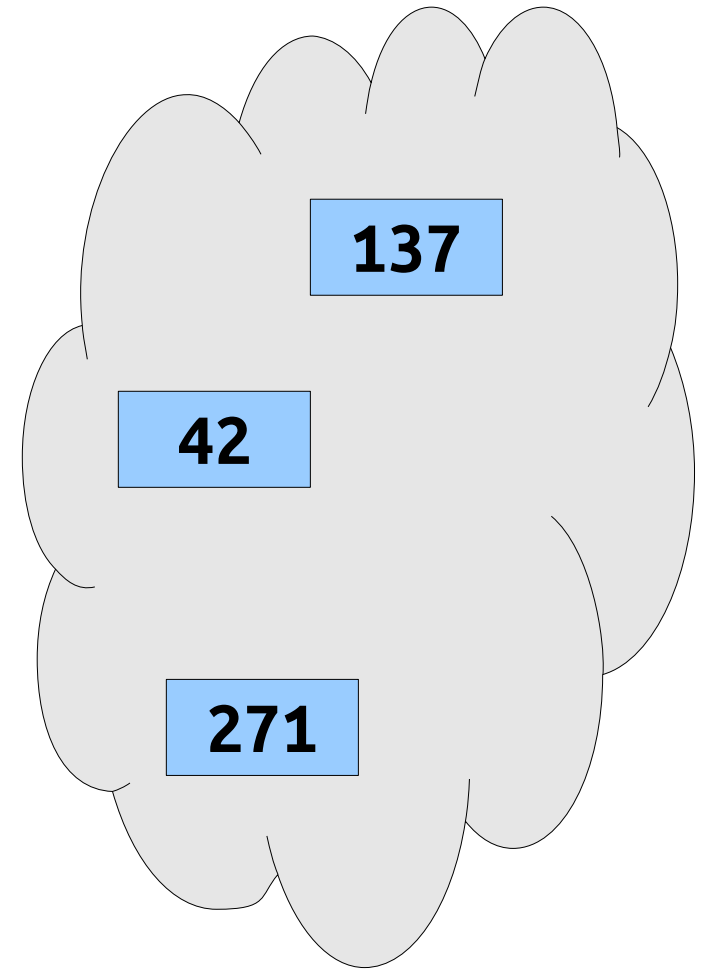
```
Set<int> values = {137, 106, 42};  
values += 271;  
values += 271; // Has no effect  
values -= 106;
```



Set

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

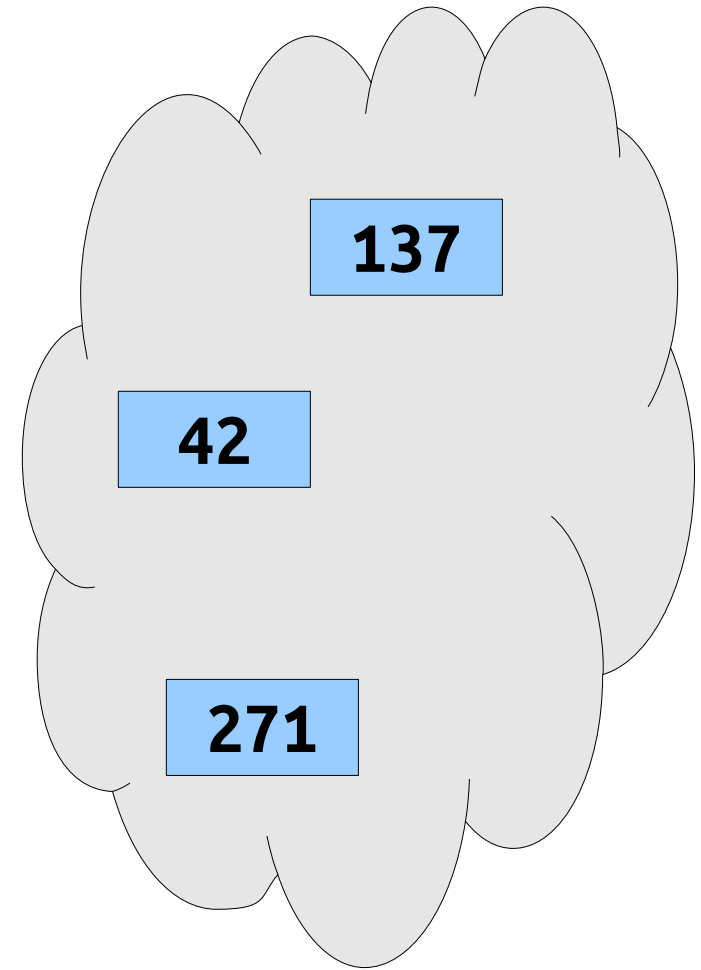
```
Set<int> values = {137, 106, 42};  
values += 271;  
values += 271; // Has no effect  
values -= 106;  
values -= 103;
```



Set

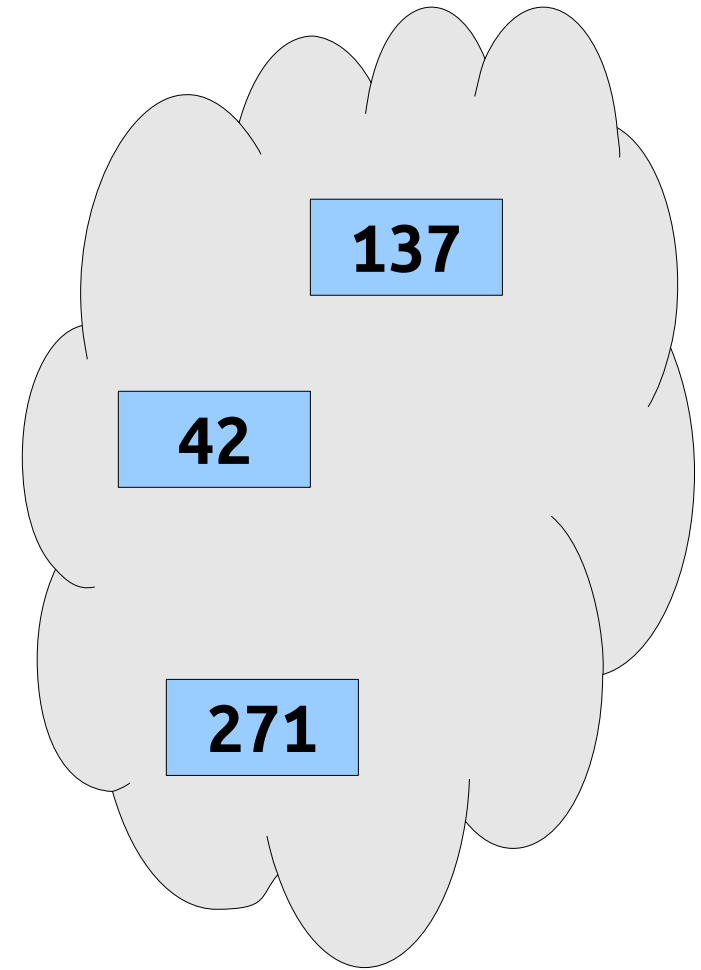
- The **Set** represents an unordered collection of distinct elements.
- Elements can be added and removed. Duplicates aren't allowed.

```
Set<int> values = {137, 106, 42};  
values += 271;  
values += 271; // Has no effect  
values -= 106;  
values -= 103; // Has no effect
```



