We have allowed you to comment and suggest on the Google Docs (ping us on the message board if you can't) if you would like to provide any feedback (anything small or big such as a typo, grammar mistake, suggested change, etc. is welcome).

CSE 332: Data Structures and Parallelism

P2

The purpose of this project is to implement various data structures and algorithms described in class. You will also implement the back-end for a chat application called "`uMessage`".

Overview

One of the most important ADTs is the `Dictionary` and one of the most studied problems is sorting. In this assignment, you will write multiple implementations (`AVLTree`, `HashTable`, etc.) of `Dictionary` and multiple sorting algorithms.

All of these implementations will be used to drive word suggestion, spelling correction, and autocompletion in a chat application called `uMessage`. These algorithms are very similar to the ones smartphones use for these problems, and you will see that they do relatively well with a small effort. Since `uMessage` has many components and is difficult to test, we will ask you to test your code by writing another client for `WordSuggestor`.

We have provided the boring pieces of these programs (e.g., GUIs, printing code, etc.), but you will write the data structures that back all of the code we've written.

Project Restrictions

- You *must* work in a group of two unless you petition to work by yourself.
- You may not use **any** of the built-in Java data structures. One of the main learning outcomes is to write everything yourself.
  - Specifically, do not use a `HashSet`, `HashMap`, `LinkedHashMap`, `LinkedHashSet`, `TreeSet`, `TreeMap`, `ArrayList`, `LinkedList`, `PriorityQueue`, etc.
  - If it feels like you are skipping a lot of steps with anything imported from Java, you probably shouldn't use it.
- You may use the `Math` package.
- You may not edit any file in the `cse332.*` packages.
- You may not edit any of the public interfaces.
- The *design* and *architecture* of your code are a *substantial* part of your grade.
- The Writeup is a *substantial* part of your grade; do **not** leave it to the last minute.
- Make sure to not duplicate fields that are in super-classes (e.g. `size`). This will lead to unexpected behavior and failures of tests.

P1 and Beyond

This project actually extends on p1 a lot! You will need to overwrite the following with your p1 code:

- `datastructures.worklists`: All your simple `WorkList`s: `ArrayStack`, `ListFIFOQueue`, `CircularArrayFIFOQueue`
- `datastructures.dictionaries`: Your `HashTrieSet` and your `HashTrieMap`

Be sure you do NOT place these in `cse332.datastructures.worklists`. After you port these files over, `CircularArrayFIFOQueue` won't compile. It defines a type parameter `E` in `CircularArrayFIFOQueue<E>` at the top of the class, but you should replace this `E` with "`E extends Comparable<E>`".

Provided Code

- `cse332.interfaces.misc`
  - `DeletelessDictionary.java`: Like a `Dictionary`, but the `delete` method is unsupported.
  - `ComparableDictionary.java`: A `DeletelessDictionary` that requires comparable keys.
  - `SimpleIterator.java`: A simplification of Java's `Iterator` that has no remove method.
- `cse332.datastructures.*`
  - `Item.java`: A simple container for a key and a value. This is intended to be used as the object stored in your dictionaries.
  - `BinarySearchTree.java`: An implementation of `Dictionary` using a binary search tree. It is provided as an example of how to use function objects and iterators. The iterators you write will not be as difficult
- `cse332.*`
  - `WordReader.java`: Standardizes inputs into lower case without punctuation.
  - `LargeValueFirstItemComparator.java`: A comparator that considers larger values as "smaller", and breaks ties by considering the keys.
  - `InsertionSort.java`: A provided implementation of `InsertionSort`.
  - `AlphabeticString.java`: This type is a `BString` that is just a wrapper for a standard `String`.
  - `NGram.java`: This type is a `BString` that represents an n-gram
- `p2.wordcorrector`
  - `AutocompleteTrie.java`: This is the trie used by `uMessage`; it is backed by `HashTrieMap`.
  - `SpellingCorrector.java`: This is the spelling corrector used by `uMessage`
- `p2.wordsuggestor`
  - `NGramToNextChoicesMap.java`: Client data structure that will be used to drive `WordSuggestor`.
  - `ParseFBMessages.java`: This program downloads your facebook messages. It is intended to be used as a way of generating a personal corpus for the `WordSuggestor`. There are more instructions for using this in the writeup spec
  - `WordSuggestor.java`: This is the word suggestor used by `uMessage`.
- chat
  - `uMessage.java`: This is the main driver program for `uMessage`.

