CS2112—Fall 2015
Assignment 3
Data Structures and Text Editing
Due: Tuesday, September 29, 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.

0 Updates

- 9/16: The current code release does not include the sample A2 code described in 8.2; the code will be rereleased with the sample A2 code after the A2 due date.
- 9/22: A2 code now included.

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 alone for this assignment. But 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 in your code outside a JUnit test suite. *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.

2 Hash tables

Refer to the lecture notes for an overview of hash tables.

2.1 Collisions

You should use chaining to handle collisions as described in lecture.

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 the load factor will keep memory usage reasonable while avoiding collisions.

2.2 Implementation

Your hash table should implement Java SE 7's `java.util.Map` interface which is generic. Methods `containsKey(Object)`, `get(Object)`, `put(K, V)`, and `remove(Object)` should have expected $O(1)$ (constant) running time. Your hash table should cost $O(n)$ (linear) space, where n is the number of entries in the hash table.

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

The method `hashCode()`, part of every Java object, can be used as an input to a hash function that computes the bucket in which to place each object. However, since `hashCode()` is not required to produce results that behaves 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 and therefore only produces numbers that are multiples of 4. Another hash function is needed to provide diffusion of the information in the hash code throughout the bucket index. The class `java.security.MessageDigest` provides high-quality hash functions that can be used for this purpose, though they are more expensive than necessary for most applications.

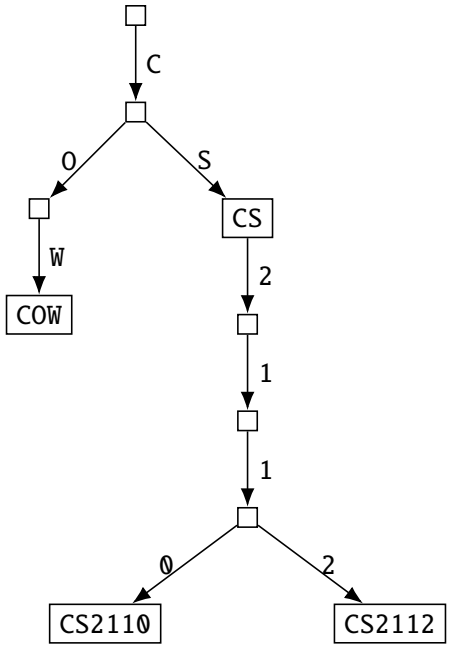

Figure 1: A trie containing the strings `COW`, `CS`, `CS2110`, and `CS2112`, where string terminations are represented by the value of the strings at the corresponding nodes.

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. Each possible next letter branches to a different child node. For each node where a string terminates, that node may contain either a flag indicating string termination or the value of the string. In the flag representation, every string in the data structure is determined by the path along the trie. Figure 1 shows an example of a trie in the latter representation.

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 method `closestWordToPrefix` which returns the shortest entry in the trie having the given prefix. This shortest string can be found using breadth-first search.

4 Bloom filters

A Bloom filter is a probabilistic constant-space data structure for testing whether an element is in a set. It is probabilistic in the sense that a false positive (i.e., that an element is reported to be in the

[1] Usually pronounced like “try”, not like “tree”.

set when it is not) may occur, but false negatives cannot. A Bloom filter with nothing in it 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 not in the Bloom filter, check all of its hash indices. If any of them is zero, the element must not be in the Bloom filter.

Each of the k different hash functions on strings may be simulated 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.

The size of the bit array backing the Bloom filter and the number of hash functions affect the probability 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, specified by interfaces `SearchModule`, `SpellCheckModule`, and `AutoCompleteModule`, must be implemented. Factory class `ModuleFactory` contains factory methods that should access your implementation of these features. Instances returned from the factory methods are used by the main text editor program.

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.

