INFO1105 2016 Semester 2, Assignment 2

October 10, 2016

Submission details

- Due: Monday 24th of October 2016 at 9pm
- Submit your **report** via Blackboard (turnitin). The report must be in pdf format, and cannot be handwritten. Note that your submission is not complete until you see the "Congratulations - your submission is complete!" message.
- Submit your **source code** on Ed, *including* any JUnit tests you make.
- Submit your **source code** on eLearning, *including* any JUnit tests you make, and a **signed cover sheet** as a zip file. The cover sheet is downloadable on Ed.
- The policy for late submissions is described on slide 21 of Lecture 1a.
- This is an **individual assignment**, so your code and report should be entirely your own work. This is an implementation assignment and as such you should predominantly write your own data structure, not just invoke classes from other libraries. If you adapt or reproduce any code from the textbook, the Java libraries, or elsewhere, then this must be acknowledged, just like any other sources.

1 Overview

In this assignment each student (individually) will write a class that could form part of a collection library. The intended domain of application is in bioinformatics, where parts of someone's DNA can be represented as strings where each character is one of A, C, G and T; for example "GATTACA". The collection consists of keys, each of which is a string that represents a DNA sequence, and each key has an associated value (which is a string that gives some textual information about the sequence, such as its discoverer).

The code you write must implement a particular interface that we have defined, called PrefixMap. The code that you write must be built according to a particular data structure, called a Trie, that we describe below in more detail.

2 The PrefixMap interface

The PrefixMap interface has some methods inspired by the usual Map ADT, with some additional methods used to group and select keys based on their prefixes. The interface restricts the set of keys so that each is a string built from the alphabet of four characters A, C, G and T.

```
public interface PrefixMap {

        public boolean isEmpty();

        /*
         * How many keys are stored in the map
         */
        public int size();

        /*
         * Get the value corresponding to the key (or null if the key is not found)
         * if the key contains any character other than A,C,G,T, throw MalformedKeyException
         * if the key is null, throw IllegalArgumentException
         */
        public String get(String key);

        /*
         * Insert the value into the data structure, using the given key. If the key
         * already existed, replace and return the old value (otherwise return null)
         * if the key contains any character other than A,C,G,T, throw MalformedKeyException
         * if the key or value is null, throw IllegalArgumentException
         */
        public String put(String key, String value);

        /*
         * Remove the value corresponding to the given key from the data structure,
         * if it exists. Return the old value, or null if no value was found.
         * if the key contains any character other than A,C,G,T, throw MalformedKeyException
         * if the key is null, throw IllegalArgumentException
         */
        public String remove(String key);

        /*
         * return the number of keys which start with the given prefix if the prefix
         * contains any character other than A,C,G,T, throw MalformedKeyException
         * if the prefix is null, throw IllegalArgumentException
         */
        public int countKeysMatchingPrefix(String prefix);

        /*
         * return the collection of keys which start with the given prefix if the
         * prefix contains any character other than A,C,G,T, throw MalformedKeyException
         * if the prefix is null, throw IllegalArgumentException
         */
        public List<String> getKeysMatchingPrefix(String prefix);

        /*
         * Return the number of unique prefixes
         * e.g. if the tree stores keys GAT, GATTC, GATTACA, this method will return 8
         * because the prefixes are G, GA, GAT, GATT, GATTC, GATTA, GATTAC, GATTACA
         * In an uncompressed trie, this is the number of trie nodes, excluding the root
         */
        public int countPrefixes();

        /*
         * Return the sum of the lengths of all keys
         * e.g. if the tree stores keys GAT, GATTC, GATTACA, this method will return 15
         */
        public int sumKeyLengths();

}
```

