CS 2112 Fall 2024

Assignment 3

Data Structures and Text Editing

Due: Tuesday, October 8, 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 and a prefix tree. 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!

Updates

- 10/4 - clarified submission criteria.

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 **must** work with a partner for this assignment. We will assign repos on the Cornell CIS Github instance for A3, so **you will not be able to see the starter code until you have chosen a partner**. You may **not** work with your A2 partner.

You must create groups on CMS by Thursday, September 26 at 11:59 PM if you are picking your partner, otherwise we will randomly assign you a partner. Post privately on Ed 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 Ed 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 implementation details that a client does not need to know.

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 §5). The class `java.math.BigInteger` may not be used in your implementation either. **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 Generative AI

Generative AI, such as ChatGPT, is not permitted in any capacity for this assignment. Basic code completion in IntelliJ is permitted. Consult with the course staff if you have specific questions.

1.6 Importing and Running

Once your github repo has been set up, open IntelliJ and choose `File` → `New` → `Project from Version Control`. Select github enterprise from the sidebar. Then select A3release, and choose the folder you want to clone into. Select `Trust Proect`.

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. IntelliJ should automatically build the project using Gradle.

Once the build is done, you will have to set up the run configuration for the project. On the right sidebar of the IDE there will be an icon of an elephant that says `Gradle` when moused over, click on that. A sidebar will open, select `A3Release` → `Tasks` → `application` → and then double click `Run`. This will run the project and reveal the GUI you will be using. To stop running the project, close the GUI as you would any normal computer application. To rerun the application, you should now just be able to select the green play arrow at the top of the screen.

An alternative way to set up the run configuration is to click the box to the left of the play button at the top of the screen. This will open up the `Run/Debug Configurations` dialog. Now click the `+` on the top left of the screen and select Gradle. Then in the `Name` input box type something such as ”Run A3”, then in the `Run` input field, simply type in ”run”. Select `Apply`, then `OK`. You should now be able to click the green play arrow to run the application.

1.7 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. The lecture notes on hash tables have some helpful pointers, but we will also provide a high level overview here, since we won’t cover them for a few more lectures.

A hash table is a data structure which maintains key value pairs. Each key is mapped 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 (multiple elements mapping to the same index) 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 in the average case. 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 class `HashTable<K,V>`. Your hash table should implement the interface `java.util.Map<K,V>`, 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 `HashTable<K,V>` constructor need not accept a specification for the exact amount of buckets instead the parameter should be a hint to the number of buckets that will exist within your implementation. 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.

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.

[1] Pronounced like "try".

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

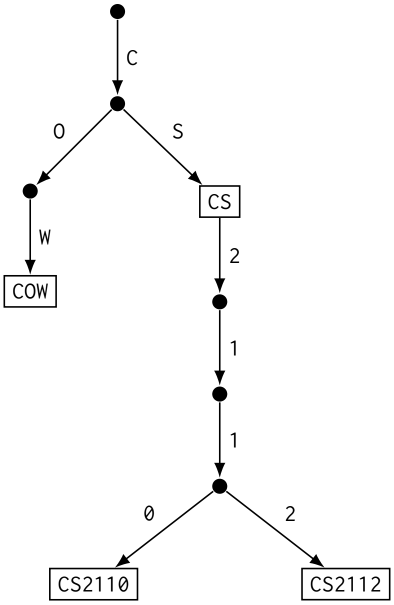

Figure 1: A trie containing the strings COW, CS, CS2110, and CS2112.

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 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.

4.1 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.

4.2 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.

4.3 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.

4.4 Implementation

You should not modify any code in the `editor` package. The functionality for the editor will come from your implementations of the interfaces in the `modules` package. Your implementation of these interfaces should stand alone and follow the given specifications without modifications to the `editor` package.

5 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. IntelliJ makes running JUnit tests very easy, just click the green arrow next to the test class name to run all tests, or run individual test methods by clicking the green arrow next to the one you would like to run.

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.

6 Performance and Correctness

6.1 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.pdf`.

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 use `System.nanoTime()` to 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.