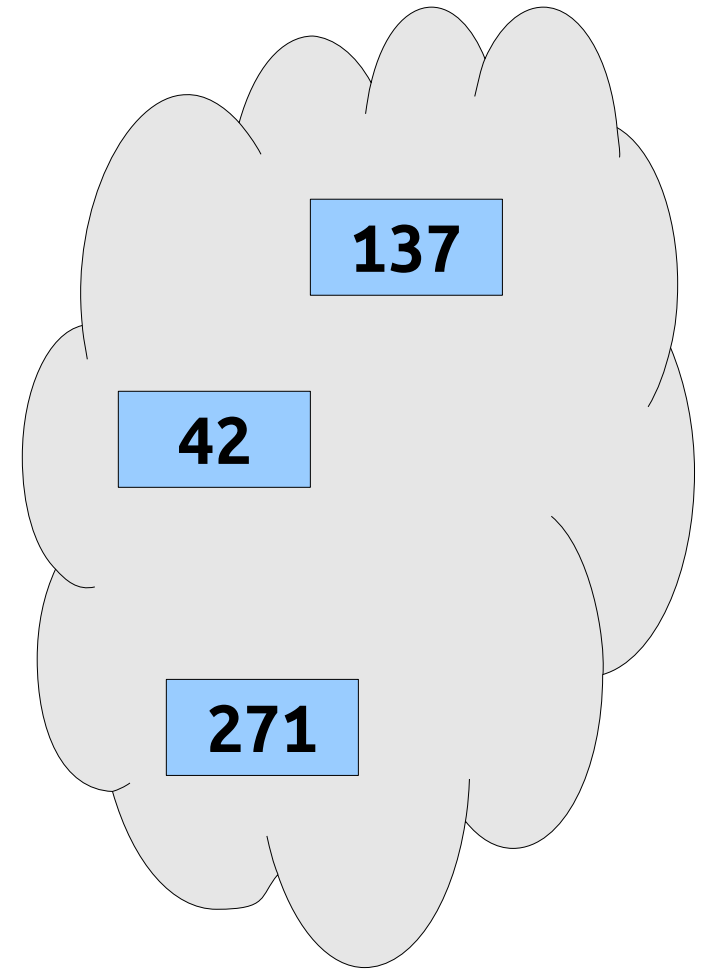
Set

- 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

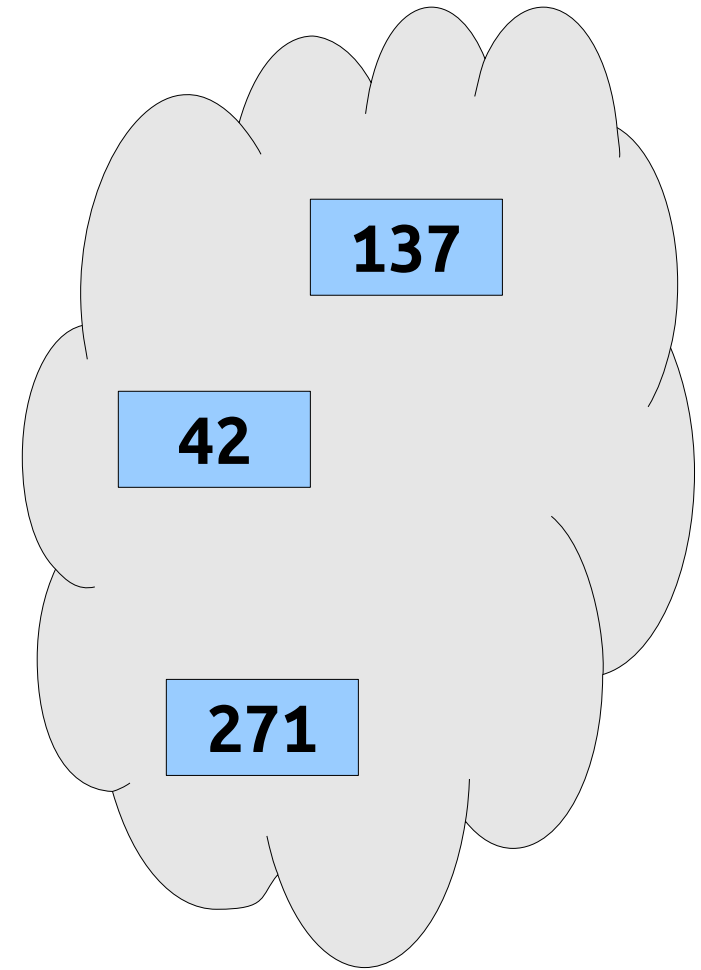
- Sets make it easy to check if you've seen something before.



Set

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

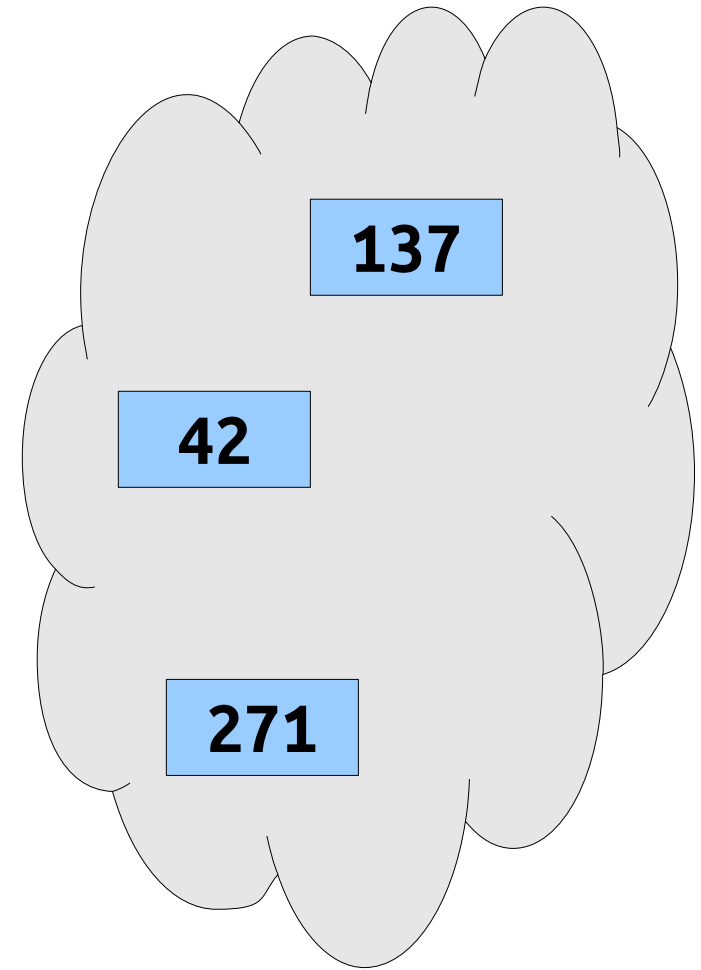
```
if (values.contains(137)) {  
    cout << "<(^_^)>" << endl;  
}
```



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.

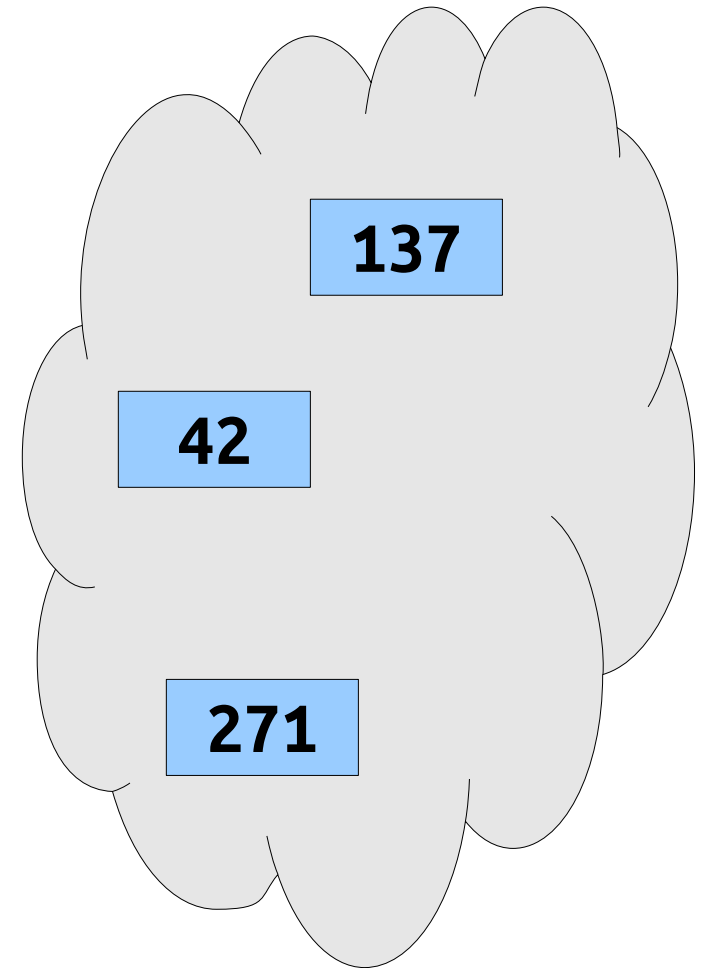
```
if (values.contains(137)) {  
    cout << "<(^_^)>" << endl;  
}
```



