Collections, Part Three
Lexicon
Lexicon

- A **Lexicon** is a container that stores a collection of words.
- No definitions are associated with the words; it is a “lexicon” rather than a “dictionary.”
- Contains operations for
  - Checking whether a word exists.
  - Checking whether a string is a prefix of a given word.
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
Introducing the Dikdik
And a Music Recommendation
Set
Set

- The **Set** represents an unordered collection of distinct elements.
- Elements can be added and removed, and you can check whether or not an element exists.
Set

- The Set represents an unordered collection of distinct elements.
- Elements can be added and removed, and you can check whether or not an element exists.
Set

- The **Set** represents an unordered collection of distinct elements.
- Elements can be added and removed, and you can check whether or not an element exists.
Set

- The **Set** represents an unordered collection of distinct elements.
- Elements can be added and removed, and you can check whether or not an element exists.
Set

- The **Set** represents an unordered collection of distinct elements.
- Elements can be added and removed, and you can check whether or not an element exists.
Operations on Sets

• You can add a value to a set by writing
  \textit{set} += \textit{value};

• You can remove a value from a set by writing
  \textit{set} -= \textit{value};

• You can check if a value exists by writing
  \textit{set}.contains(\textit{value})

• Many more operations are available (union, intersection, difference, subset, etc.), so be sure to check the documentation.
Map
Map

- The **Map** class represents a set of key/value pairs.
- Each key is associated with a unique value.
- Given a key, can look up the associated value.
Map

- The **Map** class represents a set of key/value pairs.
- Each key is associated with a unique value.
- Given a key, can look up the associated value.
Map

- The **Map** class represents a set of key/value pairs.
- Each key is associated with a unique value.
- Given a key, can look up the associated value.

<table>
<thead>
<tr>
<th>CS106B</th>
<th>Awesome!</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dikdik</td>
<td>Cute!</td>
</tr>
</tbody>
</table>
Map

- The **Map** class represents a set of key/value pairs.
- Each key is associated with a unique value.
- Given a key, can look up the associated value.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CS106B</td>
<td>Awesome!</td>
</tr>
<tr>
<td>Dikdik</td>
<td>Cute!</td>
</tr>
<tr>
<td>This Slide</td>
<td>Self Referential</td>
</tr>
</tbody>
</table>
Map

- The **Map** class represents a set of key/value pairs.
- Each key is associated with a unique value.
- Given a key, can look up the associated value.

<table>
<thead>
<tr>
<th>CS106B</th>
<th>Awesome!</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dikdik</td>
<td>Very Cute!</td>
</tr>
<tr>
<td>This Slide</td>
<td>Self Referential</td>
</tr>
</tbody>
</table>
Using the Map

- You can create a map by writing
  
  ```java
  Map<KeyType, ValueType> map;
  ```

- You can add or change a key/value pair by writing
  
  ```java
  map[key] = value;
  ```
  If the key doesn't already exist, it is added.

- You can read the value associated with a key by writing
  
  ```java
  map[key]
  ```
  If the key doesn't exist, it is added and associated with a default value.

- You can check whether a key exists by calling
  
  ```java
  map.containsKey(key)
  ```
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\langle string, \textbf{int}\rangle \textit{freqMap};

\textbf{while (true)} {
    \textit{string text} = getline("Enter some text: ");
    \textit{cout} \ll "Times seen: " \ll \textit{freqMap[text]} \ll \texttt{endl};
    \textit{freqMap[text]}++;
}

Map Autoinsertion

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

freqMap
Map Autoinsertion

```cpp
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

```cpp
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

"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

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

Oh no! I don't know what that is!
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]++;
}

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]++;
}

Phew! Crisis averted!
Map Autoinsertion

```cpp
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

\{ "Hello" | 0 \}

\{ text | "Hello" \}
Map Autoinsertion

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

Cool as a cucumber.

"Hello" 0

text "Hello"
Map Autoinsertion

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

Cool as a cucumber.

C(■■■C)
Map Autoinsertion

Map<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

"Hello" 1
Map Autoinsertion

```cpp
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

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

Map

- "Hello": 1
- "Goodbye": text
Map Autoinsertion

```cpp
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

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

```
freqMap

"Hello" 1

"Goodbye"
```
Map Autoinsertion

```cpp
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 man, 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

<table>
<thead>
<tr>
<th>Text</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hello</td>
<td>1</td>
</tr>
<tr>
<td>Goodbye</td>
<td>0</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Text</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Hello&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;Goodbye&quot;</td>
<td>0</td>
</tr>
</tbody>
</table>
Map Autoinsertion

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

![Map Autoinsertion Diagram]

- freqMap
  - "Hello" 1
  - "Goodbye" 0
- text: "Goodbye"
Map Autoinsertion

Map&lt;string, int&gt; freqMap;
while (true) {
    string text = getline("Enter some text: ");
    cout &lt;&lt; "Times seen: " &lt;&lt; freqMap[text] &lt;&lt; endl;
    freqMap[text]++;
}

Chillin' like a villain.

