CS 2112 Fall 2020
Assignment 3
Data Structures and Text Editing

Due: Monday, October 12, 11:59PM
Design Document due: Sunday, October 4, 11:59PM

Text editors must store large dictionaries of words and quickly access them when performing common tasks such as word completion, spell checking, and text search. In this assignment you will implement core data structures and algorithms for a simplified text editor. The first part introduces a generic hash table, a prefix tree, and a Bloom filter. The second part requires you to create plugins for a text editor that performs word completion and spell checking. The last part contains written problems focusing on the concepts introduced in class.

This assignment will take some time. Get started early!

1 Instructions

1.1 Grading

Solutions will be graded on both correctness and style. A correct program compiles without errors or warnings and behaves according the requirements given here. A program with good style is clear, concise, and easy to read.

A few suggestions regarding good style may be helpful. You should use brief but mnemonic variables names and proper indentation. Your code should include comments as necessary to explain how it works, but without explaining things that are obvious.

1.2 Partners

You will work with one partner for this assignment. You must create groups on CMS by Thursday, October 1 at 11:59 PM if you are picking a partner, otherwise we will randomly assign you a partner. We will assign repos on the Cornell CIS Github instance for A3. Post privately on Piazza with your netIDs when your group are ready to have your repo set up.

Remember that the course staff is happy to help with problems you run into. Use Piazza for questions, attend office hours, or set up meetings with any course staff member for help.

1.3 Documentation

For this assignment, we are especially looking for good documentation of the interfaces implemented by your data structures. Write Javadoc-compliant comments that crisply explain what all the methods do at a level of abstraction that enables a client to use your data structure effectively, while leaving out unnecessary details.

1.4 Restrictions

Your use of `java.util` will be restricted for this assignment. **Classes** from `java.util`, except for `Scanner`, may not be used anywhere in your code except in a JUnit test suite (see §7). **Interfaces** from `java.util` may be used anywhere in your code to guide your internal data structures.

While we require that you respect any interfaces we release, you are allowed (and even expected) to create your own classes and interfaces to solve portions of the assignment.

1.5 Importing and Running

Starting with this assignment, we will be using a system called Gradle in the release code. Gradle automatically adds any dependencies into your project without the need to add them manually. However, the steps to get your project into Eclipse are slightly different. You will need the Gradle extension, which can be found in Eclipse Marketplace. (This can be found in the drop-down help menu). Here, you can search for Gradle, and install the extension. After this, go to File → Import and then under the Gradle folder, select "Existing Gradle Project". Walk through the start menu, and then select the release folder.

Now go to the Run drop-down menu, and select `Run Configurations`. Under `Gradle Tasks`, select `A3release -run`. Then click Run. You should now be able to run the project as normal.

You may also run gradle from the command line. This can simply be run using the gradlew files that are in the release.

1.6 Tips

In this assignment, you will be modifying an application with a graphical user interface (GUI). The application has significant library dependencies because it builds on the JavaFX GUI library. To make sure you don't run into headaches right before the deadline, start early to make sure that you have the right setup to successfully modify, compile, and run the application.

2 Hash tables

Your task in this section is to implement a hash table with chaining, as discussed in class. The lecture notes on hash tables have some helpful pointers, but we will also provide a high level overview here.

A hash table is a data structure which maintains key value pairs. Each key is hashed to an index using a hash function. Elements have a high probability of being hashed to unique indices, but in the case of a collision, elements can either be stored in the same index through use of a linked list (chaining) or just stored in the next available index (probing).

The benefits of a hash table are that common data structure operations have a significantly better runtime. For example, lookup in an array is $O(n)$ but for a hash table, it is $O(1)$. You will learn more about this in lecture, but getting a head start and understanding it on a high level can help with this assignment.

2.1 Collisions

You should use chaining to handle collisions. You are expected to keep track of the load factor and to resize your table whenever the load factor crosses a threshold. A smart choice of load factor will keep memory usage reasonable while avoiding collisions.

2.2 Implementation

Implement the provided class `HashTable<K,V>`. Your hash table should implement Java's `java.util.Map<K,V>` interface, which is generic. The methods `containsKey`, `get`, `put`, and `remove` should have expected $O(1)$ (constant) running time. Your hash table should take up $O(n)$ (linear) space, where n is the number of entries in the hash table.

The implementation of the method `keySet()` should return an instance of an implementation of `java.util.Set<K>` that supports the following methods: `size()`, `isEmpty()`, `toArray()`, and `contains(Object)`. The remaining methods, including `toArray(T[])`, can throw an `UnsupportedOperationException`.