You will implement data structures `MinFourHeap` and `MinFourHeapComparable`, `MoveToFrontList`, `AVLTree`, and `ChainingHashTable` and sorting algorithms `HeapSort`, `QuickSort`, and `TopKSort`.

`uMessage`

After you have finished all the implementations, you will be ready to try out `uMessage`. We expect you to actually play with the application, and the Writeup will ask you to do several things with it. Importantly, there are configuration settings (n and the corpus) at the top of `uMessage.java` which you will want to edit.

Project Checkpoints

This project will have **two** checkpoints (and a final due date). A *checkpoint* is a check-in on a certain date to make sure you are making reasonable progress on the project. For each checkpoint, you (and your partner) will turn in a survey individually.

As long as you turn in the checkpoint survey, it will not affect your grade in any way.

Checkpoint 1:	**(1), (2), (3) due**
Checkpoint 2:	**(4), (5), (6), (7) due**
P2 Due Date:	**(8), (9) due**

Part 1: Another `WorkLists`

First, implement one more `WorkList`:

(1) `MinFourHeapComparable`

Your `MinFourHeapComparable` should be an implementation of the heap data structure we've discussed in class. It should be an array-based implementation which starts at index 0. Unlike the implementation discussed in lecture, it should be a *four-heap* (not a two-heap). In other words, each node should have *four* children, not two. All operations should have the efficiencies we've discussed in class:

Operation	Worst-case Runtime
Peek min	$\mathcal{O}(1)$
Delete min	$\mathcal{O}(\log n)$
Insert	$\mathcal{O}(\log n)$ (amortized)
...	...

Note: Take a look at the generics handout when initializing the array in the `MinFourHeapComparable` data structure.

Part 2: Implementing `Dictionary` Classes and Sorts

(2) MoveToFrontList: Another `Dictionary`

In this part, you will implement `MoveToFrontList`, a new type of `Dictionary`.

For the remainder of the `Dictionary` classes you will implement, we will not ask you to write delete–it is possible (and you can do it for fun), but it's not educational enough to be part of the actual project. As a result, your `Dictionary` classes will inherit from `DeletelessDictionary` which is the same as `Dictionary` except it does not require that you implement a delete method.

`MoveToFrontList` is a type of linked list where new items are inserted at the front of the list, and an existing item gets moved to the front whenever it is referenced. Although it has $\mathcal{O}(n)$ worst-case time operations, it has a very good amortized analysis.

You will also be implementing an `Iterator` for this `Dictionary`. The runtime for all `Iterator` operations should run in $\mathcal{O}(1)$. We will not be discussing iterators in class so if you need, you can reference the `Iterator` for `BinarySearchTree`.

(3) CircularArrayFIFOQueue

You might have noticed that we didn't finish implementing all the methods in `CircularArrayFIFOQueue`. If you look in `BString`, it relies on `CircularArrayFIFOQueue` having a reasonable definition of equality. In Java, we deal with this by defining an `equals` method. You may not use `toString` to implement `equals`; we expect you to build it from scratch. You might be wondering how to figure out the type of the parameter for `equals`; in Java, the `equals` method takes an Object. You will want to do research on the Java `instanceof` operator, as it will be a part of your solution.

In addition to equality testing, we also need to be able to *compare* two Objects. To do this, you should complete the `compareTo` method in `CircularArrayFIFOQueue`. You may not use `toString` to implement `compareTo`; we expect you to build it from scratch. `CircularArrayFIFOQueue` backs `BString` so `compareTo` should work the same as `String`'s `compareTo`.

The reason we implement this is that our *tree* dictionaries in the next part will need to be able to do comparisons instead of equality testing. Remember, in any `Dictionary` implementation, you may use any of your `WorkList` implementations

(4) `AVLTree`: Another Another `Dictionary`

In this part, you will implement `AVLTree`. Just like before, you do not have to implement `delete`. Your `AVLTree` should be a subclass of `BinarySearchTree` which we have written for you. You should use an array implementation of left and right children as in BinarySearchTree. Your `insert(K key, V value)` should run in $\mathcal{O}(\log(n))$. Ensure your rotation code is not repetitive and runs in $\mathcal{O}(\log(n))$.