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;  
}
```



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!

Application: Word Economy

- Some long words are use few distinct letters.
 - “caracara” has length eight, but only uses the letters c, r, and a.
- The ***character efficiency*** of a word is the ratio of its length to the number of different letters it contains.
 - “caracara” has efficiency $8/3 \approx 2.67$.
- What is the highest-efficiency English word?

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;
```

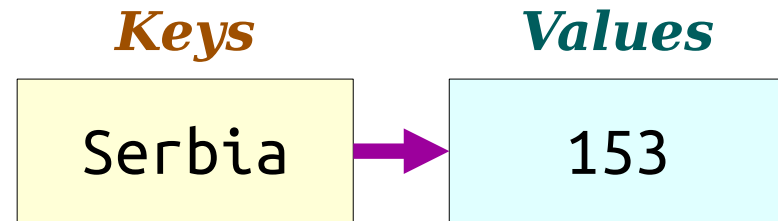
Map

- 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

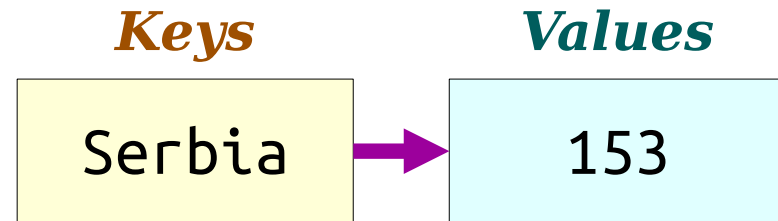
- The **Map** class represents a set of key/value pairs.
 - It's analogous to dict in Python, to Map in Java, and to objects (used as key/value stores) in JavaScript.
- Each key is associated with a value.



```
Map<string, int> heights;  
heights["Serbia"] = 153;
```

Map

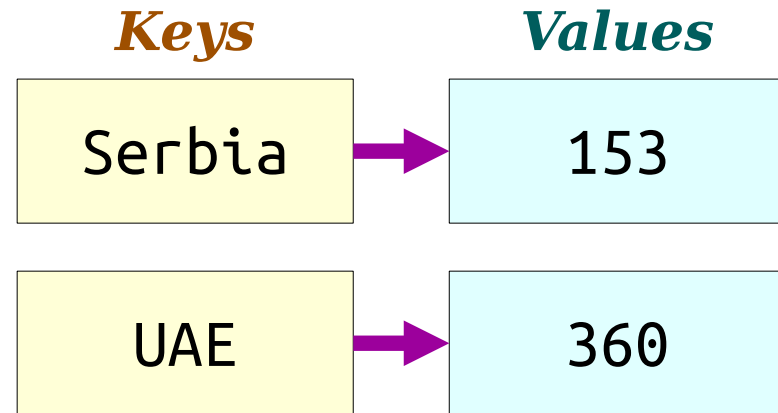
- The **Map** class represents a set of key/value pairs.
 - It's analogous to dict in Python, to Map in Java, and to objects (used as key/value stores) in JavaScript.
- Each key is associated with a value.



```
Map<string, int> heights;  
  
heights["Serbia"] = 153;  
heights["UAE"] = 360;
```

Map

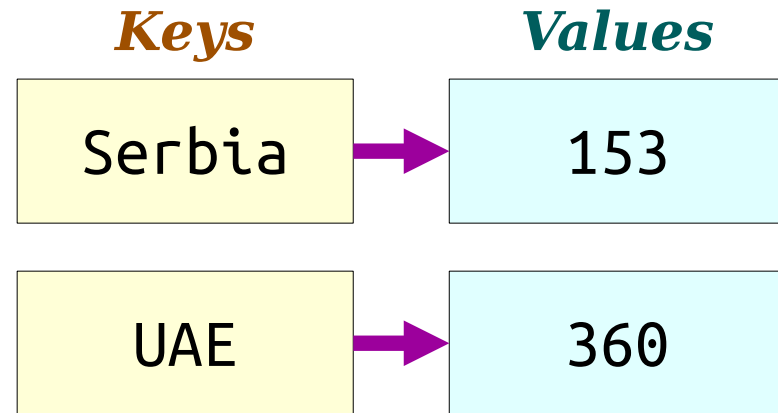
- The **Map** class represents a set of key/value pairs.
 - It's analogous to dict in Python, to Map in Java, and to objects (used as key/value stores) in JavaScript.
- Each key is associated with a value.



```
Map<string, int> heights;  
  
heights["Serbia"] = 153;  
heights["UAE"] = 360;
```

Map

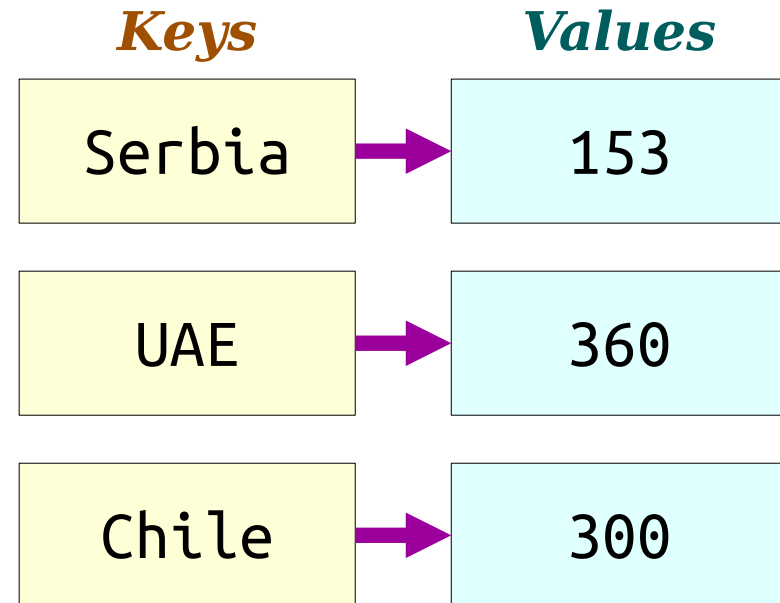
- The **Map** class represents a set of key/value pairs.
 - It's analogous to dict in Python, to Map in Java, and to objects (used as key/value stores) in JavaScript.
- Each key is associated with a value.



```
Map<string, int> heights;  
  
heights["Serbia"] = 153;  
heights["UAE"] = 360;  
heights["Chile"] = 300;
```