$\subset(▀¯▀⊂)$
Map Autoinsertion

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

Here is a sample output of the program:

```
Hello          1
Goodbye        1
```

The output shows that the program has encountered the string "Hello" one time and the string "Goodbye" one time.

Chillin' like a villain.
Map Autoinsertion

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

freqMap

<table>
<thead>
<tr>
<th>text</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Hello&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;Goodbye&quot;</td>
<td>1</td>
</tr>
</tbody>
</table>

"Goodbye"
Map Autoinsertion

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

freqMap

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Hello&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;Goodbye&quot;</td>
<td>1</td>
</tr>
</tbody>
</table>
Sorting by First Letters
Lexicon english("EnglishWords.dat");

Map<char, Lexicon> wordsByFirstLetter;
for (string word: english) {
    wordsByFirstLetter[word[0]].add(word);
}
Lexicon english("EnglishWords.dat");

Map<char, Lexicon> wordsByFirstLetter;
for (string word: english) {
    wordsByFirstLetter[word[0]].add(word);
}
Map Autoinsertion

Lexicon english("EnglishWords.dat");

Map<char, Lexicon> wordsByFirstLetter;

for (string word: english) {
    wordsByFirstLetter[word[0]].add(word);
}

wordsByFirstLetter
Lexicon english("EnglishWords.dat");

Map<char, Lexicon> wordsByFirstLetter;

for (string word: english) {
    wordsByFirstLetter[word[0]].add(word);
}

wordsByFirstLetter
Map Autoinsertion

Lexicon english("EnglishWords.dat");

Map<char, Lexicon> wordsByFirstLetter;

for (string word: english) {
    wordsByFirstLetter[word[0]].add(word);
}

wordsByFirstLetter

word "first"
Map Autoinsertion

Lexicon english("EnglishWords.dat");

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

wordsByFirstLetter
Lexicon english("EnglishWords.dat");

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

wordsByFirstLetter

"first"
Lexicon english("EnglishWords.dat");

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

wordsByFirstLetter

Oops, no f's here.
Lexicon english("EnglishWords.dat");

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

Let's insert that key.

wordsByFirstLetter

'f'

word "first"
Lexicon english("EnglishWords.dat");

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

wordsByFirstLetter

I’ll give you a blank Lexicon.
Map Autoinsertion

Lexicon english("EnglishWords.dat");

Map<\texttt{char}, \texttt{Lexicon}> wordsByFirstLetter;
\texttt{for} (string word: english) {
  \texttt{wordsByFirstLetter[\texttt{word[0]}].add(word);}
}

\texttt{wordsByFirstLetter['f'].add("first");}
Lexicon english("EnglishWords.dat");

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

wordsByFirstLetter

'f' { "first" }

word "first"
Map Autoinsertion

Lexicon english("EnglishWords.dat");

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

wordsByFirstLetter

{ 'f': { "first" } }

word "first"
Lexicon english("EnglishWords.dat");

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

wordsByFirstLetter['f'] = 
{ "first" }
Lexicon english("EnglishWords.dat");

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

wordsByFirstLetter

'f'

{ "first" }
Map Autoinsertion

Lexicon english("EnglishWords.dat");

Map<char, Lexicon> wordsByFirstLetter;

for (string word: english) {
    wordsByFirstLetter[word[0]].add(word);
}

wordsByFirstLetter

'f' { "first" }

word "foremost"
Lexicon english("EnglishWords.dat");

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

wordsByFirstLetter
    'f' { "first" }
    word "foremost"
Lexicon english("EnglishWords.dat");

Map<\text{char}, \text{Lexicon}> wordsByFirstLetter;
for (\text{string word: english}) {
    wordsByFirstLetter[word[0]].add(word);
}

wordsByFirstLetter

\{'f\'} { "first" }

word "foremost"
Lexicon english("EnglishWords.dat");

