| Imperial College of Science, Technology and Medicine | University of London |
|---|---|
| Computer Science (CS) / Software Engineering (SE) | BEng and MEng<br>Examinations Part I |
| Department of Computing | Integrated Laboratory Course |

Laboratory work is a continuously assessed part of the examinations
and is a required part of the degree assessment.
Laboratory work must be handed in for marking by the due date.
Late submissions may not be marked.

| | |
|---|---|
| Exercise: 17 | Working: Individual |
| Title: Spell-checking using a Hash Table | |
| Issue date: 12th March 2007 | Due date: 19th March 2007 |
| System: Linux | Language: Java |

## Aims

- To gain further experience with **abstract data types** (ADTs) and **references** ("pointers") in Java.
- To implement a simple **hash table** and a spell-checking program that uses it.

## The Problem

- A **spell-checker** program checks the words in a piece of text against a dictionary of "definitively correct" spellings, and flags for human attention those not found in the dictionary.

  A full-blown spell-checker would typically handle text with arbitrary punctuation, layout and so on, and would nowadays flag the problem words by rendering them in a different colour or something like that. Your task is to write a cut-down version of such a program: a plain text spell-checker which receives a piece of text consisting purely of the words themselves, and not any punctuation, and which echoes the words of the text on to the standard output, with problem words flagged like this: `>>>word<<<`
- You are provided with the interface class file **HashTableInterface.java** which offers a **hash table** - a concept explained below. In particular it offers the facility to add words to the table (these will be the dictionary words in our spell-checker application) and to check if a word is present (these will be the words in the piece of text under scrutiny, after the dictionary words have all been added).
- You should write three Java code files:

  - a main program file **SpellCheck.java** which performs spell-checking, using the hash table to store and look up the dictionary words, and ends by giving a brief statistical summary of the performance of the hash table;
  - an implementation class file **HashTable.java** implementing the hash table interface provided.
  - a further implementation class file **HashTableNode.java** providing the data stucture used by the hash table implementation to store the words input.

- The access procedures advertised by the interface are as follows.

```
public void addWord (String word);
/*
 * Adds the given word to the hash table in an appropriate location,
 * as determined by a hashing function.
 * To be precise, it's the word converted to all lower case that's added,
 * to make usage of the hash table case-insensitive.
 */

public boolean isWordPresent (String word);
/*
 * Returns whether the given word is present in the hash table.
 * Again, it's actually the word converted to all lower case
 * that's checked, to make usage of the hash table case-insensitive.
 */

public int howManyUsages();
/*
 * Returns the total number of calls to addWord() or isWordPresent()
 * made so far in the lifetime of the hash table.
 */

public int howManyInternalNodeOperations();
/*
 * Returns the total number of internal node operations
 * (inspection of an existing node or creation of a new one)
 * performed so far in the lifetime of the hash table.
 */
```

  The first two have the obvious purpose in our spell-checker application. The other two are for gauging the performance of the hash table as explained below, and are the ones to use when giving the summary statistical report at the end.

### What is a hash table?

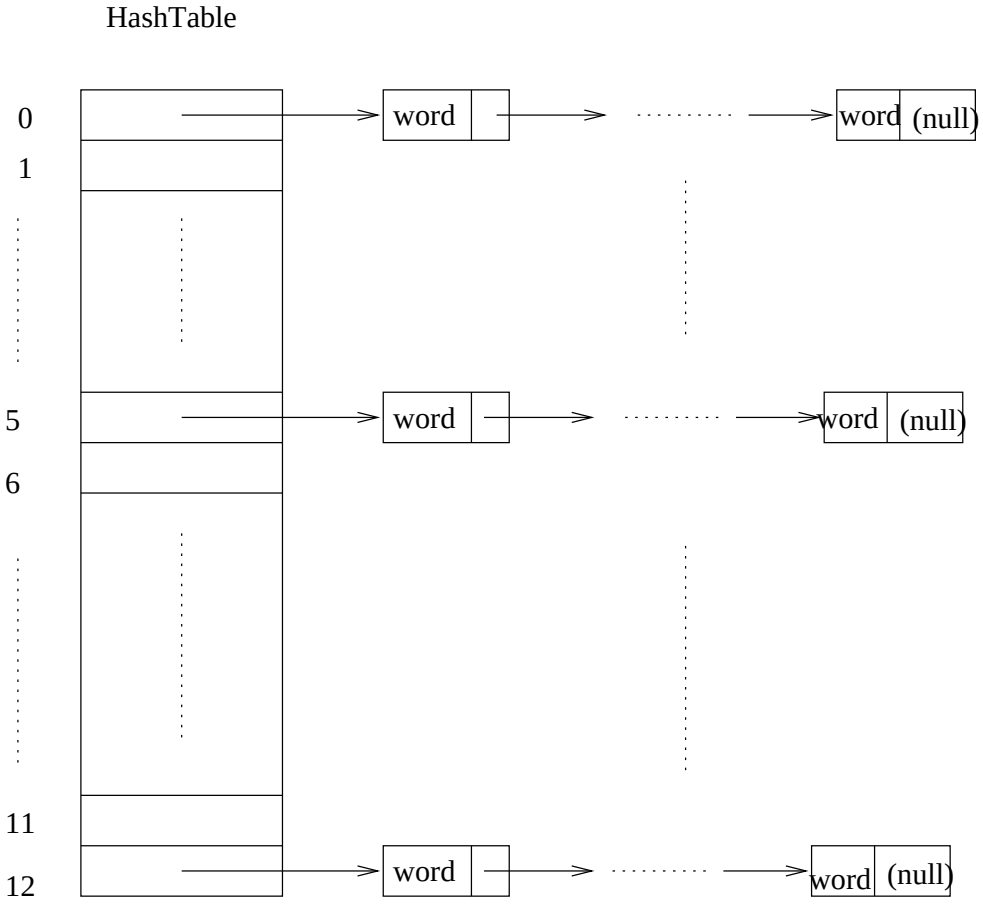
- There are many different types of hash table discussed in the computing literature. They all revolve around the concept of computing a so-called "hash value" of a piece of

