Worksheet 3: Predictive Text Entry

MSc & ICY Software Workshop, Spring term 2019-20

Seyyed Shah and Uday Reddy

Assigned: Tuesday 5 February
Intermediate deadline : parts 1 and 2, Tuesday 12th February, 2:00pm
Final Deadline : all parts, Monday 19th February, 2:00pm.

As usual, include in your submission:

1. appropriate comments and JavaDoc.
2. **thorough** testing. (You may use JUnit wherever applicable.)

As well as data structures and algorithm complexity, this exercise assesses several concepts taught on the course. If you don't understand any part of the exercise, ask tutors and lab demonstrators.

Start early. The questions get progressively harder. All work and progress on the exercise must be submitted using Canvas.

You must submit parts 1 and 2 of the Worksheet by 6th February. 5% of the marks are allocated for this timely submission.

Contents

Introduction

In this exercise, you will write the algorithms for a sample application using the Java Collection classes. In the next exercise, you will attach a Graphical User Interface (GUI) to make it a full application. The sample application is that of *predictive text*.

Before the advent of touch screens, mobile telephones in English-speaking countries used a keypad like this one:

1	2 (abc)	3 (def)
4 (ghi)	5 (jkl)	6 (mno)
7 (pqrs)	8 (tuv)	9 (wxyz)
*	space	#

As you notice, there are keys for digits 1–9, used for dialing phone numbers. But these keys were also used to enter letters a–z. When a text message needed to be entered, the keys corresponding to the letters would be used. However, since there are multiple letters on each key, the required letter needed to be disambuated somehow.

In the basic system without predictive text, the user must press the appropriate key a number of times for a particular letter to be shown. Consider the word "hello". With this method, the user must press 4, 4, 3, 3, 5, 5, 5, then pause, then 5, 5, 5, 6, 6, 6.

To enter text more easily, the system of *predictive text* (also called "T9") was devised. The user presses each key only once and the mobile phone uses a dictionary to guess what word is being typed using a dictionary, and displays the possible matches. So the word "hello" can be typed in 5 button presses "43556" without pauses, instead of 12 in the standard system. The numeric string "43556" is referred to as a "*signature*" of the world "hello". If this is the only match, the user can press space and carry on. If there are multiple matches, the user might need to select one of them before proceeding.

A given numeric-signature may correspond to more than one word. Predictive text technology is possible by restricting available words to those in a dictionary. Entering the numeric signature "4663" produces the words "gone" and "home" in many dictionaries.

In this exercise, you will design and develop a predictive text system. For simplicity, assume that the user does not need punctuation or numerals. You must also limit your solutions to producing only **lower-case** words.

The final version of your programs should use the dictionary found in `/usr/share/dict/words` on the School's file systems. However, during testing, it is better for you to *create small dictionary files of your own for which you know what outputs to expect.*

All the classes in this worksheet should be placed in a package called `predictive`. Use the class/method names given in the question.

1 Prototypes and design (25%)

This part deals with building a "prototype" for the predictive text problem, which is not expected to be efficient, but it will be simple and allow you to compare it with the efficient implementation to be done in later parts.

Write the first two methods in a class named `PredictivePrototype` inside the package `predictive`.

1. (5%) : Write a method `wordToSignature` with the type:

   ```
   public static String wordToSignature(String word)
   ```

   The method takes a word and returns a numeric signature. For example, "home" should return "4663". If the word has any non-alphabetic characters, replace them with a " " (space) in the resulting signature. Accumulate the result character-by-character.

   You should do this using the `StringBuffer` class rather than `String`. Explain, in your comments, why this will be more efficient.

2. (10%): Write another method `signatureToWords` with the type:

   ```
   public static Set<String> signatureToWords(String signature)
   ```

   It takes the given numeric signature and returns a set of possible matching words from the dictionary. The returned list must not have duplicates and each word should be in *lower-case*.

   The method `signatureToWords` will need to use the dictionary to find words that match the string `signature` and return all the matching words.

   In this part of the exercise, you should not store the dictionary in your Java program. Explain in the comments why this implementation will be inefficient.

3. (10%): Create command-line programs (classes with `main` methods) as follows:

   - `Words2SigProto` for calling the `wordToSignature` method, and
   - `Sigs2WordsProto` for calling the `signatureToWords` method.

   Each program must accept a list of strings and call the appropriate method to do the conversion.

