Mandatory assignment 1 - INF 2220

Due: 28.09.2007

In this assignment you will be implementing a spell-check application for (a subset of) the Norwegian language. If all words in the Norwegian language were to be present in all their forms, the dictionary would have to contain about one million words, so we select only a subset of these words (approximately 15.000 words). The words to be recognized by the spell-checker will be given to you in a text file, and the first part of this assignment will be to read the file and organize your dictionary.

The words that should be recognized by your spell-checker are located in the file `ordbok1.txt` on the inf2220 web-page. The number of words in this file will not surpass 15.000 and they come in random order (i.e., they are not sorted in advance). The words are separated by newlines and blank spaces, and all the words are given in lowercase letters. Remember to convert user input to lowercase as well.

First we just want to do a simple lookup of the word the user has typed in, if we are unable to find the word in the dictionary, we will be looking for "similar words" to help him out. A more precise definition of "similar words" will be given.

NOTE: In this assignment everyone should follow the same programming pattern, and we will impose some structure that all must follow. Debates could be held about whether or not this is the optimal structure for a spell-checker but part of the reason for it being so is that we want to use central aspects of the curriculum. With a larger dictionary, other structures like B-trees would probably be a more likely candidate than the binary-trees we will be using in this assignment.

Reading the dictionary and storing it

The words should be stored in a binary tree. We can assume that the words are stored randomly in our dictionary file such that no steps have to be taken in order to make our tree balanced (i.e., the first word read is the root of the tree).

After the binary tree has been built, a small change should be made to it; the word 'evneveik' (English: retard) should be removed from the tree, then inserted again. This means that you have to implement a removal scheme, which does not destroy the property that a binary tree should have (it should be searchable).

When we look for 'similar words' we will be looking at a large number of words that do not belong to the Norwegian language, and cannot be found in our subset of it contained in the binary tree. To speed up this process we will create a Boolean array of hash-values that belong to the words in the dictionary. For this to make any sense, the Boolean array has to be a great deal larger than our dictionary (we want most of the entries in the array to contain the value 'false', which means that no dictionary word hashed to this value). This will not become a hash table (HashMap) in the regular sense since we cannot retrieve objects from . We can however easily tell if no key (in this case word in the dictionary) has hashed to some value, which gives us quick negative answers.

The length of the Boolean hash table should be about ten times the length of our dictionary (150.000 entries). If the hash-function gives a perfect spread we will still have 90% of our entries 'empty' or false (worst case scenario). This implies that by performing a computation of the hash-value

of a 'similar word', we can in at least 90% of the cases exclude the word in very little time. A hash-function has to be implemented which gives us a good spread.

When the Boolean hash table and binary tree is implemented, some statistics should be reported on how well their performance became. In particular the following should be reported:

Binary tree:

- The depth of the tree (length of the path to the node furthest away from the root)
- How many nodes are there for each depth of the tree
- The average depth of all the nodes
- Final and first word of the dictionary (the nodes furthest to the right and to the left).

This can be computed while nodes are inserted into the tree, or by traversing the tree afterwards.

Hash table statistics:

- How many different hash-values were created using the words from the dictionary.
- Divide your Boolean hash-table into intervals (of size 30) and report how large the fill rate is for your different intervals. (how many intervals have 0 hits, how many intervals have 1 hit... up to 30)

Using the dictionary

With the term *similar words* to X we mean:

- A word identical to X, except that two letters next to each other have been switched.
- A word identical to X, except one letter has been replaced with another.
- A word identical to X, except one letter is missing.
- A word identical to X, except for a letter not belonging to the word added in front, or at the end, or somewhere in between. (word: fish, similar words: ifish fishi fiish)

To investigate whether or not a word is located in the dictionary, we use a 'lookup' operation. This is done by evaluating a word's hash-value, and then see if that hash-value is present in our Boolean hash. If this is not the case, we know the word is not in the dictionary. We must investigate our binary tree if the word hashes to a value that was 'true' in our Boolean hash table. Remember two different words can hash to the same value without being identical. This means that we have to look in our binary tree once the Boolean hash test has failed to exclude a word.

When a user asks for a spell-check of a word X, the following procedure should be applied: First see if the word is located in the dictionary:

1. In this is case, give a positive answer and prompt the user for another word

2. If this is not the case, generate similar words. If any of the similar words generated is found in the dictionary, print them out as suggestions to the user

For those cases where the original word was not located in the dictionary (and multiple lookups for similar words have to be made), some statistics should be written out.