data to be stored, giving a provisional indication of where it is to be put. For example, it could indicate a hoped-for location to store the item in an array.

By itself, the hash value mechanism is not robust: some other, earlier data item could have happened to have the same hash value, and thus be already sitting in the sought-after location. Therefore, we need some way of treating such occurrences (called "collisions" in the literature).

- One simple way of dealing with a collision would be to put the freshly arriving data item in the *next* still-free slot in the array (travelling cyclically around the array as if the end is joined on to the beginning). However, as the array fills up it takes longer and longer to find such free slots, and indeed eventually there may be none.

  In this exercise your task is to implement a more sophisticated approach where the array is not an array of data items (words) themselves, but rather of **dynamic** data structures in their own right, each holding an arbitrary number of words. In the illustration and discussion below we take these to be **linked lists** although you may want to try your hand at some other dynamic data structure such as binary trees. (Such alternative data structures may even be more efficient and gain you some extra marks in the matter of how few internal operations your hash table had to perform!)




The merit of the dynamic data structure approach is that there will not be an array overflow as more and more words pour in; but even in cases when there are only a

few words to be stored, they will likely get (or mostly get) an array element each to themselves, so that they'll each sit in a little single-node list (or tree or whatever), giving superb performance on later lookup.

- As explained below, we ask that your hash table array be of size **exactly 13** - no more, no less! - to give a "level playing field" for comparing how well people do in keeping the internal operations as few as possible.

## Submit by Monday 19th March 2007

## What To Do

- Copy the interface class file **HashTableInterface.java** and the example dictionary and prose text files **dict1.dat**, **prose1.txt**, **dict2.dat** and **prose2.txt** with the command **exercise 17**.

- Your program should be written so that **HashTableInterface.java**, **HashTable.java** and **HashTableNode.java** are part of the **hashtable** package. You will need to create the appropriate directory structure to ensure this. Your main program **SpellCheck.java** is not part of the package but should import the classes from the package that it needs.

- Write the main program file **SpellCheck.java** and the implementation class file **HashTable.java** and its data structure class file **HashTableNode.java**.

- Your main program should first read in a dictionary of correctly spelled words, specified as a filename in the first and only argument to the program (`args[0]`), and store them in the hash table. You may assume the dictionary file to contain one word to a line, with no whitespace or blank lines and no repetition. (The **FileReader** and **BufferedReader** classes in **java.io.*** are the most suitable for performing this phase).