The method `hashCode()`, which is defined for every Java object, can be used by a hash function that you create to compute the bucket in which to place each object. However, since `hashCode()` is not required to produce results that behave as if they are random, you don't want to use `hashCode()` directly to compute the bucket index. For example, the default implementation of `hashCode()` returns the object's memory address, therefore only produces numbers that are multiples of 4. Another hash function is needed to provide diffusion throughout the buckets. The class `java.security.MessageDigest` provides high-quality hash functions that can be used for this purpose, although they are more expensive than necessary for most applications. The course notes have tips on how to design a `hashCode()` method; see also this Wikipedia page.

3 Prefix trees

A prefix tree, also known as a **trie**,[1] is a data structure tailored for storing and retrieving strings. The root node represents the empty string.[2] Each possible next character branches to a different child node. Strings stored in the trie must be inserted explicitly by the user; prefixes of such strings, although they occur along paths in the trie, are not considered to be stored in the trie unless they have been explicitly inserted.

[1] Pronounced like "try".

[2] Note that the empty string is `""`, the string of length 0, not `null`.

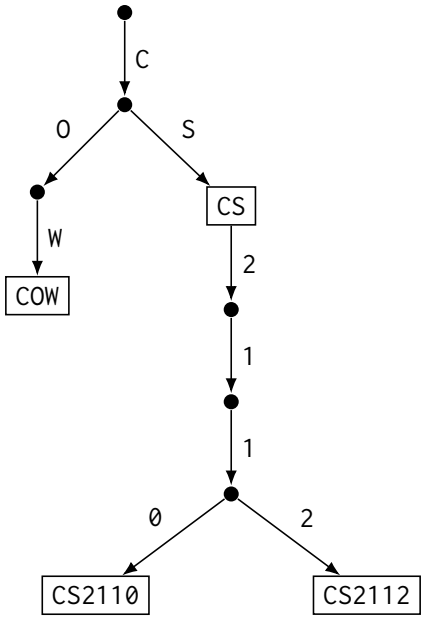

Figure 1: A trie containing the strings `COW`, `CS`, `CS2110`, and `CS2112`.

For example, the trie of Fig. 1 contains the four strings `''COW''`, `CS`, `CS2110`, and `CS2112`. The strings `C`, `CS211`, `CO`, and the empty string, although they appear as prefixes of strings stored in the trie, are not considered to be stored in the trie themselves.

If a string is stored in the trie, there is a unique node corresponding to that string and a unique path from the root down to that node obtained by tracing the characters in the string. That node can contain a boolean flag to indicate that that string has been stored in the trie. There is no need to store the string itself at that node; the string can be recovered by tracing the path from the root down to that node, keeping track of the characters along the way.

3.1 Implementation

Implement the provided `Trie` class. The operations `insert`, `delete`, and `contains` should have $O(k)$ running time, where k is the length of the string. In other words, the running time of these operations should be proportional to the length of the given string. Your trie should also implement the method `closestWordToPrefix()`, which returns the shortest entry in the trie having the given prefix. This shortest string can be found using breadth-first search.

The method `closestWordToPrefix()` should be case-sensitive. For example, it should report `CS2110` or `CS2112` if the argument is `CS211`, but not if the argument is `cs211`.

4 Bloom filters

A Bloom filter is a probabilistic constant-space data structure for maintaining a set of elements and testing whether a given element is in the set. It is probabilistic in the sense that false positives may occur with small probability (that is, an element may be reported to be in the set when it is not), but false negatives never occur (that is, if an element is reported not to be in the set, then it is definitely not in the set).

An empty Bloom filter is a bit array of 0s. To insert an element into a Bloom filter, put the element through k different hash functions. Use the results of these hash functions as indices into the bit array. Set those k bits in the bit array to 1.

To determine if an element is in the Bloom filter, check all of its hash indices. If all of them are 1 in the bit array, report that the element is in the set. If at least one of them is 0, report that the element is not in the set.

If the objects contained in the Bloom filter are strings, the k different hash functions can be simulated with a single hash function by appending a different single character (e.g., `a`, `b`, `c`, ...) to the end of the string before hashing.

4.1 Example of a false positive

Consider a Bloom filter for strings represented by a bit array of length 2, initially empty. Suppose only one hash function is used to index strings. First, the string `CS2112`, whose (hypothetical) hash value is 0, was inserted into the Bloom filter, setting the 0^{th} bit to 1 in the bit array. Now, to check whether `CS2110`, whose hypothetical hash value is also 0, is in the Bloom filter, we check if the bit at position 0 is 1. Since this is the case, we conclude that the Bloom filter does contain the String `CS2110` when in fact it does not.