Map

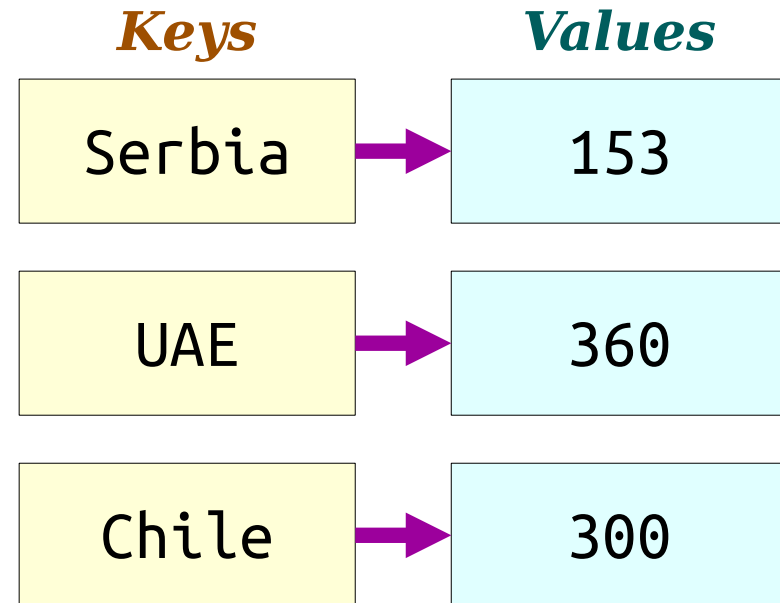
- The **Map** class represents a set of key/value pairs.
 - It's analogous to dict in Python, to Map in Java, and to objects (used as key/value stores) in JavaScript.
- Each key is associated with a value.



```
Map<string, int> heights;  
  
heights["Serbia"] = 153;  
heights["UAE"] = 360;  
heights["Chile"] = 300;
```

Map

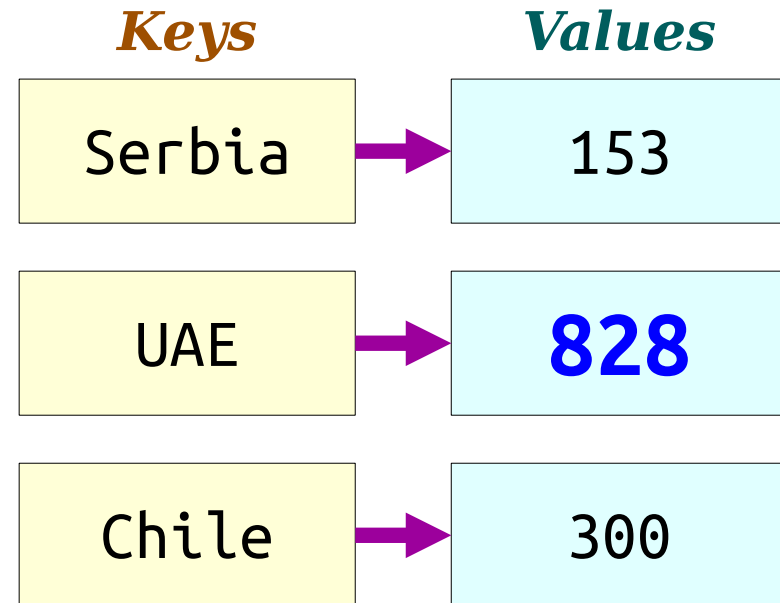
- The **Map** class represents a set of key/value pairs.
 - It's analogous to dict in Python, to Map in Java, and to objects (used as key/value stores) in JavaScript.
- Each key is associated with a value.



```
Map<string, int> heights;  
  
heights["Serbia"] = 153;  
heights["UAE"] = 360;  
heights["Chile"] = 300;  
heights["UAE"] = 828;
```

Map

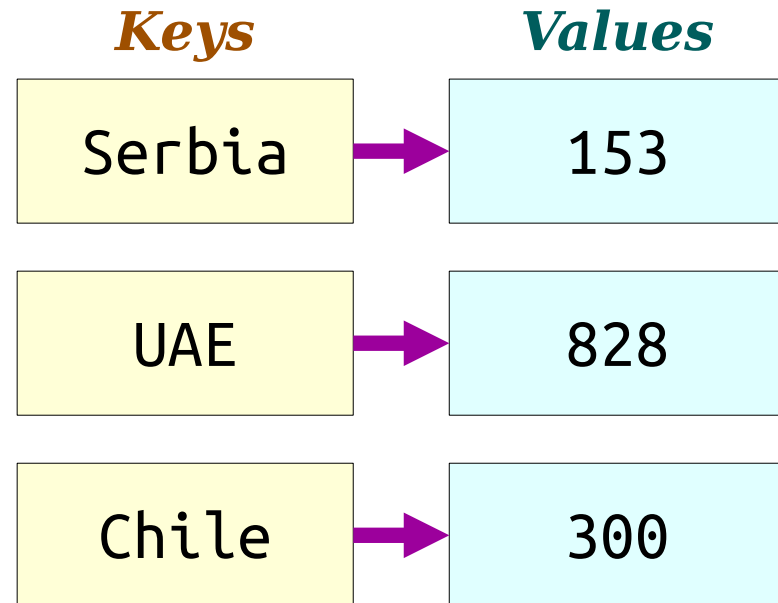
- The **Map** class represents a set of key/value pairs.
 - It's analogous to dict in Python, to Map in Java, and to objects (used as key/value stores) in JavaScript.
- Each key is associated with a value.



```
Map<string, int> heights;  
  
heights["Serbia"] = 153;  
heights["UAE"] = 360;  
heights["Chile"] = 300;  
heights["UAE"] = 828;
```

Map

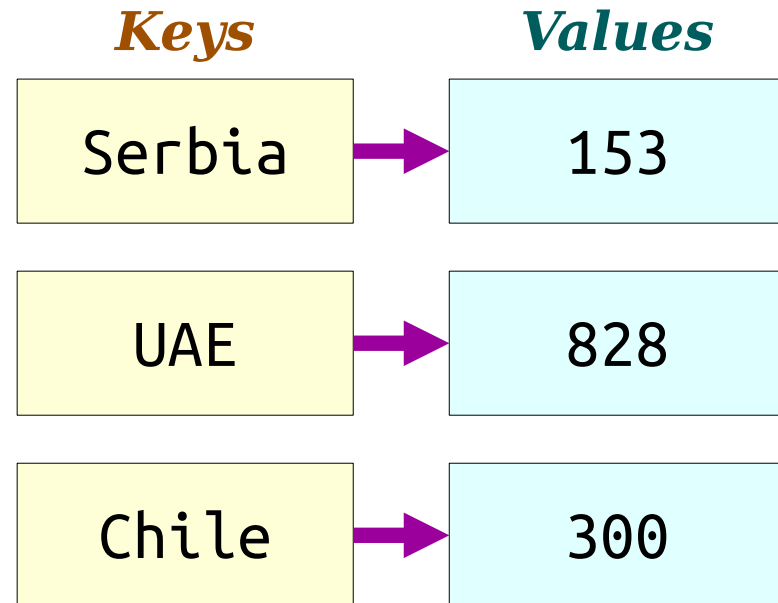
- The **Map** class represents a set of key/value pairs.
 - It's analogous to dict in Python, to Map in Java, and to objects (used as key/value stores) in JavaScript.
- Each key is associated with a value.



```
Map<string, int> heights;  
  
heights["Serbia"] = 153;  
heights["UAE"] = 360;  
heights["Chile"] = 300;  
heights["UAE"] = 828;
```