A note on `AVLTree` Inheritance. `AVLTree` extends BinarySearchTree, and BinarySearchTree has a couple methods we might think could be useful: `find(K key, V value)` and `find(K key)`. Some of you may be trying to use the former `find(K key, V value)` to access the appropriate spot in your tree without duplicating code, but there's actually an issue with this: `find(K key, V value)` puts `BSTNodes` in your `AVLTree` and returns them to you. These nodes can't be cast to `AVLNode` (because they were initialized as `BSTNodes`), and, since they are `BSTNodes`, they don't have that all-important height field, so you can't use them.

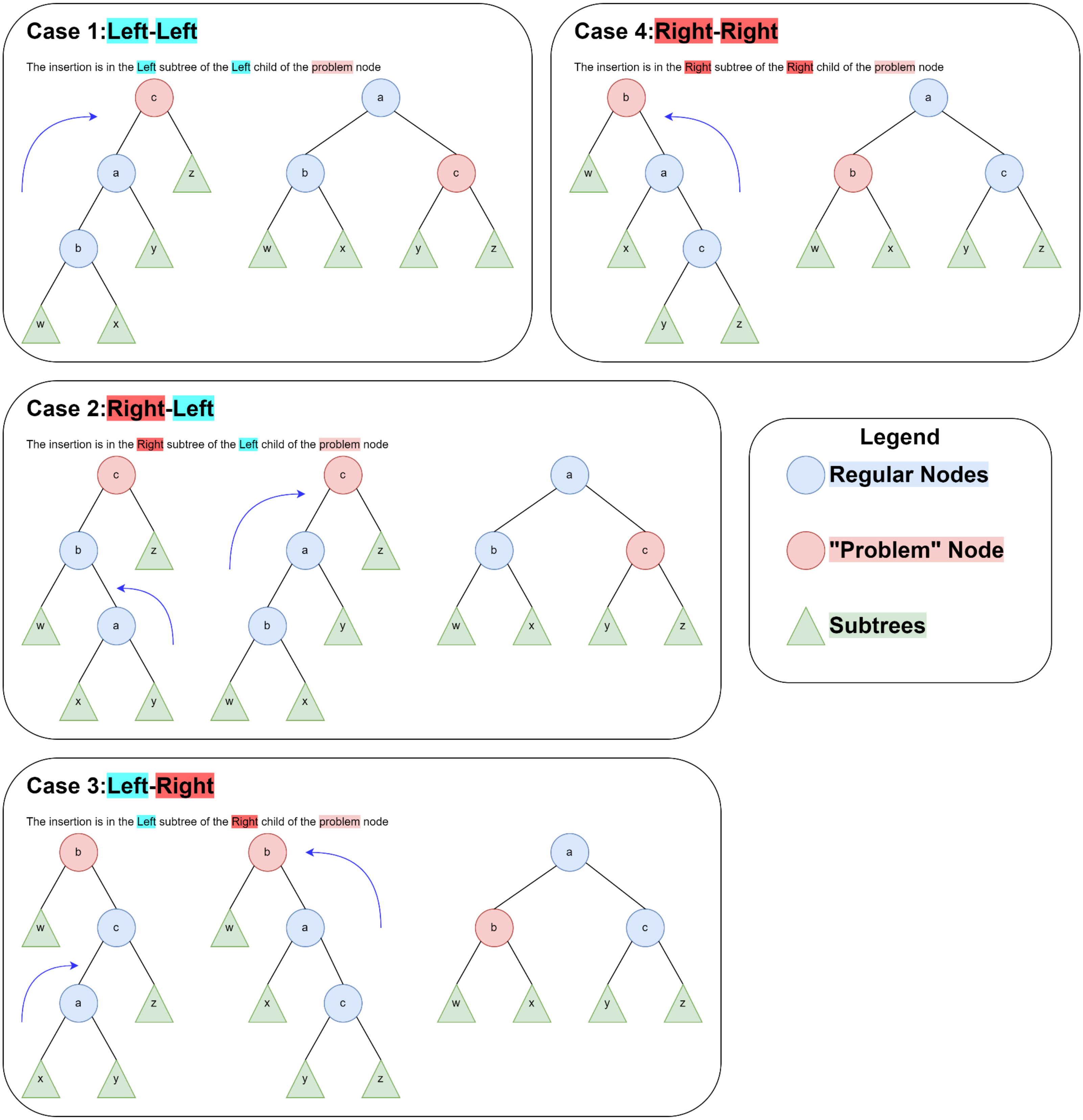
In other words, you should not call the `find(K key, V value)` method (with a non-null second argument) in `BinarySearchTree` as part of your `insert` method. It's okay if you end up duplicating some of the `find(K key, V value)` logic in your `insert` method.