Statistics

- The number of lookups that gave a positive answer.
- The number of lookups that gave a negative answer, but passed the hash-value test.
- The number of lookups that gave a negative answer, but were stopped by the hash-value test.
- Time used to generate and look for similar words.

If your hash function gives a good spread, the third number of this small statistic should be about nine or ten times larger than the second number.

User interaction

The focus of this assignment should not lie in the making of a nice user interface. The program could just be an endless loop which prompts the user for a word, then tries to locate this word. If none can be found, suggestions base on the 'similar words' are printed. To abort this loop (quit the program) 'q' could be typed, which is not a valid Norwegian word anyway. A small menu should be printed out before the program goes into its endless loop of spell-checking. Exceptions should be handled in a reasonable fashion (for example, if a word consist of nothing but numbers, prompt the user for something which at least is made up of letters).

A few suggestions

When the program starts, something along these lines should be done:

- Read one line at a time from the file 'ordbox.txt', split each line into the different words
- Insert the words into the binary tree and hash table
- Print out some statistics about them (hash table, binary tree)
- Print a small menu ('q' to quit and so on...)
- Enter an endless loop which reads user input
- Find out whether a word in the dictionary hashes to the same value as the user-input-word
  - if so, try to find that word in the binary tree, and if so it's correct
  - if not, the word is wrong and we must generate similar words and test them the same way, and then print out suggestions if we found any.

Hint
Conversion between character arrays and strings is very simple in Java, to convert a string to a character array:

```
char[] alphabet = "abcdefghijklmnopqrstuvwxyzæøå".toCharArray();
```

and back again by initializing a new String object:

```
String str_alphabet = toCharArray(alphabet);
```

This will come in handy during generation of similar words.

A good strategy is to implement functions that generate different types of similar words, then return these so you can investigate whether or not they are present in your dictionary. The combination of the two functions below, compute all words that have two letters next to each other switched (i.e., they will generate all similar words of type 1 from the definition).

```
public String[] similarOne(String word){

    char[] word_array = word.toCharArray();
    char[] tmp = word_array.clone();

    String[] words = new String[word_array.length-1];

    for(int i = 0; i < word_array.length - 1; i++){
     words[i] = swap(i, i+1, tmp);
     tmp = word_array.clone();
    }

    return words;
}

public String swap(int a, int b, char[] word){
    char tmp = word[a];
    word[a] = word[b];
    word[b] = tmp;

    return new String(word);
}
```

Notice the use of the clone() function here, since arrays are passed by reference the original array is destroyed during the swap function, and we get a fresh copy of the original word by calling clone() on the original. It can also be a bit difficult to see in advance how long the String array should be with this approach, or how many similar words that will be created by such a swapping algorithm. In this case we are going to switch characters which are next to each other, and this can only be done in n-1 ways if the word has n characters (we actually count spaces between the characters, each of these give us two elements that can be swapped). If it is hard to compute the number of

suggested words that will be generated in advance, more dynamic structures can be used, although that will naturally cost you time.

In this assignment you will have to construct your own hash function, which generates an integer based on a given input string. You can look at the documentation for java.lang.String to see how the function hashCode() computes its hash value as an example. Remember that we can easily find characters' numeric value by a cast from char to int.

```
char letter = 'p';
int numeric_val = (int) letter;

System.out.println(numeric_val); // 112 is printed
```

Handing in your assignment

The assignment should be carried out individually and delivered to the teaching assistant responsible for your group.

When you submit your assignment it should contain the following:

- Source code.
- Print-out of execution (on file utskrift.txt) which shows:
  - The different statistics
  - Spell-check of these words:
    - etterfølger
    - eterfølger
    - etterfolger
    - etterfølgern
    - tterfølger

To get your assignment evaluated, it has to compile and run in a terminal-window on the department's Unix/Linux machines which means that it must have a text based interface. (You cannot only have a GUI. You can make a GUI also naturally, but it must also be possible to start a text based version of the program from the command line). The source code should be documented, and self explanatory where comments are left out.

The assignment should be emailed to the teaching assistant assigned to your group. You should send it as a tar'ed and zipped archive where the top-level directory is your username.

So if your username is 'andrevondrei':

```
> mkdir andrevondrei
> mv YourProgram.java utskrift.txt README.txt andrevondrei

> tar czf andrevondrei.tgz andrevondrei/
```

Class files should not be included into your tar-archive, and if there are any options required (or CLASSPATH settings) in order to compile, these should be described in the file README.txt. If you have any questions about the format, please do not hesitate to ask your teaching assistant.

Good luck!