Map

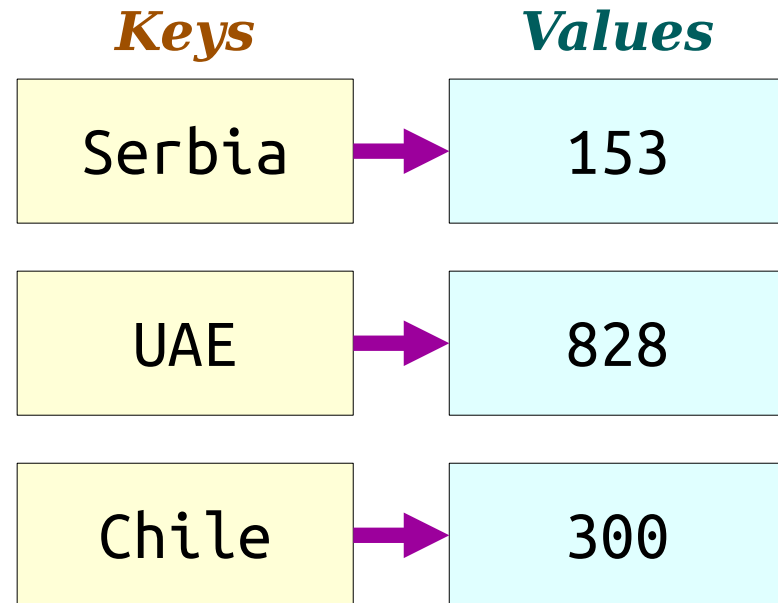
- The **Map** class represents a set of key/value pairs.
 - It's analogous to dict in Python, to Map in Java, and to objects (used as key/value stores) in JavaScript.
- Each key is associated with a value.
- Given a key, we can look up the associated value.



```
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.



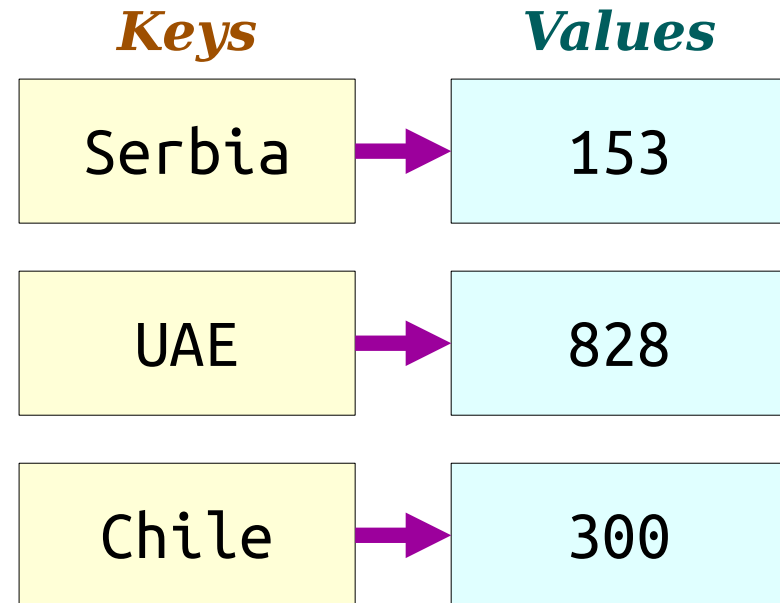
```
Map<string, int> heights;

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

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

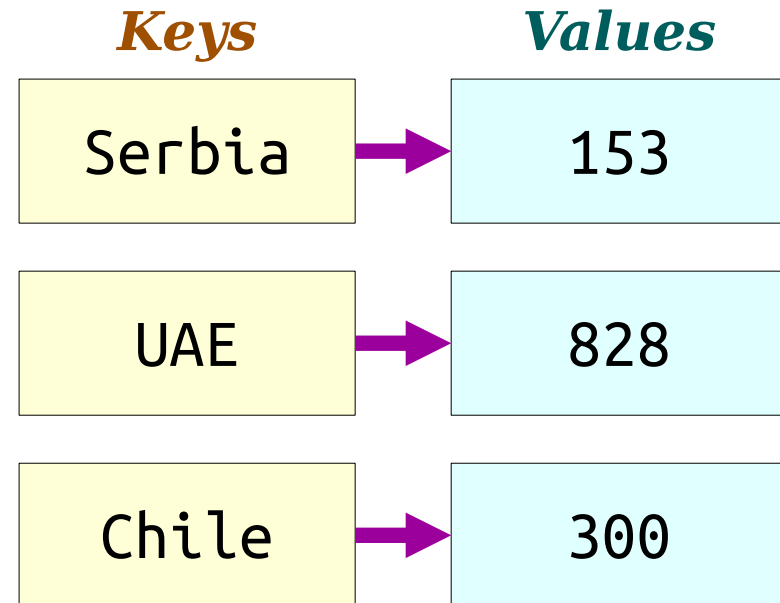
Map

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



Map

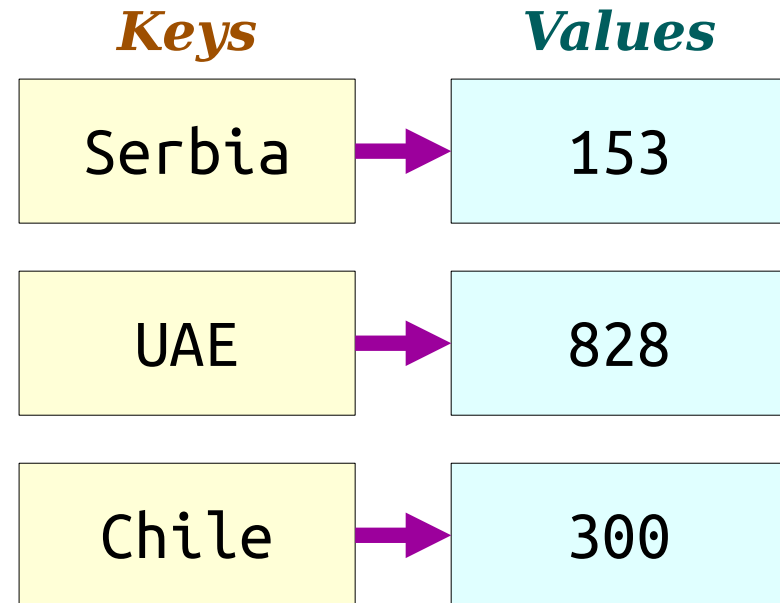
- We can loop over the keys in a map with a range-based for loop.



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

Map

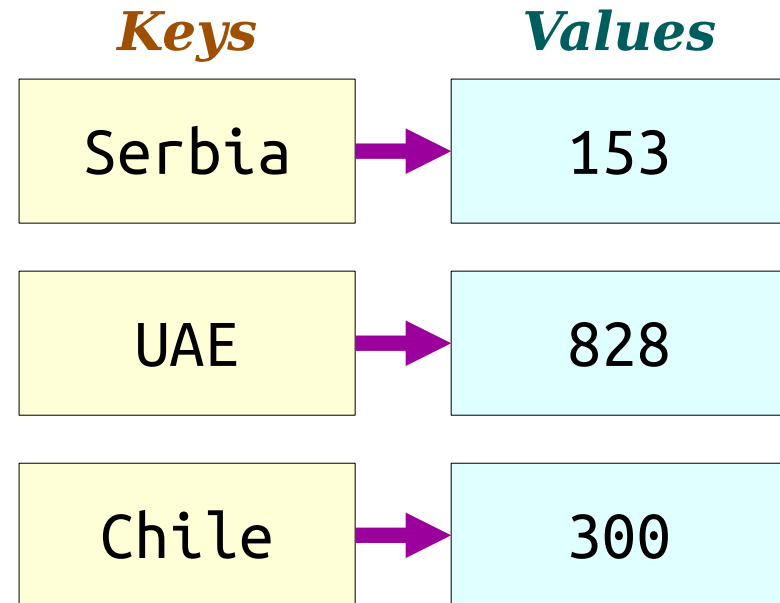
- We can loop over the keys in a map with a range-based for loop.
- We can check whether a key is present in the map.



```
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;  
}
```

What'd I Say?

What'd I Say?

- Our program will prompt the user to repeatedly type in text.
- Each time, we'll report how many previous times the user has typed in that text.
- We'll use a Map to track frequencies!