Hints:

- Use the `Scanner` class to read the dictionary line by line, assuming there is only one word per line.
- When reading the dictionary, ignore lines with non-alphabetic characters. A useful helper method to accomplish this would be:

  ```
  static boolean isValidWord(String word)
  ```

  in `PredictivePrototype`, which checks if a given word is valid.
- Words in the dictionary with upper case letters should be converted to lower-case because only lower-case letters should be returned by the `signatureToWords` method.

- You should be able to complete this part of the Worksheet and test it in about one lab session.
- To create the command-line programs, you will need to use the `args` array of the method:

  ```
  public static void main(String[] args)
  ```

  which contains the command line input. For example, when executing

  ```
  sxs@cca112:~$ java predictive.Words2SigProto Hello World! this is the input
  ```

  the `args` array will contain

  ```
  ["Hello", "World!", "this", "is", "the", "input"]
  ```
- You should ignore any words with non-alphabetic characters given in the input of `Sigs2WordsProto`.
- Format the output of `Sigs2WordsProto` as one line per signature, as there may be more than one word for a given numeric signature. E.g.

  ```
  sxs@cca112:~$ java predictive.Sigs2WordsProto 4663 329
  4663 : good gone home hone hood hoof
  329 : dax fax faz day fay daz
  ```

  the actual output you get will depend on the dictionary used.
- Notice that the package name `predictive` qualifies the class name, and this command works in the main directory. You can also use the `-cp ..` option to run the command from a different directory, e.g.,

  ```
  sxs@cca112:~/predictive$ java -cp .. predictive.Sigs2WordsProto 4663 329
  ```
- The program `Words2SigProto` can be tested by converting large amounts of text to signatures, the output can be used to test `Sigs2WordsProto` (and later, in timing comparisons). Try using news articles to start with.

2 Storing and searching a dictionary(20%)

In the remaining parts of the worksheet, you are asked to implement a number of dictionary classes that will be more efficient than the prototype. All of these classes should implement this interface:

```
public interface Dictionary{
        public  Set<String> signatureToWords(String signature);
}
```

The required method `signatureToWords` finds the possible words that could correspond to a given `signature` and returns them as a set.

In this part, you will read and store the dictionary in memory as a list of pairs. As the list will be sorted and in memory, a faster look-up technique can be used.

1. (15%) : Create a class named `ListDictionary`.

   Write a constructor for the class `ListDictionary` that takes a `String` path to the dictionary, reads stores it in an `ArrayList`. Each entry of the `ArrayList` must be a pair, consisting of the word that has been read in and its signature. For this purpose, you will need to create a class named `WordSig` that pairs words and signatures (see the hints).

   The `wordToSignature` method will be the same so you can re-use the code from the first part.

   The `signatureToWords` method must be re-written as an *instance method* in the `ListDictionary` class to use the stored dictionary. The `ArrayList<WordSig>` must be stored in *sorted order* and the `signatureToWords` method must use *binary search* to perform the look-ups.

2. (5%) : Design and create a command-line program `Sigs2WordsList` for testing the `ListDictionary` class.

   Compare the time taken to complete the execution of `Sigs2WordsList` and `Sigs2WordsProto` with the same large input(s). Is it possible to make the time difference between `Sigs2WordsList` and `Sigs2WordsProto` noticeable? Make a note of the data you use and your timing results.

Hints :

- Create a class which pairs the numeric signatures with words, like this:

```
public class WordSig implements Comparable<WordSig>{
   private String words;
   private String signature;

   public WordSig (...) { ... }

   public int compareTo(WordSig ws) { ... }

   ...
}
```

- When you read the dictionary you will need to create new `WordSig` objects.
- A list of `Comparable` objects can be sorted using the method `Collections.sort`[1].

[1] Find out more about collections and the comparable interface in the Java tutorial on Collections: https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html

- To automatically sort a list using the collections API, the objects `WordSig` stored in the list must implement the `Comparable` interface. That means they must have a `compareTo(...)` method. `compareTo` returns -1, 0 or 1 according to whether the current object is less than, equal to, or greater than the argument object, in the intended ordering.
- Sort the dictionary only once.
- You must decide how to define the `compareTo` method so as to allow efficient search for signatures. Even though, normally, you are expected to redefine the `equals` method to be consistent with `compareTo`, for this version of the program, you can ignore this requirement. That is, you should *not* attempt to define an `equals` method.
- You can search a sorted list using `Collections.binarySearch`. Its simplified type can be written as follows:

  ```
  static <T> int binarySearch(List<T>, T)
  ```

  Note that the type variable `T` in both the arguments must be *the same*.
- Binary search will return the index of the first match it finds. You must return *all* matching words. Scan above and below the found index to collect all matching words.
- The `time` command-line program on Linux machines will tell you how long a given command takes to complete. E.g.

  ```
  sxs@cca112:~/predictive/$ time java -cp .. predictive.Sigs2WordsList <input>
  <output>

  real    0m0.286s
  user    0m0.260s
  sys     0m0.010s
  ```

  Use the "real" elapsed time in all comparisons.