Map<\texttt{char}, \texttt{Lexicon}> \texttt{wordsByFirstLetter};
for (\texttt{string word: english}) {
   \texttt{wordsByFirstLetter}[\texttt{word[0]]}.\texttt{add(word)};
}

\texttt{wordsByFirstLetter}

\{ \texttt{"f"} \}

\{ \texttt{"first"} \}

\texttt{word}

\texttt{"foremost"}

\texttt{Easy peasy.}

\texttt{⊂ (▀¯▀⊂)}
Map Autoinsertion

Lexicon english("EnglishWords.dat");

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

wordsByFirstLetter

    { "first" }

    'f'

word

"foremost"
Map <char, Lexicon> wordsByFirstLetter;
for (string word: english) {
    wordsByFirstLetter[word[0]].add(word);
}

Lexicon english("EnglishWords.dat");

wordsByFirstLetter

'f' { "first", "foremost" }

word "foremost"
Lexicon english("EnglishWords.dat");

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

wordsByFirstLetter

'm' { "first", "foremost" }

word "foremost"
Lexicon english("EnglishWords.dat");

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

wordsByFirstLetter['f'] = { "first", "foremost" }
Lexicon english("EnglishWords.dat");

Map<char, Lexicon> wordsByFirstLetter;

for (string word : english) {
    wordsByFirstLetter[word[0]].add(word);
}

wordsByFirstLetter

'f' { "first", "foremost" }
Map Autoinsertion

Lexicon english("EnglishWords.dat");

Map<char, Lexicon> wordsByFirstLetter;

for (string word: english) {
    wordsByFirstLetter[word[0]].add(word);
}

wordsByFirstLetter

'f'

{ "first", "foremost" }

word

"initial"
Lexicon english("EnglishWords.dat");

Map<\text{char}, \text{Lexicon}> \text{wordsByFirstLetter};
for (\text{string word: english}) {
    \text{wordsByFirstLetter}[\text{word[0]}].add(\text{word});
}

\text{wordsByFirstLetter[\text{f}]} = \{ \text{"first", "foremost"} \}

\text{word} = \text{"initial"}
Lexicon english("EnglishWords.dat");

Map<\texttt{char}, \texttt{Lexicon}> wordsByFirstLetter;
for (string word: english) {
    wordsByFirstLetter[word[0]].add(word);
}

wordsByFirstLetter
\left\{
    \begin{array}{|c|c|}
        \hline
        'f' & \{ "first", "foremost" \} \\
        \hline
        'i' & \{ \} \\
        \hline
    \end{array}
\right\}

word \quad \text{"initial"}
Lexicon english("EnglishWords.dat");

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

wordsByFirstLetter

<table>
<thead>
<tr>
<th>'f'</th>
<th>{ &quot;first&quot;, &quot;foremost&quot; }</th>
</tr>
</thead>
<tbody>
<tr>
<td>'i'</td>
<td>{ }</td>
</tr>
</tbody>
</table>

word "initial"
Lexicon english("EnglishWords.dat");

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

wordsByFirstLetter

<table>
<thead>
<tr>
<th>'f'</th>
<th>{ &quot;first&quot;, &quot;foremost&quot; }</th>
</tr>
</thead>
<tbody>
<tr>
<td>'i'</td>
<td>{ &quot;initial&quot; }</td>
</tr>
</tbody>
</table>

word "initial"
Anagrams

- Two words are **anagrams** of one another if the letters in one can be rearranged into the other.

- Some examples:
  - “Senator” and “treason.”
  - “Praising” and “aspiring.”
  - “Arrogant” and “tarragon.”

- **Question for you:** does this concept exist in other languages? If so, please send me examples!
Anagrams

- **Nifty fact:** two words are anagrams if you get the same string when you write the letters in those words in sorted order.
- For example, “praising” and “aspiring” are anagrams because, in both cases, you get the string “aiignprs” if you sort the letters.
Anagram Clusters

• Let’s group all words in English into “clusters” of words that are all anagrams of one another.

• We’ll use a Map<string, Lexicon>.
  • Each key is a string of letters in sorted order.
  • Each value is the collection of English words that have those letters in that order.
Assignment 2 Demo
Assignment 2

• Assignment 2 (Word Play) goes out today. It’s due a week from today at the start of class.
  • Play around with properties of words and discover some new things along the way!
  • Solidify your understanding of container types and procedural decomposition.