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]++;  
}
```

freqMap {

Map Autoinsertion

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

freqMap {

Map Autoinsertion

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

freqMap {

Map Autoinsertion

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

freqMap

text

"Hello"

Map Autoinsertion

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

freqMap {

text

"Hello"

Map Autoinsertion

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

freqMap {

text

"Hello"

Map Autoinsertion

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

freqMap

text

"Hello"

Oh no! I don't
know what that is!

Map Autoinsertion

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

freqMap

"Hello"

text

"Hello"

Let's pretend
I already had that
key here.

Map Autoinsertion

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

freqMap

"Hello"

0

text

"Hello"

The values are
all ints, so I'll pick
zero.

Map Autoinsertion

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

freqMap

"Hello"

0

text

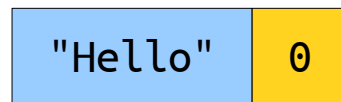
"Hello"

Phew! Crisis
averted!

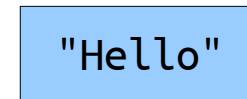
Map Autoinsertion

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

freqMap



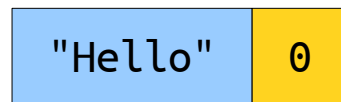
text



Map Autoinsertion

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

freqMap



text

"Hello"

Map Autoinsertion

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

freqMap

"Hello"

0

text

"Hello"

Cool as a cucumber.

c(■■c)

Map Autoinsertion

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

freqMap

"Hello"

1

text

"Hello"

Cool as a cucumber.

c(■■c)

Map Autoinsertion

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

}

freqMap



"Hello"	1
---------	---

text

"Hello"

Map Autoinsertion

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

}

freqMap

"Hello"

1

Map Autoinsertion

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

freqMap

"Hello"	1
---------	---

Map Autoinsertion

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

freqMap

"Hello"

1

Map Autoinsertion

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

freqMap

"Hello"

1

text

"Goodbye"

Map Autoinsertion

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

freqMap

"Hello"

1

text

"Goodbye"

Map Autoinsertion

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

freqMap

"Hello"

1

text

"Goodbye"

Map Autoinsertion

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

freqMap

"Hello"

1

text

"Goodbye"

Oh no, not again!

Map Autoinsertion

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

freqMap

"Hello"	1
"Goodbye"	0

text

"Goodbye"

I'll pretend
I already had that
key.

Map Autoinsertion

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

freqMap

"Hello"	1
"Goodbye"	0

text

"Goodbye"

Map Autoinsertion

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

freqMap

"Hello"	1
"Goodbye"	0

text

"Goodbye"

Map Autoinsertion

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

freqMap

"Hello"	1
"Goodbye"	0

text

"Goodbye"

Chillin' like a villain.

c(■■c)

Map Autoinsertion

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

freqMap

"Hello"	1
"Goodbye"	1

text

"Goodbye"

Chillin' like a villain.

c(■■c)

Map Autoinsertion

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

}

freqMap

"Hello"	1
"Goodbye"	1

text

"Goodbye"

Map Autoinsertion

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

}

freqMap

"Hello"	1
"Goodbye"	1

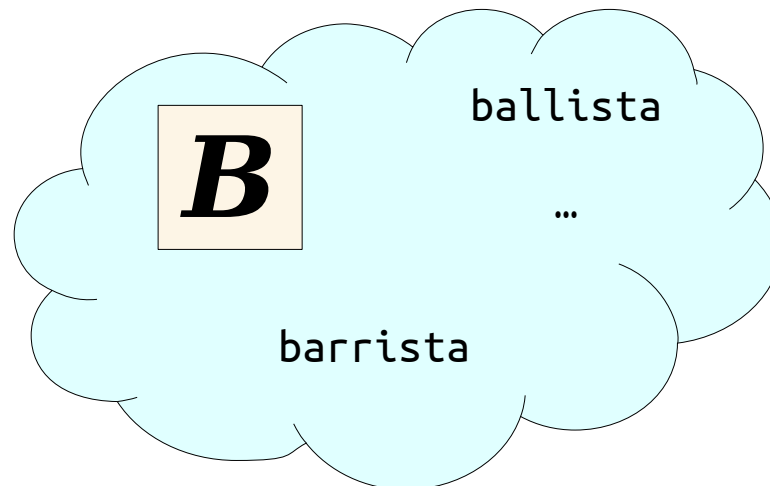
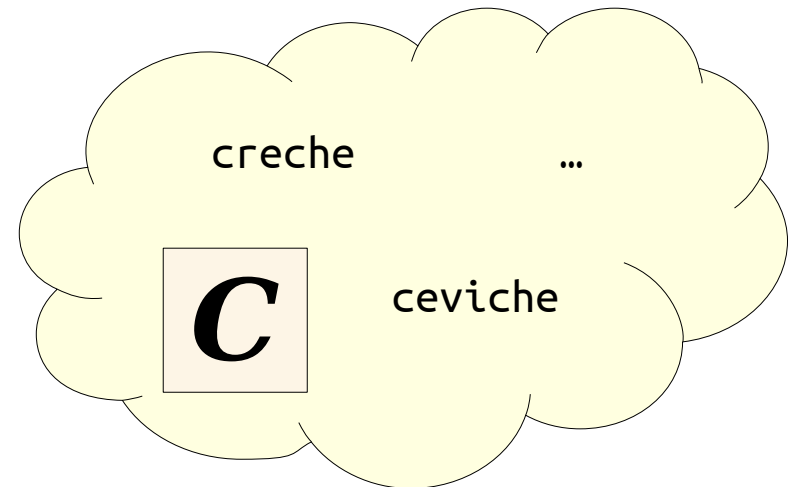
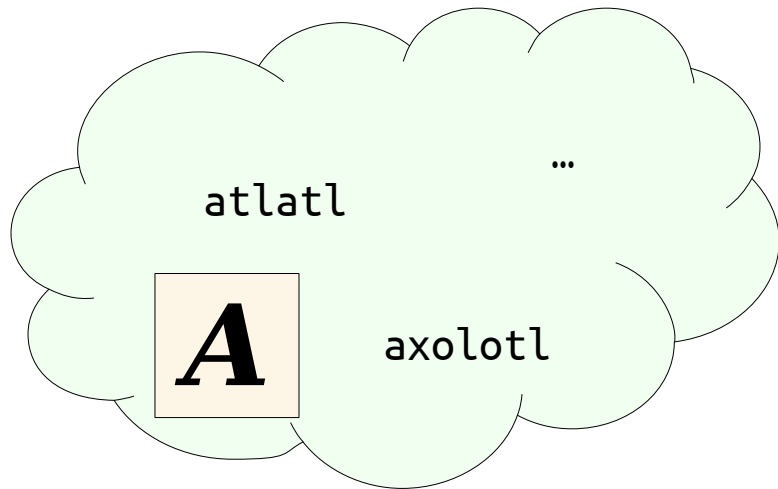
Map Autoinsertion

- 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.

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

Grouping by First Letters

Grouping by First Letters



Grouping by First Letters

- We'll partition all English words into groups based on their first letter.
- To do so, we'll create a Map that associates each letter with words starting with that letter.
- What specific type of Map should it be (e.g. Map<int, double>, Map<string, string>, etc.)?

Answer online at
<https://cs106b.stanford.edu/pollev>

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;  
}
```

wordsByFirstLetter {

word

"first"

Map Autoinsertion

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

wordsByFirstLetter {

word

"first"

Map Autoinsertion

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

wordsByFirstLetter {

word

"first"

Map Autoinsertion

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

wordsByFirstLetter {

word

"first"

Oops, no f's here.

Map Autoinsertion

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

wordsByFirstLetter { 'f'

word

"first"

Let's insert
that key.

Map Autoinsertion

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



word

"first"

I'll give you a
blank Lexicon.

