CSE 143
Lecture 14 (B)

Maps and Grammars

reading: 11.3

slides created by Marty Stepp
http://www.cs.washington.edu/143/
Exercise

• Write a program to count the occurrences of each word in a large text file (e.g. *Moby Dick* or the King James Bible).
  
  – Allow the user to type a word and report how many times that word appeared in the book.

  – Report all words that appeared in the book at least 500 times, in alphabetical order.

• How will we store the data to solve this problem?
The Map ADT

- **map**: Holds a set of unique *keys* and a collection of *values*, where each key is associated with one value.
  - a.k.a. "dictionary", "associative array", "hash"

- basic map operations:
  - **put**(key, value): Adds a mapping from a key to a value.
  - **get**(key): Retrieves the value mapped to the key.
  - **remove**(key): Removes the given key and its mapped value.

```
myMap.get("Juliet") returns "Capulet"
```
Maps and tallying

• a map can be thought of as generalization of a tallying array
  – the "index" (key) doesn't have to be an `int`

• recall previous tallying examples from CSE 142
  – count digits: `22092310907`
    - index: 0 1 2 3 4 5 6 7 8 9
    - value: 3 1 3 0 0 0 0 0 1 0 2

    // (M)cCain, (O)bama, (I)ndependent
  – count votes: "MOOOOOOMMMMMOOOOOOOOMMMIMOMMIMOMMOO"
• in Java, maps are represented by `Map` interface in `java.util`

• `Map` is implemented by the `HashMap` and `TreeMap` classes
  
  – `HashMap`: implemented using an array called a "hash table"; extremely fast: \( O(1) \); keys are stored in unpredictable order
  
  – `TreeMap`: implemented as a linked "binary tree" structure; very fast: \( O(\log N) \); keys are stored in sorted order

  – A map requires 2 type parameters: one for keys, one for values.

```java
// maps from String keys to Integer values
Map<String, Integer> votes = new HashMap<String, Integer>();
```
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>put(key, value)</td>
<td>adds a mapping from the given key to the given value; if the key already exists, replaces its value with the given one</td>
</tr>
<tr>
<td>get(key)</td>
<td>returns the value mapped to the given key (null if not found)</td>
</tr>
<tr>
<td>containsKey(key)</td>
<td>returns true if the map contains a mapping for the given key</td>
</tr>
<tr>
<td>remove(key)</td>
<td>removes any existing mapping for the given key</td>
</tr>
<tr>
<td>clear()</td>
<td>removes all key/value pairs from the map</td>
</tr>
<tr>
<td>size()</td>
<td>returns the number of key/value pairs in the map</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>returns true if the map's size is 0</td>
</tr>
<tr>
<td>toString()</td>
<td>returns a string such as &quot;{a=90, d=60, c=70}&quot;</td>
</tr>
<tr>
<td>keySet()</td>
<td>returns a set of all keys in the map</td>
</tr>
<tr>
<td>values()</td>
<td>returns a collection of all values in the map</td>
</tr>
<tr>
<td>putAll(map)</td>
<td>adds all key/value pairs from the given map to this map</td>
</tr>
<tr>
<td>equals(map)</td>
<td>returns true if given map has the same mappings as this one</td>
</tr>
</tbody>
</table>
Using maps

- A map allows you to get from one half of a pair to the other.
  - Remembers one piece of information about every index (key).

```java
// key   value
put("Marty", "206-685-2181")
```

- Later, we can supply only the key and get back the related value:
  Allows us to ask: *What is Marty's phone number?*

```java
get("Marty")
```

```
"206-685-2181"
```
// read file into a map of [word --> number of occurrences]
Map<String, Integer> wordCount = new HashMap<String, Integer>();
Scanner input = new Scanner(new File("mobydick.txt"));
while (input.hasNext()) {
    String word = input.next();
    if (wordCount.containsKey(word)) {
        // seen this word before; increase count by 1
        int count = wordCount.get(word);
        wordCount.put(word, count + 1);
    } else {
        // never seen this word before
        wordCount.put(word, 1);
    }
}