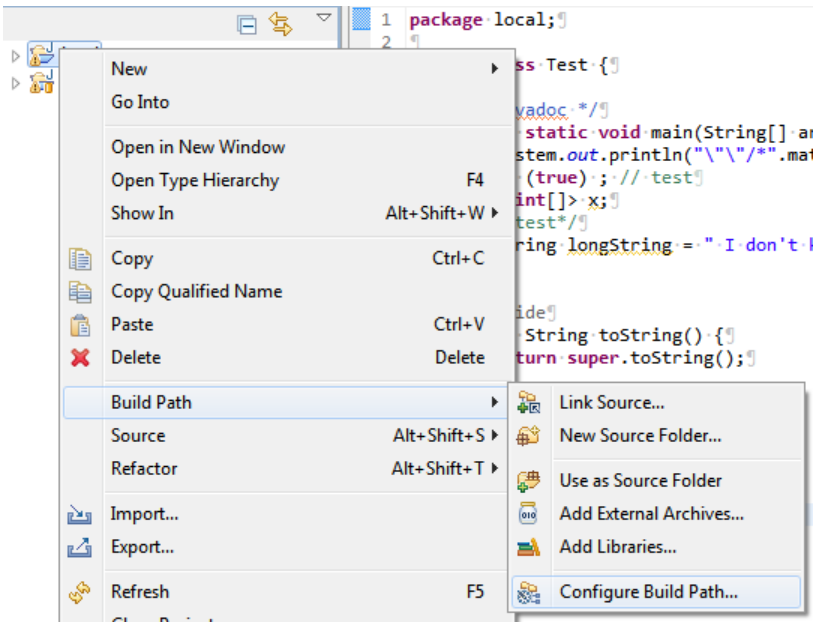

Figure 2: A screenshot of the Eclipse project context menu for configuring build path.

5.2 JavaFX

JavaFX runtime jar is required to run the text editor. To add JavaFX to your build path, open the Eclipse project properties as shown in Figure 2. On the "Libraries" tab, select "Add External JARs..." (see Figure 3). Browse for the JavaFX runtime jar, located at

```
<jdk_path>/jre/lib/jfxrt.jar
```

where `<jdk_path>` is the directory containing the installation of your Java JDK.

5.3 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`; the Ubuntu website also provides such a file. To load a dictionary file, click the top left button of the text editor.

5.4 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

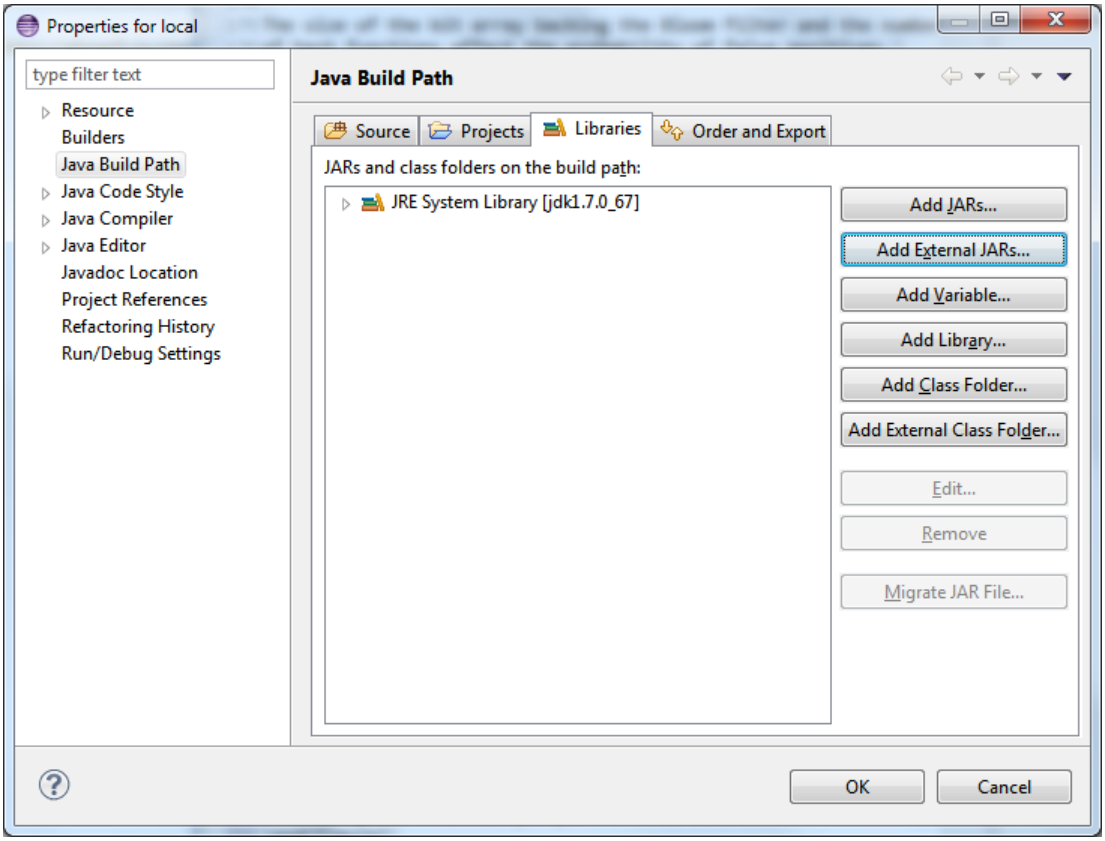

Figure 3: A screenshot of Java build path.

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 performance. 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. VisualVM can be used for memory profiling. While `System.nanoTime()` can be used for timing operations in general, the built-in timing functionality can be used when timing spell checking. 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.