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

'f'

{ "first" }

word

"first"

Map Autoinsertion

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

```
Map<char, Lexicon> wordsByFirstLetter;
```

```
for (string word: english) {  
    wordsByFirstLetter[word[0]] += word;
```

```
}
```

wordsByFirstLetter

'f'

{ "first" }

Map Autoinsertion

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



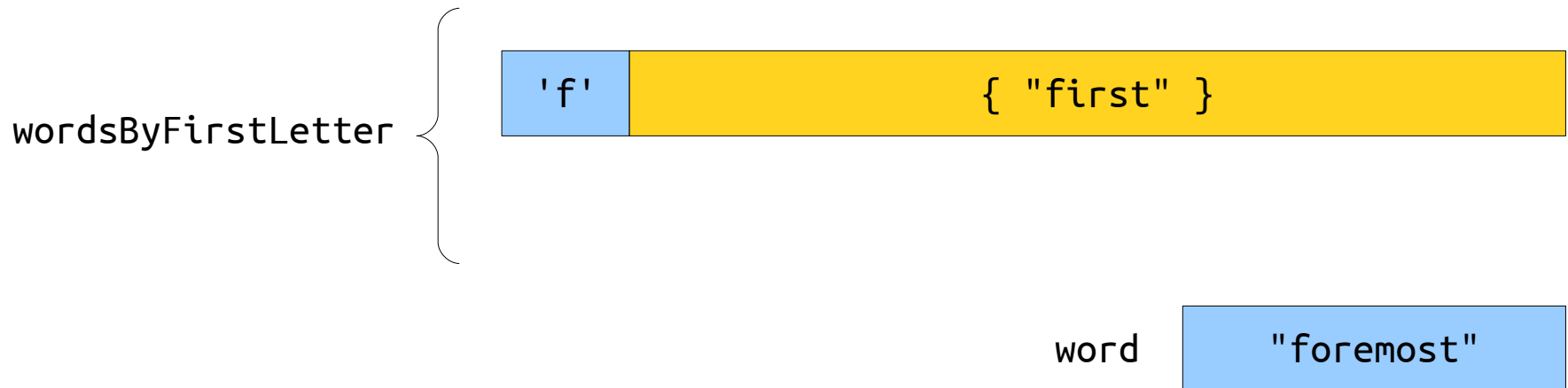
Map Autoinsertion

```
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;  
}
```



word

A blue box containing the string "foremost".

Easy peasy.

c(■■c)

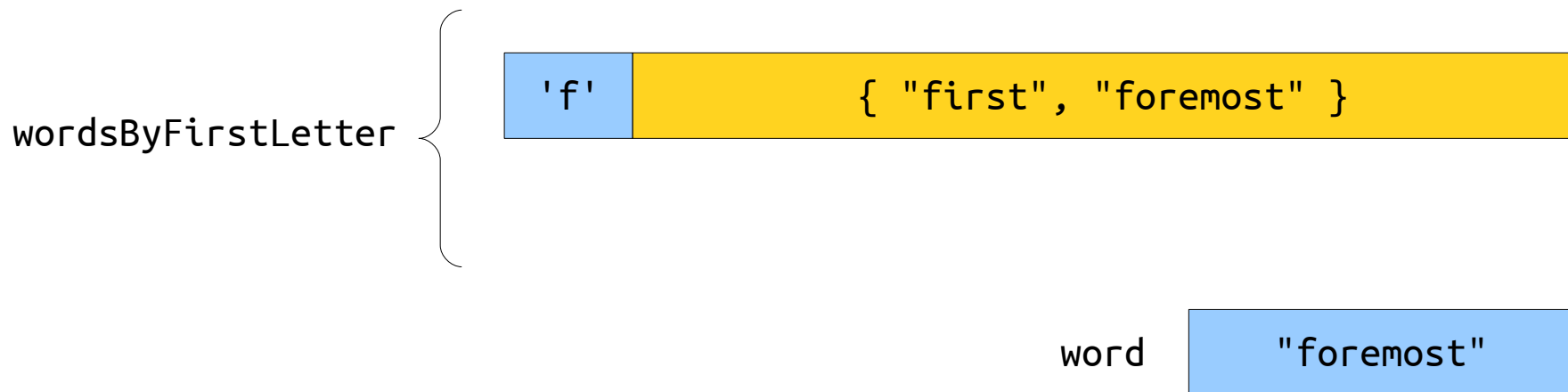
Map Autoinsertion

```
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

'f'

{ "first", "foremost" }

word

"foremost"

Map Autoinsertion

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