You will not need to write a separate `find`(`key`) method, though, since the behavior of that method will be the same for both tree types, meaning that the inherited method already behaves correctly.

Recall that all BSTs rely on a reasonable definition of comparison. Our BST and your `AVLTree` will both rely on the `compareTo` that you wrote in the previous part.
A note on debugging. You can "fail fast" by adding your `verifyAvl` code as a private helper method, checking validity after every modification to the tree, and throwing an exception if the check fails. This will help you identify which sections of the code are breaking the tree. These checks will be expensive and should be disabled in the final version, but can be helpful when debugging.

Although we do not check your style (such as in the introductory courses), you should still plan your data structure and write many good quality comments for **yourself and your partner**. This is especially true for AVLTree, where we see many students stumble. Plan your data structures now and you can avoid stressful debugging later. If you don't put bugs in, you don't have to take them out.

Here is a nice diagram on all the rotation cases:



(5) `ChainingHashTable`: Another Another Another `Dictionary`

In this part, you will implement `ChainingHashTable`. Just like before, you do not have to implement delete. Your hash table must use separate chaining–not probing. Furthermore, you must make the type of chain generic. In particular, you should be able to use *any* `Dictionary` implementation as the type inside the buckets. Your `HashTable` should rehash as appropriate (use an appropriate load factor as discussed in the class), and its capacity should, for a constant amount of cases, be a prime number. Your `HashTable` should be able to work with `uMessage` which means there shouldn't be a hard cap on how much it can grow; though, it doesn't have to use primes past 200,000.

Pick a reasonable starting size for your `HashTable`. You should use a hardcoded list of primes to resize up to 200,000 (make it a constant in the `ChainingHashTable.java` file). Do not hard code every prime up to 200,000 (i.e., you don't want to resize to just the next prime number, but the next prime number that provides ~twice the original capacity). After this point, you should continue to resize your table using some other mechanism. Note that you MUST GROW the table past 200,000.

Recall that all Hash Tables rely on a reasonable definition of hash code. Just like you needed to define `equals` and `compareTo` for various other data structures, you will need to define `hashCode` in `CircularArrayFIFOQueue` for `ChainingHashTable`. You may not use `toString` to implement `hashCode`; we expect you to build it from scratch.

At some point, you will want to test various types of chains in your `ChainingHashTable`. It is confusing to do this initially; so, we have provided some examples in the `NGramTester` class.

(6) HashTrieMap: Full Circle!

Now that you have written your own hash map, replace the dependency on Java's HashMap with your `ChainingHashTable`! This is not only okay, it's a great example of unexpected refactoring. Refactoring will usually set off a chain reaction where you also have to edit other code.

You will want to look at the [SimpleEntry javadoc] (you are allowed to import and use the SimpleEntry class). Remember that you may edit any class that is not in a `cse332`.* package.

Here is a general guide on what to change:

- First, work on refactoring HashTrieMap. Red squiggly lines should start appearing in IntelliJ, and you can follow those compiler errors as guidance of what to fix next.
- You will eventually need to fix `AutocompleteTrie.java` as part of your refactor.
- Some methods might become impossible to implement with `ChainingHashTable`. In this case, it is okay to throw a `UnsupportedOperationException`.
- You will notice a mismatch between the type of `Iterator` returned from `ChainingHashTable` and the one that you need in `HashTrieNode`. This is an example of a common issue you run into while refactoring code
    - You'll need to add a (small) bit of code to `HashTrieNode`/`HashTrieMap` to work around this type mismatch. You can do it using what you've already learned about iterators.
    - Note that you shouldn't modify the `ChainingHashTable.iterator()` return type, because then it wouldn't match the `Dictionary` interface, and you also shouldn't add superfluous `Iterator` methods to `ChainingHashTable` to solve a problem in `HashTrieMap`.