There is no specific requirement for the amount of evidence you provide, as long as it is enough to justify your conclusions.

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

7 Testing

In addition to the code you write for data structures and the 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, and a JUnit tutorial has been scheduled for a lab.

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.

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 result as defined by the specification. We expect you to define traditional 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 calls to a single randomly chosen method or generate a sequence of randomly chosen method calls against an object of the tested class. This form of testing can catch bugs simply when the code fails with an exception or assertion error. An often 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. JUnit has some support for random testing in its Theories module, but use of this JUnit feature is optional.

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 simplified version of the interface:

```
1  /** A set of unique elements kept sorted in ascending order. */
2  interface SortedSet<T extends Comparable<T>> {
3      /** 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      /** Remove element x. */
10     void remove(T x);
11
12     /** Return the first element in the set. */
13     T first();
14 }
```

1. The specifications of two of these methods have a serious problem. Clearly identify the problem and write a better specification for each method that needs to be improved. You may change method signatures if you justify the change. (**Note**: Failure to produce beautifully formatted Javadoc output is not a serious problem.)

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

```
class SortedList<T extends Comparable<T>> implements SortedSet<T> {
    /* Represents: the set of all values contained in the linked list
     * starting at "head", which may be null to represent an empty
     * set.
     *
     * Invariant: the list nodes starting from "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; }
}
```

2. 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.

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> {
    /* Represents: the set of all values contained in the linked list
     * starting at "head", which may be null to represent an empty
     * set. Duplicate values may occur.
     */