Create the following 4 graphs, across multiple sample sizes n, and multiple trials per n:

- `Put` time
- `Get` time
- Number of empty buckets
- Number of collisions

Include a line of best fit. Excel or Google Sheets can be helpful in creating these graphs.

Include these graphs in the file `perf.pdf`.

Be aware that Java programs run very slowly when they first start, because libraries are being loaded and code is run in a slower, interpreted mode initially. Frequently used code is compiled "just in time" by the JIT compiler to machine code that runs at least an order of magnitude faster. Try to collect performance measurements only after the program has run for, say, 10 seconds.

In lab, we will cover VisualVM and profiling, which can give a lot of insight about where time is being spent in your code. VisualVM is more applicable to larger applications, so we will not require you use it to profile your assignment. However, it is helpful to understand and verify your program's behavior and asymptotic complexity.

6.2 Correctness

A good way to see if your tests are actually *testing* your code well is to try and trace what branches of code are executed. For instance, you may have inadvertently constructed a test suite that tests one method very thoroughly, but that omits another method altogether. You also may could be always avoiding one buggy `else if` statement that only executes for edge cases and all your tests pass because they bypass the bugs. Regardless of whether you choose to approach testing from a randomized, glass or black box method, you should always strive to make sure your code runs at least once in a test suite for sanity's sake.

Tracing these branches manually can be rather difficult, but there are tools that can help you design your tests to achieve better overall **coverage** of your code. IDEs like IntelliJ often have built in coverage tools. These tools helpfully tell you exactly how much of your code is being executed in your unit tests. You should use Run with Coverage to achieve as close to 100% coverage as possible on your tests for `HashTable`.

To run a specific test with coverage, you should already have an existing active test/run configuration that you wish to run with coverage. Then, you can select `Run` → `Run <configuration> with Coverage` from the menu bar; it should look like a run with a shield icon. If the configuration runs as expected and no coverage pops up, click into the edit run configuration window, and scroll all the way to the bottom to add the specific directories that you are interested in tracing coverage for. Once you have successfully run a test with coverage on, you should see a Coverage tab pop up. If not, you can go to `View` → `Tool Windows` → `Coverage` to open it. The fourth button on the Coverage tool window will allow you to export your test results as an HTML file; include this export in the `Coverage/` subdirectory with your final submission in your ZIP.

7 Written problems

7.1 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:

```java
/** A set of unique elements kept sorted in ascending order. */
interface SortedSet<T extends Comparable<T>> {
    /** Effect: Add x to the set if it is not already there. */
    void add(T x);

    /** Tests whether x is in the set. */
    boolean contains(T x);

    /** Effect: Remove element x. */
    void remove(T x);

    /** Returns the first element in the set. */
    T first();
}
```

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:

   ```java
   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:

   ```java
   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`?

7.2 Asymptotic complexity

Recall that a function $f(n)$ is $O(g(n))$ if there exist positive constants k and n_0 such that for all $n \geq n_0$, $f(n) \leq kg(n)$. The constants k and n_0 together are a **witness** to the fact that $f(n)$ is $O(g(n))$.

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

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

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

7. Show that $f_1(n) + f_2(n)$ is $O(n^2)$, if each of $f_i(n)$ is $O(n^2)$.

8. Is it true that 3^{2n} is $O(3^n)$? Give a witness if true, or argue that no such witness exists.

7.3 Hashing

9. 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 4, 15, 54, 43, 25, 42, 30, 2112, 2024, 21, 9, 65, 44, 219.

The initial length of the array is 5, and the maximum load factor for the chaining implementation is 2, and for the probing implementation is 1. The array size is doubled when the maximum load factor is reached.

8 Submission

You should submit these items on CMS:

- `README.pdf`: 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 (including clarifications from 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. You can follow the design overview guidelines on the course web site.
- `written.txt` or `written.pdf`: This file should include your response to the written problems.
- `perf.pdf`: This file should include your performance analysis.
- **Source code**: Please compress your code into a zip file with the following structure:

```
|__ src
|   |__ main
|   |__ test
|__ Coverage
```

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. You should include code for all your test cases in `test`. Subpackages are permitted. Do not include any `.class` files or any other extraneous files. Please export your code coverage report into a subdirectory named `Coverage` in the root directory of your zip.

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.*