You have now written pretty much all of the data structures that you've used from Java's library! You now understand all the magic under the hood! Take a minute to bask in the glory that is data structures nirvana.

(7) `MinFourHeap` and The Sorts

The `MinFourHeapComparable` you wrote before was only able to compare elements in a single way (based on the `compareTo`). There is a more general idea called a `Comparator` which allows the user to specify a comparison *function*. The first thing you should do in this part is implement `MinFourHeap` to use a `Comparator`. You can copy the logic from `MinFourHeapComparable` and modify the logic to use a `Comparator` instead of `compareTo`. Make sure to modify the array type such that it no longer extends `Comparable`. This is necessary to make the `sort`s (below) work.

NOTE: Take a look at the generics handout when initializing the array backing the `MinFourHeap` data structure

After you've edited `MinFourHeap`, you will be ready to write the following sorting algorithms:

- `HeapSort`: Consists of two steps:

  - Insert each element to be sorted into a heap (`MinFourHeap`)
  - Remove each element from the heap, storing them in order in the original array.
- `QuickSort`: Implement quicksort. As with the other sorts, your code should be generic. Your sorting algorithm should meet its expected runtime bound.
- `TopKSort`: An easy way to implement this would be to sort the input as usual and then just print k largest of them. This approach finds the k largest items in time $\mathcal{O}(n \log(n))$. However, your implementation should have $\mathcal{O}(n \log(k))$ runtime, assuming k is less than or equal to n. `TopKSort` should put the top k elements in the first k spots in the array, and **all the other indices should be null**. In other words, if $A = Quicksort(B)$ for some array B, then:
  $TopKSort(k,\ A) = [A[n-k],\ A[n-(k-1)], \ldots,\ A[n-1],\ null,\ null, \ldots,\ null]$.

  Hint: There are many ways to go about TopKSort, but the key idea is to use a heap and never put more than k elements into it. Think about why this gives $\mathcal{O}(n \log(k))$ runtime bound!

Part 3: `uMessage` and The Writeup

(8) `uMessage` - Do not wait until the last minute for this!

Now that you are done with all of the coding (and most of the Writeup) for the project, you are ready to attempt to run `uMessage`. As many folks saw when they ran zip on P1, this may expose problems with code you wrote earlier. Do not wait until the last minute for this step!

Note: `uMessage` connects to a live server and you will be able to talk to others who are working on the assignment at the same time. When using `uMessage`, our course policy requires that you use your CSE or UWNetID as your username. This is a fun program to play around with (please do!) but anyone found using the system to annoy or harass others will be referred to the appropriate conduct offices.

Before you run `uMessage`, you will want to do the following:

- Increase the allowed heap size in IntelliJ. In particular, `uMessage` runs significantly more smoothly if you give it **6144MB (6GB)** of memory.
- Make sure your laptop is plugged in (Yes, this will make a difference.)
- Finish the `getWordsAfter` method in `NGramToNextChoicesMap` (see next section). You should replace InsertionSort with a faster, standard sort, and if $k \geq 0$, you should run TopKSort. You might have to do something more than just run TopKSort to get the most frequent words out. Figuring out exactly what to do here is part of the challenge. Note: you will need to handle the case where the array length is 0.

There are several variables at the top of `uMessage` which you will have to edit: the corpus, the "n", the "inner dictionary" and the "outer dictionary". If you leave the corpus as eggs.txt, the suggestions will be garbage. If you leave the inner and outer dictionaries as tries, `uMessage` will probably be too slow. The point of `uMessage` is that it is a cool application that uses all of the code you wrote. Just like Zip was a good stress test for P1, `uMessage` is a good stress test for P2.

Once you start working on `uMessage`, if you've implemented `getWordsAfter` correctly, the word suggestions you get in `uMessage` should be sorted by frequency (conditioned on the previous words), with highest frequency on the left and lowest frequency on the right. Note that inputs with apostrophes may not work, (e.g. can't, wouldn't), and throw an `SSLPeerUnverifiedException`. This is a bug in `uMessage` and you can just ignore it :)

As a simple example, with `irc.corpus`, the words suggested as first words on a newly-opened chat with nothing typed should be ["i", "and", "yeah", "well"], in that order, since those are the four words with the highest frequency at the start of a line, with "i" being the most frequent of the four.