Scanner console = new Scanner(System.in);
System.out.print("Word to search for? ");
String word = console.next();
System.out.println("appears " + wordCount.get(word) + " times.");
keySet and values

- **keySet** method returns a set of all keys in the map
  - can loop over the keys in a foreach loop
  - can get each key's associated value by calling **get** on the map

```java
Map<String, Integer> ages = new HashMap<String, Integer>();
ages.put("Marty", 19);
ages.put("Geneva", 2);
ages.put("Vicki", 57);
for (String name : ages.keySet()) {
    int age = ages.get(name); // Geneva -> 2
    System.out.println(name + " -> " + age); // Marty -> 19
    System.out.println(name + " -> " + age); // Vicki -> 57
}
```

- **values** method returns a collection of all values in the map
  - can loop over the values in a foreach loop
  - there is no easy way to get from a value to its associated key(s)
Languages and Grammars
• (formal) **language**: A set of words or symbols.

• **grammar**: A description of a language that describes which sequences of symbols are allowed in that language.
  – describes language *syntax* (rules) but not *semantics* (meaning)
  – can be used to generate strings from a language, or to determine whether a given string belongs to a given language
Backus-Naur (BNF)

- **Backus-Naur Form (BNF):** A syntax for describing language grammars in terms of transformation *rules*, of the form:

  \[
  \text{<symbol>} \ ::= \text{<expression>} \mid \text{<expression>} \ldots \mid \text{<expression>}
  \]

  - **terminal:** A fundamental symbol of the language.
  - **non-terminal:** A high-level symbol describing language syntax, which can be transformed into other non-terminal or terminal symbol(s) based on the rules of the grammar.

- developed by two Turing-award-winning computer scientists in 1960 to describe their new ALGOL programming language
An example BNF grammar

<s>::=<n> <v>
<n>::=Marty | Victoria | Stuart | Jessica
<v>::=cried | slept | belched

• Some sentences that could be generated from this grammar:

Marty slept
Jessica belched
Stuart cried
BNF grammar version 2

<s>::=<np> <v>

<np>::=<pn> | <dp> <n>

<pn>::=Marty | Victoria | Stuart | Jessica
<dp>::=a | the

<n>::=ball | hamster | carrot | computer
<v>::=cried | slept | belched

• Some sentences that could be generated from this grammar:
  
  the carrot cried
  Jessica belched
  a computer slept
BNF grammar version 3

\[ <s> ::= <np> <v> \]
\[ <np> ::= <pn> | <dp> <adj> <n> \]
\[ <pn> ::= \text{Marty} | \text{Victoria} | \text{Stuart} | \text{Jessica} \]
\[ <dp> ::= \text{a} | \text{the} \]
\[ <adj> ::= \text{silly} | \text{invisible} | \text{loud} | \text{romantic} \]
\[ <n> ::= \text{ball} | \text{hamster} | \text{carrot} | \text{computer} \]
\[ <v> ::= \text{cried} | \text{slept} | \text{belched} \]

- Some sentences that could be generated from this grammar:
  - the invisible carrot cried
  - Jessica belched
  - a computer slept
  - a romantic ball belched
• Grammar rules can be defined recursively, so that the expansion of a symbol can contain that same symbol.
  – There must also be expressions that expand the symbol into something non-recursive, so that the recursion eventually ends.
Could this grammar generate the following sentences?
Fred honored the green wonderful child
big Jane wept the fat man fat

Generate a random sentence using this grammar.
Sentence generation

<s>
  <np>
    <pn> Fred 
    <vp>
      <tv>
        honored
      <np>
        <dp>
          the
        <adjp>
          <adj> green
          <adjp>
            <adj> wonderful
        <n> child
    
  
</np>
</s>