- To read from a file you will need to create a first a **FileReader** and then a **BufferedReader**.

```
FileReader dictfilereader = new FileReader(args[0]);
BufferedReader dictbufferedreader = new BufferedReader(
                                        dictfilereader);
```

  You can the use methods such as `readLine()` to input the dictionary. Note that this method returns `null` when `EOF` is reached. When you read in the file you should `catch` exceptions such as `FileNotFoundException`, `IOException` and print an error message before terminating the program. You can do this by using the `try .. catch` construct provided by Java.

```
try {
    // CODE THAT MAY THROW AN EXCEPTION
    // such as opening a FileReader
    // or reading from a file
}
```

```
catch { (PARTICULAR-TYPE-OF-EXCEPTION e)
    // Code for dealing with exception e goes here
}
```

- Your program should then read the prose words from the standard input. Java 1.5 introduces a class **java.util.Scanner** which reads input from a variety of sources including File, String, or InputStream. Since the standard input, **System.in**, is a type of InputStream, you may use this class to retrieve input fed in from the console. The Scanner breaks up input into 'tokens', which can be converted to different types via different methods. e.g.:

  ```
  Scanner sc = new Scanner();
  while(sc.hasNext()){
    String word = sc.next();
    int number = sc.nextInt();
  }
  ```

  Refer to the Java-API for more information on this class. Alternatively, the simple I/O facilities of **uk.ac.imperial.kenya.nio.KenyaNIO**, in particular, **KenyaNIO.get().isEOF()** and **KenyaNIO().get().readString()** can be used to retrieve each word from the standard input.

- You may assume this to be non-empty and with no trailing whitespace at the end, so that the simple I/O facilities Kenya can be used to get each word (string) in the standard input. Words that are in the dictionary should simply be printed as they are. If a word is *not* in the dictionary it should be flagged in this style: `>>>word<<<`, as explained earlier.

  After all of the standard input has been dealt with your program should print a simple statistical summary of the hash table's performance. The summary should report the number of usages of the hash table, the number of internal node operations, and finally the *average* number of node operations per usage (which is simply the latter divided by the former - take care to use real ("double") arithmetic when dividing).

- If you use **get().readString()** to read the prose text words you will inevitably lose all layout information about the original text (its structuring into lines, paragraphs, etc). It is however not desirable for your prose text *output* to be all one very long horizontal line of words, nor for it to be a vertical chain of words, one word per line. Therefore, as a courtesy, you should **start a new line of output after every ten words**. This is of course an arbitrary layout relative to the original text, but it will make the output tolerably readable.

- Your **HashTable.java** and **HashTableNode.java** implementation class files should implement the interface given in **HashTableInterface.java**. A real-life hash table would probably be a quite large array (say a few thousand elements) and would only really come close to filling up if a great deal of data was supplied. To make sure your hash table fills up quickly in an interesting and challenging way even on quite small example test data, **we require that your hash table array be of size 13**. No more, no less!

- Each element of the array should be implemented as a dynamic data structure using references ("pointers"). The simplest choice would be linked lists - the illustration above is of that choice. However, you are welcome to choose an alternative dynamic data structure if you want. You may be able to cut down on the number of internal node operations by doing so. It ought to go without saying that your counting of these operations should be scrupulously honest!

  Note that the hash table itself never appears as a parameter to any of the access procedures; rather, it should be an **encapsulated variable**: that is, a private variable, global *within* your implementation class file **HashTable.java** but invisible outside it.

- Test your program on our supplied data files and on any data files of your own you create. The simplest way to supply both the dictionary and the standard input as files is to use Unix redirection on the standard input, like this:

  ```
  java -ea SpellCheck dictfile < prosefile
  ```

  Your program then receives the dictionary explicitly as a file, but the prose text as standard input as if it had been typed at the keyboard, even though it too is really coming from a file.

- For the hash value of a word, simply adding up its characters as integer ASCII values and taking the result modulo 13 (the specified size of the array) will suffice. You are however welcome to try a more sophisticated hash function if you want. Whatever scheme you use, remember to perform your computation not on the word itself as supplied, but on the word converted to all lower case, to make usage of the hash table case-insensitive. (If `word` is a string then `word.toLowerCase()` gives the string with any upper case converted to lower case.)

### The dynamic data structure for an array element

- The dynamic data structure should be implemented using a recursive class (called **HashTableNode**) containing **references** to any further nodes of the structure. If null, a data structure node indicates that that particular data structure (or part thereof) is currently empty. If non-null, it can store a word and whatever references to further nodes are appropriate - so for a linked list there would be a reference to the next node (which may in turn be null or non-null as appropriate).

- Your implementation class code for the hash table **should be stored in the file HashTable.java**.

- You should initialize every element of your hash table array encapsulated variable to **null** to represent that the hash table starts off empty. Your counts of method usage and node operations should of course be initialized to zero.

- The implementation class code the data stucture used for the words **should be stored in the file HashTableNode.java**.

## Submission

- Submit **SpellCheck.java**, **HashTable.java** and **HashTableNode.java** in the usual way using the **CATE** system.

## Assessment

| | |
|---|---|
| Main program | 2 |
| Hash table, including credit for how few node operations it performs | 3 |
| Design, style, readability | 5 |
| Total | 10 |