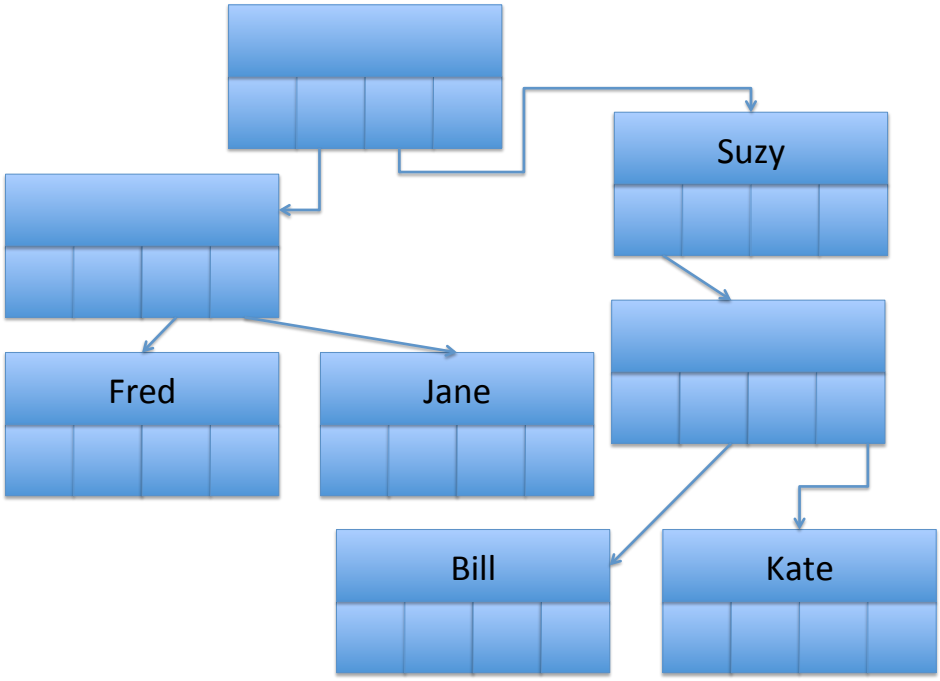

Figure 1: An example of the data structure

3 Trie Data Structure

3.1 Description

A Trie (also sometimes called a Prefix Tree), is a type of search tree where instead of storing the keys in each node, instead the path to reach that node defines the key. In data sets where keys often share a common prefix this can be an efficient way to store them, as those common prefixes are only represented once instead of many times. The Trie structure was described in lecture in week 10. In this assignment you will implement a variation of the Trie structure for the case where the keys are made up of only 4 possible characters (A, C, G and T) and the values are arbitrary strings. Thus each Node can have only 4 possible children (one where the next character is A, one where the next character is C, etc), and so we can define a Node class where there is an array of length 4 to hold the references to children Nodes. When we do this, we do not store the character in the Node, instead the character is found by looking at which child of the parent this Node is. Each Node can also hold a value, if the sequence of characters used to reach that position is one of the keys.

The diagram below shows the data structure storing the following key-value pairs: (G, Suzy), (GAC, Bill), (GAT, Kate), (CG, Fred), (CT, Jane). Notice how the path to the node containing Kate goes from the root to its the third child (corresponding to the character G), then from that node to its first child (corresponding to A), and from that node to its fourth child (T).

In this assignment you will write the code for a class `Assignment` which implements

the `PrefixMap` interface, using a Prefix Tree. The full skeleton code for the assignment is available for download on the resources section of Ed.

4 Deliverables

4.1 Code

You must produce a class called Assignment that is suitable to be in a collection library. It must implement the PrefixMap interface exactly as we have defined that. Your class should contain an appropriate private nested class Node that represents the Node objects in the Trie.

You are advised to use recursion when writing the methods, but this is not a requirement.

4.2 Report

You must write a short report:

- For each of the interface methods, describe the algorithm used, state the running time of this algorithm in big-Oh notation, and give a brief argument justifying that this cost is correct. You should express the costs in terms of n (the number of key-value pairs in the collection), m (the length in characters of the argument key or prefix), and k (the number of keys that start with the given prefix).
- Describe how you tested your code. List the test cases you wrote, stating briefly the purpose of each test.

5 Marking

5.1 Code automarking [50%]

Part of the marking of your code will be done automatically on Ed. You are encouraged to submit early and often – there is no penalty for making multiple submissions. There will be some visible tests to help you verify that your solution is configured correctly. The remaining test cases will be invisible until *after* the submission deadline. Test your code carefully to ensure that your solution works correctly. Note that even if your code passes the tests, you will only receive these marks if you have implemented the trie data structure (as checked by the tutor who looks at your code).

- Pass level: The code passes all of the publically visible tests, and follows the described data structure.
- Distinction level: As for Pass, and the code also passes a majority of the private tests.

5.2 Code quality [20%]

This is based on examination of the code. Note that you will only receive these marks if you have implemented the trie data structure and the code compiles.

- Pass level: You have implemented the trie data structure in a sensible way, and the code compiles. Most of the code is understandable without particular effort, with mostly sensible layout, and meaningful comments where needed.

- Distinction level: As for Pass, and the code is written to be efficient and also easy to follow, helping the reader and avoiding distraction (this implies that the code is well-structured with appropriate use of helper methods; it has a good choice of layout and comments, well chosen names; and idiomatic use of Java language).