A larger bit array, more hash functions, and better quality hash functions all reduce the likelihood of false positives.

4.2 Implementation

Implement the provided `BloomFilter` class.

5 Text editor

The text editor supports text search, spell checking, and autocompletion. These features are specified by the interfaces `SearchModule`, `SpellCheckModule`, and `AutoCompleteModule`. You are to provide implementations. The factory class `ModuleFactory` contains factory methods that should access your implementations. Instances returned from the factory methods are used by the main text editor program.

Search, spell checking, and autocompletion should all convert dictionary words to lowercase before searching. The editor already converts all input to lowercase letters.

5.1 Architecture

The text editor project is broken up into three packages. The `editor` package includes all of the view and model code for the editor. The `modules` package contains all of the plugins providing functionality for text search, spell checking, and autocompletion. The `util` package contains all of the data structures you will implement. These data structures store and manipulate data for the plugins. While all the code you are required to write resides in the `modules` and `util` packages, you are welcome to look inside the `editor` package to get a taste of graphical user interface (GUI) code.

5.2 Dictionary file

After the text editor is started, spell checking and autocompletion are unavailable until a dictionary file is loaded. Any newline-separated list of words will work as a dictionary file. WinEdt provides such a file. On Macintosh and most Linux distributions, a good dictionary file can be found at `/usr/share/dict/words`. To load a dictionary file, click the top left button of the text editor.

5.3 User interaction

If your modules work correctly, word-completion suggestions from the autocomplete module should be displayed in the lower-left corner of the editor window. Misspelled words should be highlighted if you click the "check" button in the top left. To reset spell checking, click the adjacent "X" button. Additionally, the time spent spell checking should be reported in the lower-right corner after each run of spell checking. If you enter a string in the search window at the bottom and click the search button, the first occurrence of this string should be highlighted.

6 Performance

Performance analysis is a component of the grade for this assignment. You should choose data structure(s) wisely to be efficient in both memory usage and runtime. Justify your design in `README.txt`. We are looking for quantified comparisons of performance when you use different data structures to back the text editor modules. This week in lab, we covered VisualVM, which can give a lot of insight about where time is being spent in your code.

Both correctness and performance are important when we evaluate how well the editor plugins work.

In addition to justifying your choice of data structures, you should perform the following specific performance tests:

- Verify that the `put` and `get` methods of your hash table are $O(1)$ by reporting the running time for each as the number of elements in the hash table increases.

- Verify that your hash function produces reasonable diffusion by reporting the number of empty buckets and the number of collisions for various sizes of the hash table.

`System.nanoTime()` can be very useful for finding running times directly.

As you saw in lab, VisualVM can be very useful for determining relative runtime of specific functions in your code. Add screenshots of your profiling from VisualVM and provide a description of your findings.

Report your performance evaluation in file `perf.pdf`.

7 Testing

In addition to the code you write for the data structures and text editor plugins, you should also submit any tests that you write. Testing is a component of the grade for this assignment.

You should implement your test cases using JUnit, a framework for writing test suites. JUnit has excellent Eclipse integration that makes it easy to use. A small example demonstrating how to write JUnit tests is included. To include the JUnit runtime support in Eclipse, right click on your project and select Build Path > Add Libraries... > JUnit > JUnit 5.

You should not only test whether the program works correctly from the command line interface, but also write test cases for each of the data structures you implement.

Test cases should be placed in a top-level directory named `src/tests`, whereas the rest of your implementation would be in `src/main`.

There are several good strategies for writing test cases. In **black-box functional testing**, the tester defines input–output pairs in which the inputs provide good coverage of the input space. Each input is accompanied by the expected output as defined by the specification. We expect you to define functional test cases for your program as a whole and for each data structure you implement.

A second approach to testing is **random testing**, in which the inputs are generated randomly but in a way that satisfies the preconditions. A random test case might generate a sequence of randomly chosen inputs to a single method or to a randomly chosen method from a set of methods. This form of testing can catch bugs simply when the code fails with an exception or assertion error. Often an effective way to randomly test functional correctness is to test whether the behavior of the code matches that of a simple **reference implementation** on which the same operations are performed. For example, the `java.util` libraries may be used to build simple reference implementations for each of the abstractions you are implementing. We expect you to use random testing on at least one abstraction you develop in this assignment.