• **Start this one early.** You’ll want to have some time to let things percolate and to ask for help when you need it.

• **You must complete this assignment individually.** Working in pairs is not permitted yet.
Assignment 2

- Our illustrious and industrious head TA Anton will be holding an assignment review session (YEAH Hours) tonight from 7PM in room 420-041.

- Highly recommended!
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?
Extra Content: How to Sort a String
Counting Sort
Counting Sort

banana
Counting Sort

banana

letterFreq
Counting Sort

```java
for (char ch: input) {
    letterFreq[ch]++;
}
```
Counting Sort

```java
for (char ch: input) {
    letterFreq[ch]++;
}
```
Counting Sort

```java
for (char ch: input) {
    letterFreq[ch]++;
}
```
Counting Sort

```
for (char ch: input) {
    letterFreq[ch]++;  
}
```
Counting Sort

for (char ch: input) {
    letterFreq[ch]++;
}

letterFreq

banana

<table>
<thead>
<tr>
<th>a</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>1</td>
</tr>
</tbody>
</table>
Counting Sort

for (char ch: input) {
    letterFreq[ch]++;
}
Counting Sort

for (char ch: input) {
    letterFreq[ch]++;
}

letterFreq

banana

<table>
<thead>
<tr>
<th>a</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>n</td>
<td>1</td>
</tr>
</tbody>
</table>
Counting Sort

```
for (char ch: input) {
    letterFreq[ch]++;
}
```
Counting Sort

for (char ch: input) {
    letterFreq[ch]++;
}
Counting Sort

for (char ch: input) {
    letterFreq[ch]++;
}

letterFreq

banana

a 2
b 1
n 1
Counting Sort

for (char ch: input) {
    letterFreq[ch]++;
}

letterFreq

banana
Counting Sort

```java
for (char ch: input) {
    letterFreq[ch]++;
}
```
Counting Sort

for (char ch: input) {
    letterFreq[ch]++;
}
Counting Sort

banana

letterFreq

<table>
<thead>
<tr>
<th>a</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>n</td>
<td>2</td>
</tr>
</tbody>
</table>
Order in Range-Based `for` Loops

• When using the range-based `for` loop to iterate over a collection:
  • In a `Vector`, `string`, or array, the elements are retrieved in order.
  • In a `Map`, the `keys` are returned in sorted order.
  • In a `Set` or `Lexicon`, the values are returned in sorted order.
Counting Sort

banana

letterFreq

for (char ch: letterFreq) {
    for (int i = 0; i < letterFreq[ch]; i++) {
        result += ch;
    }
}
Counting Sort

```java
for (char ch: letterFreq) {
    for (int i = 0; i < letterFreq[ch]; i++) {
        result += ch;
    }
}
```
Counting Sort

```java
for (char ch: letterFreq) {
    for (int i = 0; i < letterFreq[ch]; i++) {
        result += ch;
    }
}
```
Counting Sort

banana

letterFreq

\[
\begin{array}{|c|c|}
\hline
\text{a} & 3 \\
\hline
\text{b} & 1 \\
\hline
\text{n} & 2 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{a} & \\
\hline
\text{a} & \\
\hline
\text{a} & \\
\hline
\end{array}
\]

\[
\text{for (char} \ ch: \ \text{letterFreq}) \ { \\
\text{\quad \ for (int} \ i = 0; \ i < \ \text{letterFreq[ch]}; \ i++) \ { \\
\text{\quad \quad \ result} \ += \ ch; \\
\text{\quad \}} \\
\text{\}}
\]
Counting Sort

```java
for (char ch : letterFreq) {
    for (int i = 0; i < letterFreq[ch]; i++) {
        result += ch;
    }
}
```

letterFreq

a 3
b 1
n 2

banana

a a a a b
Counting Sort

for (char ch: letterFreq) {
    for (int i = 0; i < letterFreq[ch]; i++) {
        result += ch;
    }
}

letterFreq

banana

| a | 3 |
| b | 1 |
| n | 2 |

result = a a a a b b
Counting Sort

For (char ch: letterFreq) {
    for (int i = 0; i < letterFreq[ch]; i++) {
        result += ch;
    }
}
Counting Sort

banana

letterFreq

<table>
<thead>
<tr>
<th>a</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>n</td>
<td>2</td>
</tr>
</tbody>
</table>

aaabnnn