3 More efficiency (25%)

This part involves creating an improved implementation of the `Dictionary` interface using a `Map` data structure.

1. (15%) : Implement a new class `MapDictionary`.

   Write a constructor for the class `MapDictionary` that takes a `String` path to the dictionary and stores the dictionary in a multi-valued `Map`. In this context, a "multi-valued map" is a data structure that maps each signature to *set* of words. Using a `Map`, data can be retrieved quickly by looking up a signature as in `ListDictionary`, but now it does not require scanning either side of the index as earlier. `MapDictionary` will also allow efficient insertion of new words in the dictionary while still allowing fast look-up.

   You must choose a `Map` implementation from the Java Collections API. Explain how the map works and justify your choice in a comment.

   Write a method `signatureToWords` that returns, in a `Set<String>`, only the matching *whole words* for the given signature. The character length of each returned word must be the same as the input signature.

2. (5%) : Create a program `Sigs2WordsMap` that uses the `MapDictionary` class. It should be possible to modify just one line in your `Sigs2WordsList` program so that it can work with any given implementation of the `Dictionary` interface.

Hints:

- The `MapDictionary` class must implement `Dictionary`. Do not use the `WordSig` class.
- When deciding what your `Map` will store in `MapDictionary`, keep in mind that one signature often corresponds to several words.
- When developing `ListDictionary`, you may have noticed that it was useful to create helper methods to add words to the data structure. Creating add helpers will simplify the constructors of both `MapDictionary` and `TreeDictionary`.

4 Prefix-matching (25%)

This part involves creating another improved implementation of the `Dictionary` interface using your own tree data structure. This should allow the words or parts of words that match partial signatures, so that the users will be able to see the parts of the words they are typing as they type.

1. (20%) : Implement a new class `TreeDictionary` that now stores the dictionary in your own tree implementation. It should support efficient search as well as efficient insertion

of new words. In addition, `TreeDictionary` should support finding words when only some initial part of the signature (a *prefix*) is known. This is so that the user can see the part of the word they intend to type as they are typing.

The `TreeDictionary` class forms a recursive data structure, similar to, but more general than binary trees in the Worksheet 2 of this semester. This tree differs in that each node now has up to *eight branches*, one for each number (2-9) that is allowed in a signature. Each path of the tree (from the root to a node) represents a signature or part of a signature. At each node of the tree, you must store a collection of all the words that can possibly match the partial signature along the path. That means that every word that has a *prefix* corresponding to the partial signature appears in the collection. For example, if the dictionary has the words `a`, `ant` and `any`, then the words at nodes corresponding to paths would be as follows:

- at node 2, we have `a`, `ant` and `any`,
- at node $2, 6$, we have `ant` and `any`.
- at node $2, 6, 8$, we have only `ant`.

Write a constructor for the class `TreeDictionary` that takes a `String` path to the dictionary and populates the tree with words.

Write a method `signatureToWords` that returns, in a `Set<String>`, the matching words (and prefixes of words) for the given signature. The character length of each of the returned words or prefixes must be the same as the input signature.

2. (5%) : Create a program `Sigs2WordsTree`, similar to `Sigs2WordsMap`, that uses the `TreeDictionary` class.

   Compare the time taken to complete the execution of `Sigs2WordsMap` and `Sigs2WordsTree` with large inputs. Is it possible to make the time difference between `Sigs2WordsList` and `Sigs2WordsMap` or `Sigs2WordsTree` and `Sigs2WordsMap` noticeable? Again, make a note of the data you use and your timing results.

Hints:

- The `TreeDictionary` class must implement `Dictionary`. Do not use the `WordSig` class.
- Before starting `TreeDictionary`, sketch a tree-dictionary containing 2-3 words.
- Every node of `TreeDictionary` will have a collection of words and eight `TreeDictionary`s. You may use an array of `TreeDictionary` or just store several objects, as you prefer.
- The root node of `TreeDictionary` should not store any words.
- *In* `TreeDictionary` *it is more memory efficient to store only* ***whole words*** *as read-in from the dictionary.* You should do this and write a helper-method to trim all the words in a given list to produce the output of `signatureToWords`.