```
Map<char, Lexicon> wordsByFirstLetter;
```

```
for (string word: english) {  
    wordsByFirstLetter[word[0]] += word;
```

```
}
```

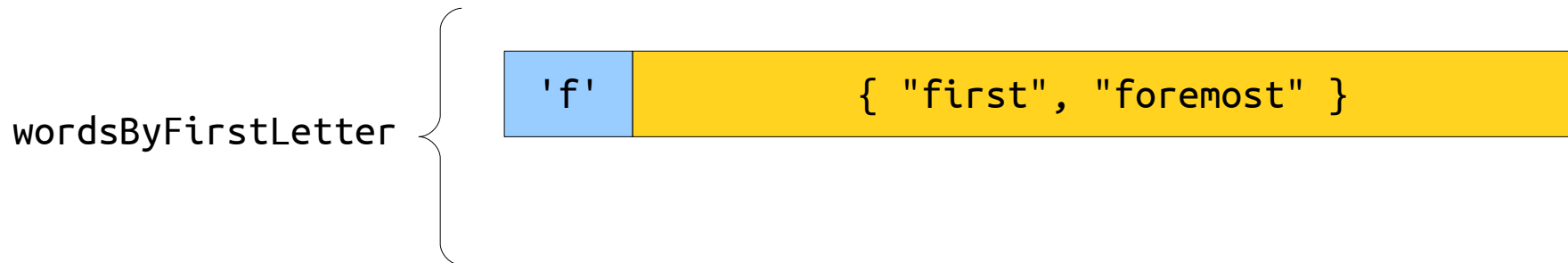
wordsByFirstLetter

'f'

{ "first", "foremost" }

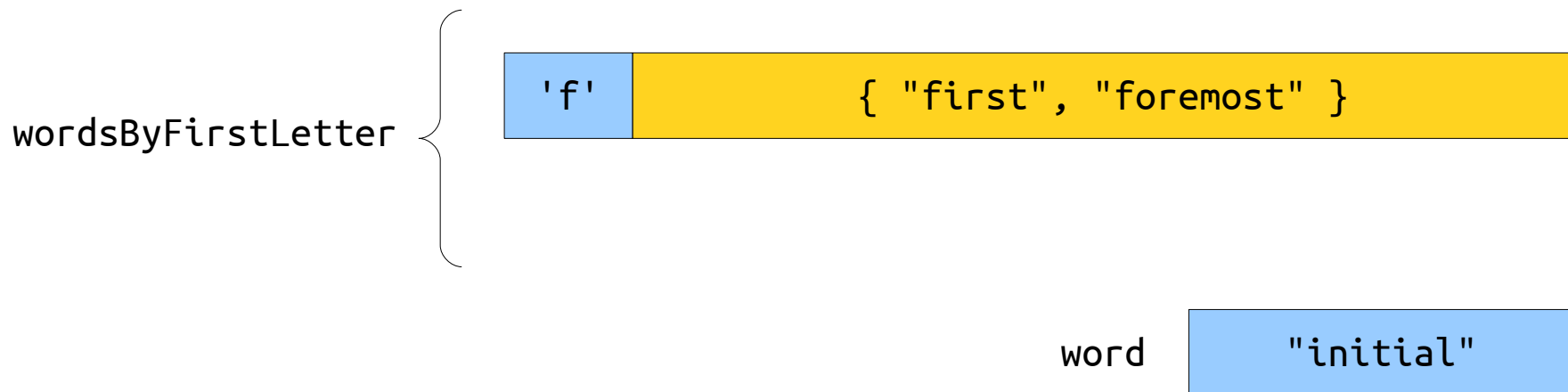
Map Autoinsertion

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



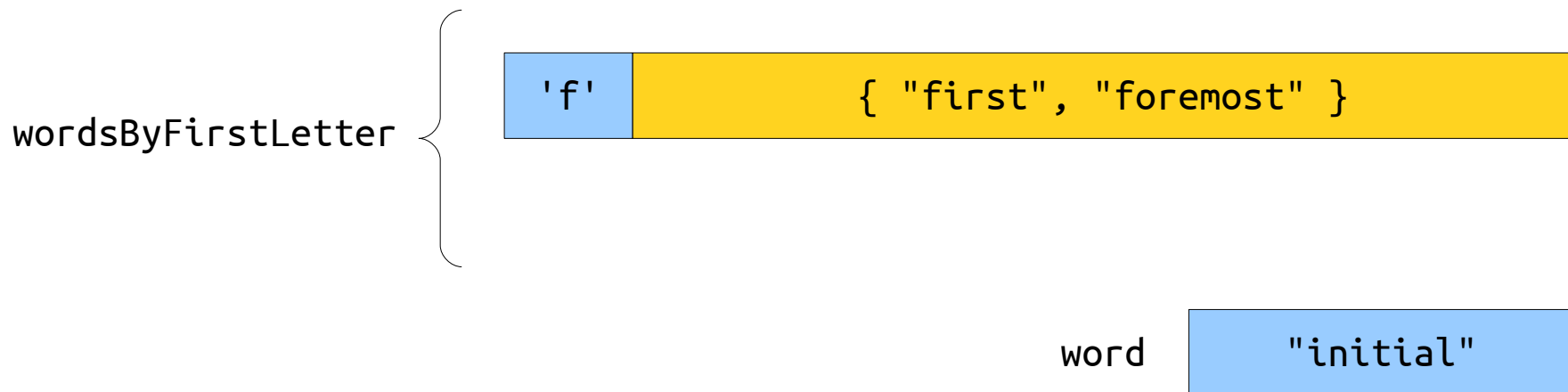
Map Autoinsertion

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



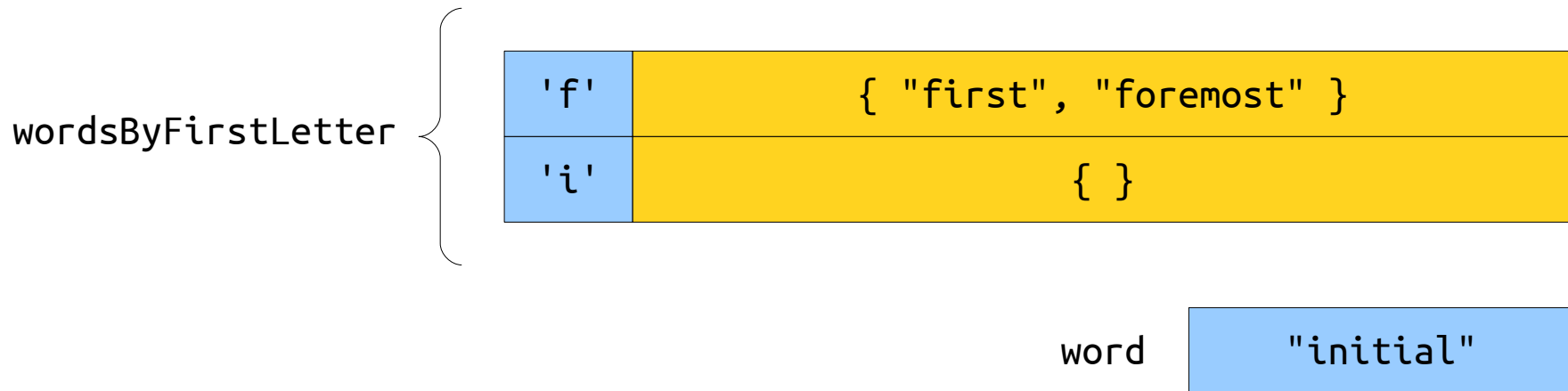
Map Autoinsertion

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



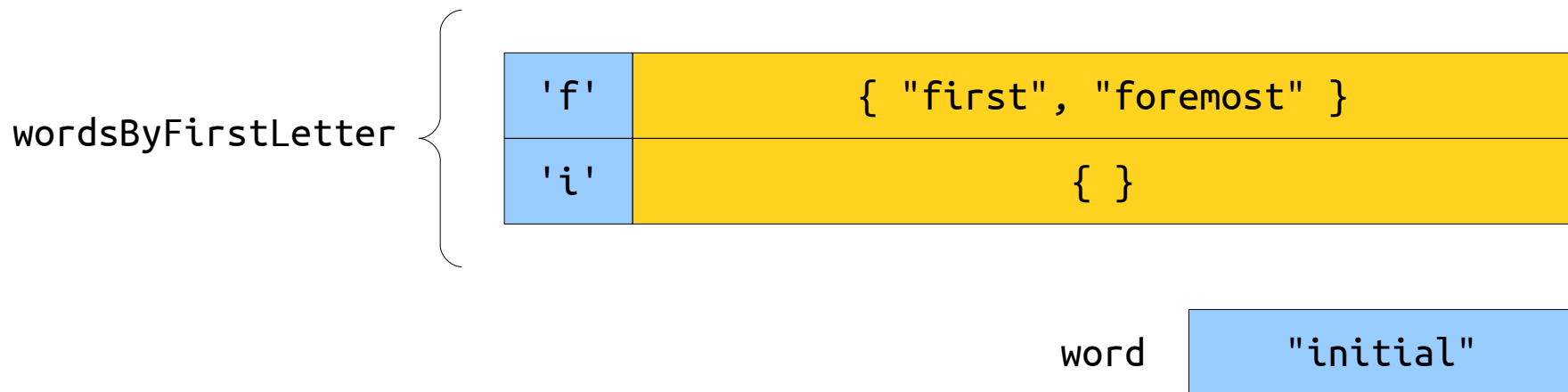
Map Autoinsertion

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



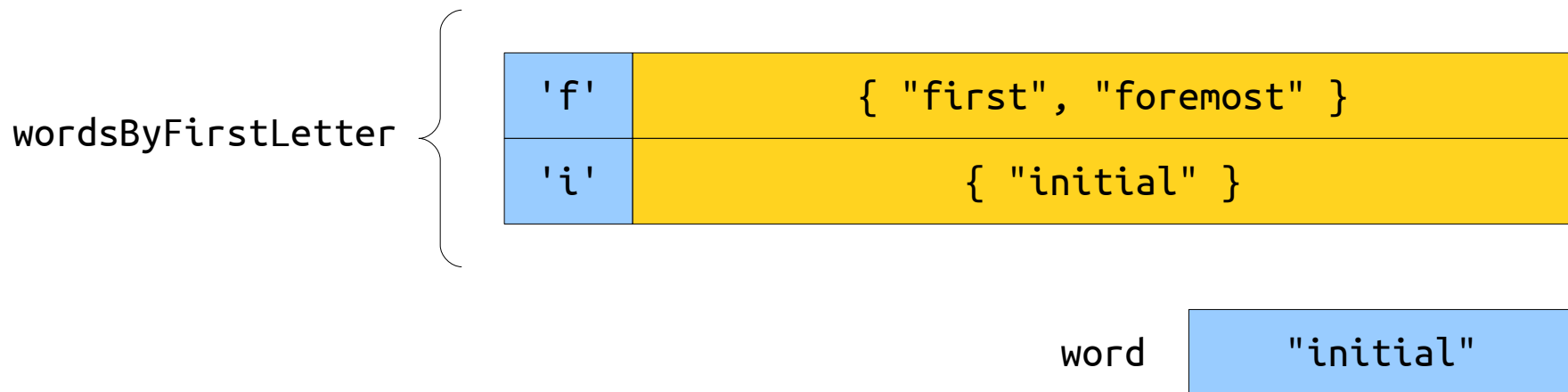
Map Autoinsertion

```
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;  
}
```



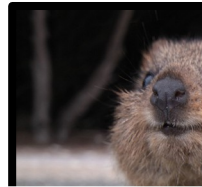
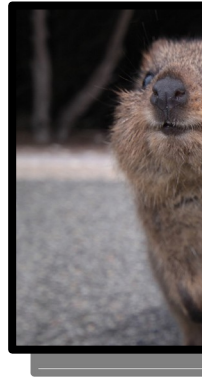
Quokka



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.
- ***Read the Style Guide***
 - Coding style is important! We want to be clear with our expectations.
- ***Keep Working on Assignment 1.***
 - If you're following our recommended timetable, you'll have finished Debugger Warmups and Fire at this point and will be working on Only Connect.

Next Time

- ***Stacks and Queues***
 - Specialized containers for specialized sequences.
 - Applications to text analysis and music.