    ListNode<T> head;
    ...
}
```

3. `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. (**Hint**: Since `SortedList` and `UnsortedList` implement the same specification, the client should not be able to tell which one is being used.)

4. Briefly discuss how to choose between these two implementations.

8.2 Object-Oriented Design: Assignment 2 Revisited

Look at the files `AbstractCipher.java`, `MonoAlphabeticCipher.java`, and `VigenereCipher.java` in the package `a2_sample`. Describe at least two distinct problems with the specifications and at least four distinct problems with the use of inheritance. Briefly explain how you would fix each problem. You do *not* need to determine whether any of the code actually works.

8.3 Loop invariants

In class we saw an iterative implementation of the binary search algorithm. Recall that its specification said it returned an index containing the desired element, under the precondition that the array was sorted in ascending order and the element was already in the array. We used a loop invariant to show (see the notes) that the code satisfied its specification.

In fact, the code satisfies a stronger specification: the code returns the *smallest* index `i` such that `a[i]==k`. This specification differs from the one we used in class in the case where the element appears more than once in the array.

Suppose instead that we wanted the search to always find the *largest* index `i` such that `a[i]==k`. The code we saw does not satisfy this spec.

a. Change the code of the algorithm as little as possible so that it meets this new spec.
b. What is the postcondition of the loop in your algorithm?
c. Give a loop invariant that is strong enough to show that your code satisfies the new spec.
d. Give each of the 4 parts of the argument for correctness of the revised code. Your argument should be similar to the argument we used in class.

9 KARMA

KARMA questions do not affect your raw score for any assignment. They are given as interesting problems that present a challenge.

9.1 Written problems

5. Is it true that $\sin n$ is $O(\cos n)$? Give a witness if it is true, or argue that no such witness exists otherwise.

KARMA:

6. Is it true that 2^{2^n} is $O(4^n)$? Give a witness if it is true, or argue that no such witness exists otherwise.

KARMA:

9.2 Cuckoo hashing

Cuckoo hashing is an alternative to chaining for resolving collisions in a hash table. The name derives from the behavior of a species of cuckoo, where the mother pushes eggs or nestlings out of the nest to make way for hatching new eggs[2]. Similarly, inserting a new key into a cuckoo hash table may push an older key to a different location in the table.

9.2.1 Collision resolution

In cuckoo hashing, at least two hash functions are used instead of one, providing at least two possible locations in the hash table for each key. Suppose we have a hash table T and a pair of hash functions f_1 and f_2. Insertion of an element x works as follows. First, compute $f_1(x)$, which is the location to store x. If $T[f_1(x)]$ is initially empty, the insertion is complete. Otherwise, x must displace some element y at $T[f_1(x)]$. The newly displaced y is reinserted using f_2 to compute its new location. This process continues as $T[f_2(y)]$ may also be inhabited. A risk with cuckoo hashing is that this procedure could enter an infinite loop such as $T[f_2(T[f_2(y)])] = y$. In this case, the table

[2]Description taken from Wikipedia.

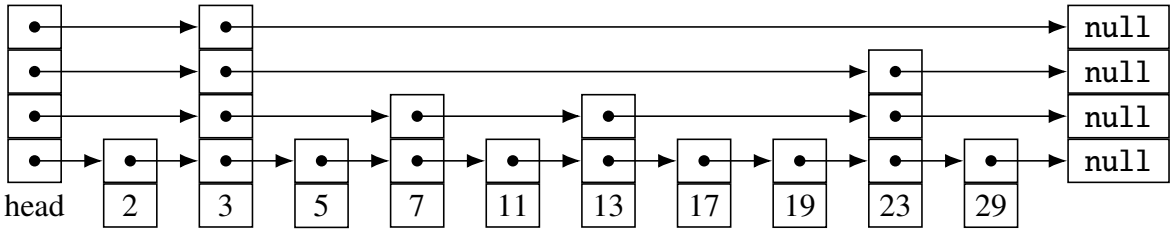

Figure 4: A skip list containing the first ten prime numbers.

is resized and all elements in it are rehashed. Further information about Cuckoo hashing can be found in this paper.

HARMA: ☘ ☘ ☘ ☘ ☘

9.3 Fast string search

While it may sound counterintuitive, searching for a pattern in a large document in sublinear time is possible. One effective algorithm for searching strings is the Boyer–Moore algorithm, which attempts to align the pattern in the document. The key idea in the algorithm is to compare characters from the end of a pattern. On a mismatch, this strategy allows the algorithm to skip over multiple characters of the document being searched, as they can never align with the pattern. The algorithm is described in the original research paper, and an interactive demonstration of the algorithm is available.

To implement fast string search, update your implementation of `SearchModule` to use Boyer–Moore or another fast string-search algorithm.

HARMA: ☘ ☘ ☘ ☘ ☘

9.4 Skip lists

A skip list is a data structure that supports fast search within a collection of ordered elements. A hierarchy of linked lists, each holding a subset of the elements of its predecessor, makes fast search possible. Searching starts in the sparest list as an express lane which bypasses many elements. A denser list—a slower lane—is used for lookups only after the express lane determines an appropriate region of the list that may contain the value. This lowers the time to find a value in the sorted list, which is linear in regular linked lists. Figure 4 shows an example of a skip list.

One challenge in designing a skip list is determining which linked lists should contain a given value. We will use a randomized approach outlined below.

This YouTube video is a helpful resource for understanding skip lists. Watching the video before diving into an implementation will make your job substantially easier.

9.4.1 Initialization

A skip list is initialized with at least one empty linked list. We need not worry about the exact number of initial linked lists as this number will grow as elements are inserted into the skip list.

9.4.2 Lookup

The search for an element x begins at the head of the highest list in the collection of linked lists, e.g., the top list in Figure 4. This list is traversed until the next node in the list is `null`, or its value is greater than x. At this point, the search continues from the current node in the lower list. The search terminates when x is found or the lowest list has been traversed.

For example, looking up 13 in the skip list in Figure 4 yields the following sequence of visited nodes, where x_y denotes the node containing value x at level y, level 0 being the bottom list:

$$[\text{head}]_3 \to 3_3 \to 3_2 \to 3_1 \to 7_1 \to 13_1$$

This traversal encounters three different values instead of six, which would result from the naïve lookup on a regular linked list.

9.4.3 Insertion

To insert an element x into a skip list, locate a pair of adjacent nodes n_1 and n_2 such that $v_{n_1} \leq x \leq v_{n_2}$, where v_n denotes the value at node n. Create a new node n' with value x and insert it into the list at level 0 between n_1 and n_2. Now we use randomization to decide whether n' should also appear in the list at level 1 by flipping a coin. If the flip is heads, n' is inserted into the list at level 1; otherwise, insertion is complete. This process continues inserting n' into lists at higher levels as long as heads is flipped. That is, the probability that n' appears in level k is 2^{-k}. If n' is to be inserted into a list at a previously nonexistent level, a new linked list is created for that level, and the head node is updated to include this new list, whose only element is n'.

9.4.4 Deletion

To delete an element x from a skip list, find a node containing x and remove it from the hierarchy of linked lists.

HARMA: 

10 Submission

You should compress exactly these files into a `zip` file that you will then submit on CMS:

- *Source code*: Because this assignment is more open than the last, you should include all source code 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.
- `README.txt`: This file should contain your name, your NetID, all known issues you have with your submitted code, and the names of anyone you have discussed the assignment with. It should also include the descriptions of any **HARMA** problems you attempted.
  In addition, you should briefly describe and justify your design, noting any interesting design decisions you encountered, and briefly discuss your testing strategy.
- `written_problems.txt/pdf`: This file should include your response to the written problems.
- `perf.txt` or `perf.pdf`: This file should include your performance analysis.

Do not include any files ending in `.class`.

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.*