8 Written problems

8.1 Abstraction

The standard Java interface `SortedSet` describes a set whose elements have an ordering. Abstractly, the set keeps its elements in sorted order. Here is a much simplified version:

```
1  /** A set of unique elements kept sorted in ascending order. */
2  interface SortedSet<T extends Comparable<T>> {
3      /** Effect: Add x to the set if it is not already there. */
4      void add(T x);
5
6      /** Tests whether x is in the set. */
7      boolean contains(T x);
8
9      /** Effect: Remove element x. */
10     void remove(T x);
11
12     /** Returns the first element in the set. */
13     T first();
14 }
```

1. The specifications of some of these methods are incomplete. Clearly identify the problems and write better specifications for the methods that need to be improved. You may change method signatures if you justify the change.

2. There are many ways to implement this set abstraction. One possibility is as a linked list data structure in which there are no duplicates and the elements are kept in sorted order:

```
class SortedList<T extends Comparable<T>> implements SortedSet<T> {
    /**
     * A linked list of values starting at {@code head}, which may be {@code null}
     * to represent an empty list.
     *
     * <p>Invariant: the list nodes starting from {@code head} have values in ascending
     * sorted order with no duplicates.
     */
    ListNode<T> head;
}

class ListNode<T extends Comparable<T>> {
    T value;
    ListNode<T> next;

    ListNode(T v, ListNode<T> n) {
        value = v;
        next = n;
    }
}
```

The `SortedList` implementation is obviously incomplete. Give the most efficient, concise code you can to implement the `first` and `remove` methods, taking into account the representation and class invariant.

3. Now, suppose we want a different implementation `UnsortedList` that is similar to `SortedList` and uses the same `ListNode` class, but has no class invariant:

```
class UnsortedList<T extends Comparable<T>> implements SortedSet<T> {
  /**
   * A linked list of values starting at {@code head}, which may
   * be {@code null} to represent an empty list.
   */
  ListNode<T> head;
  ...
}
```

`UnsortedList` should still correctly implement the `SortedSet` interface. Implement the `add`, `first`, and `remove` methods as simply and concisely as you can, taking into account the representation and class invariant.

Since `SortedList` and `UnsortedList` implement the same specification, the client should not be able to tell which one is being used, except perhaps by timing.

4. Briefly discuss the advantages and disadvantages of each of these two implementations. Under what conditions it would be more appropriate to use `SortedList`? ...`UnsortedList`?

8.2 Asymptotic complexity

5. Consider the code snippet below. Give a tight bound on its time complexity using big-O notation, and briefly justify your answer.

```
1 for (int i = 5; i < n; i ++) {
2     if (i % 2 == 0) {
3         for (int j = i + 1; j < n; j ++) {
4             for (int k = 7; k < 70000; k++) {
5                 System.out.println("2112 is great!");
6             }
7         }
8     }
9 }
```

6. Show that $n^2 \lg n$ is $O(n^3)$. Be sure to specify a witness pair (k, n_0).

7. Is it true that 5^{5^n} is $O(25^n)$? Give a witness if true, or argue that no such witness exists.

8.3 Hashing

8. Show the state of the underlying array of a hash table, when implemented with chaining and then with linear probing. Assume the hash function is simply n modulo the

length of the array. The elements inserted into the array are 0, 5 ,2, 1, 10, 42, 56, 2112, 2019, 7, 3, 4, 11, 115. The initial length of the array is 5, and the maximum load factor for the chaining implementation is 2.

For the probing implementation, assume the maximum load factor is one and that the array size is doubled when it is reached.

9 Submission

Compress exactly these files into a `zip` file to submit on CMS:

- `README.txt`: This file should contain your name, the netIds of you and your partner, all known issues with your submitted code, the names of anyone you discussed the assignment with (excluding course staff), and any other sources that should be acknowledged.
  In addition, you should briefly describe your design, noting any interesting design decisions you encountered, and briefly discuss your testing strategy.
- **Source code**: Because this assignment is more open than the last, you should include all source code and resources required to compile and run your project. All source code should reside in the `src` directory with an appropriate package structure.
- **Tests**: You should include code for all your test cases in a package named `tests` separate from the rest of your source code. Subpackages are permitted.
- `written.txt` or `written.pdf`: This file should include your response to the written problems.
- `perf.pdf`: This file should include your performance analysis.

Do not include any `.class` files or any other extraneous files.

All `.java` files should compile and conform to the prototypes we gave you. We write our own classes that use your classes' public methods to test your code. *Even if you do not use a method we require, you should still implement it for our use.*