Performance client. `FrequencyCounter` (on the next page) is a symbol-table client that finds the number of occurrences of each string (having at least as many characters as a given threshold length) in a sequence of strings from standard input, then iterates through the keys to find the one that occurs the most frequently. This client is an example of a *dictionary* client, an application that we discuss in more detail in SECTION 3.5. This client answers a simple question: Which word (no shorter than a given length) occurs most frequently in a given text? Throughout this chapter, we measure performance of this client with three reference inputs: the first five lines of C. Dickens's *Tale of Two Cities* (`tinyTale.txt`), the full text of the book (`tale.txt`), and a popular database of 1 million sentences taken at random from the web that is known as the *Leipzig Corpora Collection* (`leipzig1M.txt`). For example, here is the content of `tinyTale.txt`:

```
% more tinyTale.txt
it was the best of times it was the worst of times
it was the age of wisdom it was the age of foolishness
it was the epoch of belief it was the epoch of incredulity
it was the season of light it was the season of darkness
it was the spring of hope it was the winter of despair
```

Small test input

This text has 60 words taken from 20 distinct words, four of which occur ten times (the highest frequency). Given this input, `FrequencyCounter` will print out any of `it`, `was`, `the`, or `of` (the one chosen may vary, depending upon characteristics of the symbol-table implementation), followed by the frequency, 10.

To study performance for the larger inputs, it is clear that two measures are of interest: Each word in the input is used as a search key once, so the total number of words in the text is certainly relevant. Second, each *distinct* word in the input is put into the

	`tinyTale.txt`		`tale.txt`		`leipzig1M.txt`	
	words	**distinct**	**words**	**distinct**	**words**	**distinct**
all words	60	20	135,635	10,679	21,191,455	534,580
at least 8 letters	3	3	14,350	5,737	4,239,597	299,593
at least 10 letters	2	2	4,582	2,260	1,610,829	165,555

Characteristics of larger test input streams

A symbol-table client

```
public class FrequencyCounter
{
   public static void main(String[] args)
   {
      int minlen = Integer.parseInt(args[0]);   // key-length cutoff
      ST<String, Integer> st = new ST<String, Integer>();
      while (!StdIn.isEmpty())
      {  // Build symbol table and count frequencies.
         String word = StdIn.readString();
         if (word.length() < minlen) continue;  // Ignore short keys.
         if (!st.contains(word)) st.put(word, 1);
         else                    st.put(word, st.get(word) + 1);
      }
      // Find a key with the highest frequency count.
      String max = "";
      st.put(max, 0);
      for (String word : st.keys())
         if (st.get(word) > st.get(max))
            max = word;
      StdOut.println(max + " " + st.get(max));
   }
}
```

This ST client counts the frequency of occurrence of the strings in standard input, then prints out one that occurs with highest frequency. The command-line argument specifies a lower bound on the length of keys considered.

```
% java FrequencyCounter 1 < tinyTale.txt
it 10

% java FrequencyCounter 8 < tale.txt
business 122

% java FrequencyCounter 10 < leipzig1M.txt
government 24763
```

symbol table (and the total number of distinct words in the input gives the size of the table after all keys have been inserted), so the total number of distinct words in the input stream is certainly relevant. We need to know both of these quantities in order to estimate the running time of `FrequencyCounter` (for a start, see EXERCISE 3.1.6). We defer details until we consider some algorithms, but you should have in mind a general idea of the needs of typical applications like this one. For example, running `FrequencyCounter` on `leipzig1M.txt` for words of length 8 or more involves millions of searches in a table with hundreds of thousands of keys and values. A server on the web might need to handle billions of transactions on tables with millions of keys and values.

THE BASIC QUESTION THAT THIS CLIENT and these examples raise is the following: Can we develop a symbol-table implementation that can handle a huge number of `get()` operations on a large table, which itself was built with a large number of intermixed `get()` and `put()` operations? If you are only doing a few searches, any implementation will do, but you cannot make use of a client like `FrequencyCounter` for large problems without a good symbol-table implementation. `FrequencyCounter` is surrogate for a very common situation. Specifically, it has the following characteristics, which are shared by many other symbol-table clients:

- Search and insert operations are intermixed.
- The number of distinct keys is not small.
- Substantially more searches than inserts are likely.
- Search and insert patterns, though unpredictable, are not random.

Our goal is to develop symbol-table implementations that make it feasible to use such clients to solve typical practical problems.

Next, we consider two elementary implementations and their performance for `FrequencyCounter`. Then, in the next several sections, you will learn classic implementations that can achieve excellent performance for such clients, even for huge input streams and tables.