5.3 Testing [10%]

This is based on your report and on examination of the code. Note that you can gain these marks even if the code doesn't work correctly or even compile, as long as JUnit tests are written properly.

- Pass level: There is a reasonable range of tests for normal cases of the public methods, which are provided in JUnit and are listed in the report.
- Distinction level: As for Pass, and also the tests are well chosen and justified, covering a significant range of both normal and corner cases.

5.4 Analysis of runtime [20%]

This is based on your report. Note that you can gain these marks even if you don't write any code, as long as you analyse algorithms that you describe which operate on the Trie.

- Pass level: You state the correct big-Oh for majority of public methods, when each is implemented from the algorithm as you described it.
- Distinction level: As for Pass, and also you provide convincing and valid arguments in most cases.

6 Appendix: Skeleton code

The skeleton code is included here for your reference, but we **strongly** recommend that you download it from Ed instead, to avoid any text encoding artifacts that might be introduced by copying it from this pdf.

6.1 MalformedKeyException.java

```
// Do not edit this file
public class MalformedKeyException extends RuntimeException {

        private static final long serialVersionUID = -5961637854101313320L;

}
```

6.2 Assignment.java

```
import java.util.List;

public class Assignment implements PrefixMap {

        //TODO implement a nested Node class for your linked tree structure
        private class Node {
                //TODO implement this
        }
```

```
        /*
         * The default constructor will be called by the tests on Ed
         */
        public Assignment() {
                // TODO Initialise your data structure here
        }

        @Override
        public int size() {
                // TODO Implement this, then remove this comment
                return 0;
        }

        @Override
        public boolean isEmpty() {
                // TODO Implement this, then remove this comment
                return false;
        }

        @Override
        public String get(String key) {
                // TODO Implement this, then remove this comment
                return null;
        }

        @Override
        public String put(String key, String value) {
                // TODO Implement this, then remove this comment
                return null;
        }

        @Override
        public String remove(String key) {
                // TODO Implement this, then remove this comment
                return null;
        }

        @Override
        public int countKeysMatchingPrefix(String prefix) {
                // TODO Implement this, then remove this comment
                return 0;
        }

        @Override
        public List<String> getKeysMatchingPrefix(String prefix) {
                // TODO Implement this, then remove this comment
                return null;
        }

        @Override
        public int countPrefixes() {
                // TODO Implement this, then remove this comment
                return 0;
        }

        @Override
        public int sumKeyLengths() {
                // TODO Implement this, then remove this comment
                return 0;
        }

}
```

6.3 PrefixMap.java

```
// Do not edit this file
```

```
import java.util.List;

public interface PrefixMap {

        public boolean isEmpty();

        /*
         * How many keys are stored in the map
         */
        public int size();

        /*
         * Get the value corresponding to the key (or null if the key is not found)
         * if the key contains any character other than A,C,G,T, throw MalformedKeyException
         * if the key is null, throw IllegalArgumentException
         */
        public String get(String key);

        /*
         * Insert the value into the data structure, using the given key. If the key
         * already existed, replace and return the old value (otherwise return null)
         * if the key contains any character other than A,C,G,T, throw MalformedKeyException
         * if the key or value is null, throw IllegalArgumentException
         */
        public String put(String key, String value);

        /*
         * Remove the value corresponding to the given key from the data structure,
         * if it exists. Return the old value, or null if no value was found.
         * if the key contains any character other than A,C,G,T, throw MalformedKeyException
         * if the key is null, throw IllegalArgumentException
         */
        public String remove(String key);

        /*
         * return the number of keys which start with the given prefix if the prefix
         * contains any character other than A,C,G,T, throw MalformedKeyException
         * if the prefix is null, throw IllegalArgumentException
         */
        public int countKeysMatchingPrefix(String prefix);

        /*
         * return the collection of keys which start with the given prefix if the
         * prefix contains any character other than A,C,G,T, throw MalformedKeyException
         * if the prefix is null, throw IllegalArgumentException
         */
        public List<String> getKeysMatchingPrefix(String prefix);

        /*
         * Return the number of unique prefixes
         * e.g. if the tree stores keys GAT, GATTC, GATTACA, this method will return 8
         * because the possible prefixes are G, GA, GAT, GATT, GATTC, GATTA, GATTAC, GATTACA
         */
        public int countPrefixes();

        /*
         * Return the sum of the lengths of all keys
         * e.g. if the tree stores keys GAT, GATTC, GATTACA, this method will return 15
         */
        public int sumKeyLengths();

}
```