Trying to debug issues with your ordering code on `irc.corpus` will take a long time (since this corpus takes a while to load), so it might be a good idea to make a simple test corpus with only a few sentences where you can work out what the suggested words should be, and using that to quickly figure out `getWordsAfter`.

NGramToNextChoices

`NGramToNextChoicesMap` is a file you will need to work with to complete your project, so let's introduce it real quick. `NGramToNextChoicesMap` will map `NGrams` to **words** to **counts**.

Let's walk through an example to better understand this. Suppose that the `n` in `n-gram` is **2** and the following are the contents of our input file:

```
Not in a box.
And not in a house.
```

The key set of the **outer map** will contain all of the 2-grams in the file. That is, it will be:
`{"not in", "in a", "a box", "box SOL", "SOL and", "and not"}`

Notice several interesting things about the output:

- All input is standardized by removing non- alphanumeric characters converting everything to lowercase
- The word `SOL` has been added at the beginning of every line except the first one. `SOL`, which stands for "start of line", is inserted by uMessage so that individual pieces of the corpus do not get mushed together.
- "a house" does not appear in the outer map; the reason for this is that there is nothing after it to include!

The "top level" maps to another dictionary whose keys are the possible words following that n-gram. So, for example, the keys of the dictionary that "in a" maps to are {"box", "house"}, because "box" and "house" are the only two words that follow the 2-gram "in a" in the original text.

Finally, the values of the inner dictionary are a count of how many times that word followed that n-gram. So for example, we have:

- `"not in"={a=2}`, because the word `a` follows the 2-gram `not in` twice
- `"in a"={box=1, house=1}`, because `box` and `house` each only appear once after "in a"

The entire output for the sample input file above looks like:

```
"not in"={a=2}, "in a"={box=1, house=1}, "a box"={SOL=1}, "box
SOL"={and=1}, "SOL and"={not=1}, "and not"={in=1}
```

The order of the entries does not matter (remember, dictionaries are not ordered), but the contents do.

Performance of Implementations of Dictionary

Part of this project is comparing and contrasting the performance of various implementations of Dictionary. To do this, we will use different outer and inner `Dictionary` types in `NGramToNextChoicesMap`. The outer type is the map from `NGrams` to words; the inner type is the map from words to counts.

To make this easier, `NGramToNextChoicesMap` takes two initializers in its constructor representing these types. For example, to initialize an NGramToNextChoicesMap with the outer dictionary as `ChainingHashTable` and inner dictionary as `MoveToFrontList`, we would write:

```
new NGramToNextChoicesMap(() -> new ChainingHashTable(), () -> new MoveToFrontList());
```

The `() -> X` notation tells Java to make a function that takes no arguments and returns the thing on the right.

Instead of using `String` for the **words** in `NGramToNextChoicesMap`, we will use the type `AlphabeticString` so that we can use `TrieMap` if possible. Note that `AlphabeticString` extends `CircularArrayFIFOQueue`, so the `hashCode()` and `equals()` method of your `CircularArrayFIFOQueue` can affect your `NGramToNextChoicesMap`. **Hint: This is the first place to look for bugs when uMessage does not work as expected.**

(9) Writeup

Approximately half of your grade will be based on your Writeup. The analysis part of this project is incredibly important, and we expect you to spend an entire week's worth of work on it. You will find the Writeup questions in the P2 Writeup Template on the website. Remember to follow the instructions on the first and second page to receive full credit!

DO NOT MIX any of your experiments or above and beyond files with the normal code. Before changing your code for experiments or above and beyond, copy the relevant files into the corresponding package (e.g., `aboveandbeyond`, `experiments`). If your code does not compile because you did not follow these instructions, you will receive a 0 for all automated tests.

Make sure your experiment code actually makes it onto Gradescope. Your folder name should be exactly src/main/java/experiments/, otherwise it will not be included in your Gradescope submission. When you submit your code on Gradescope, go to the "Code" tab and double check the list of files that are submitted.

Submission and Grading

Submission instructions can be found on the Handouts page of the website. We will grade based on correctness of code (**given** tests + **hidden** tests) and **manual grading** on whether you are following the spec. There will be **no grading** on style (such as in the introductory course) but we suggest writing comments to help yourself understand your code and to help us grade your implementation.