

P2: uMessage**Checkpoint 1:** Due Tue, January 29**Checkpoint 2:** Due Tue, February 12**P2 Due Date:** Due Tue, February 19

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 successfully 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.
- You may use the `math` package.
- You may not edit any file in the `cse332.*` packages.
- The *design and architecture* of your code are a *substantial* part of your grade.
- The Write-Up is a *substantial* part of your grade; do **not** leave it to the last minute.
- **DO NOT MIX** any of your experiment 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 to not duplicate fields that are in super-classes (e.g., `size`). This will lead to unexpected behavior and failures of tests.

NGrams and Generating Text

An NGram is a list of n words appearing in order in a text. They are often used in textual analysis to see how frequent patterns are. In this assignment, you will use them to generate *new text* that sounds like the author of *an original text*. This type of text generation is how word prediction works on your phone.

There are two main backend programs that drive uMessage: WordSuggestor and SpellingCorrector. We recommend you only attempt to run uMessage directly when you believe you no longer have any bugs.

P1 and Beyond

This project actually extends on p1 a lot! You will need to port over (i.e., put them in the same packages) the following:

- `datastructures.worklists`: All your simple WorkLists: `ArrayStack`, `ListFIFOQueue`, `CircularArrayFIFOQueue`
- `datastructures.worklists`: Your `MinFourHeap` (Note that it will not immediately compile, because the interfaces have changed slightly—more on that later.)
- `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

Several of the interfaces and implementations from p1 also appear in p2. We will only describe the *new* classes in an attempt to be less overwhelming.

- `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`
 - `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`.
- `p2.clients`
 - `NGramTester.java`: This class can be used to test your `NGramToNextChoicesMap`.
- `chat`
 - `uMessage.java`: This is the main driver program for `uMessage`.

You will implement `NGramToNextChoicesMap` (in `p2.wordsuggestor`), `MoveToFrontList`, `AVLTree`, and `ChainingHashTable` (in `datastructures.dictionaries`), `HeapSort`, `QuickSort`, and `TopKSort` (in `p2.sorts`).

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 Write-Up 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. It is very likely that you will need to read the “out of memory” handout as you do this.

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 sign up for a 10-minute time slot during which you will meet with a staff member and discuss where you are on the project.

As long as you show up to a time-slot and you do not miss multiple checkpoints in a row, the checkpoint will not affect your grade in any way.

Checkpoint 1: (1), (2)

Tue, January 29

Checkpoint 2: (3), (4), (5), (6)

Tue, February 12

P2 Due Date: (7), (8)

Tue, February 19

Part 1: A Dictionary Client & A new Dictionary

Perhaps confusingly, you will begin by writing the *client data structure* that will use all of your code. This data structure is called `NGramToNextChoicesMap`. We have written part of it for you, but we’re asking you to implement most of this data structure so you become familiar with the expected behavior of the data structures you will be writing later.

One skill that you will need to pick up over your career is learning new APIs; to help you with this, we have (without significant explanation) used a few Java 8 features. In particular, you will want to look up the `Supplier` class. Although it is overkill, parts of [this tutorial](#) are helpful.

(1) `NGramToNextChoicesMap`

Before continuing, it is *imperative* that you understand what an `NGram` is.

The *very general idea* of `NGramToNextChoicesMap` is the following:

`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.  
>> Not with a fox.  
>> Not in a house.  
>> Not with a mouse.
```

The key set of the outer map will contain all of the 2-grams in the file. That is, it will be

```
{“box SOL”, “house SOL”, “in a”, “a fox”, “a house”, “with a”, “not with”, “fox SOL”, “a box”, “not in”, “SOL  
not”}
```

Notice several interesting things about the output: (1) all input is standardized by removing non-alphanumeric characters converting everything to lower case, and (2) 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 mashed together. Furthermore, note that “a mouse” 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 “with a” maps to are {“mouse”, “fox”}, because “mouse” and “fox” are the only two words that follow the 2-gram “with 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
- "with a"={mouse=1, fox=1}, because “mouse” and “fox” each only appear once after “with a”

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

```
"SOL not"={in=1, with=2}, "a box"={SOL=1}, "a fox"={SOL=1}, "a house"={SOL=1},  
"box SOL"={not=1}, "fox SOL"={not=1}, "house SOL"={not=1}, "in a"={box=1, house=1},  
"not in"={a=2}, "not with"={a=2}, "with a"={fox=1, mouse=1}
```

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

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 use outer = ChainingHashTable and inner = 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. This is handy, because our NGramToNextChoicesMap needs to be able to create new inner maps for each key in the outer map.

One more important implementation detail is that instead of using type “String” for the words, we use type “AlphabeticString”. The reason for this should be clear: we’d like to use TrieMap if possible!

To use a HashTrieMap, we need to jump through a few extra hoops, because the constructor takes an extra argument. We’ve provided a method for you in NGramTester called trieConstructor which does this for you; it returns a Supplier which can be given directly to WordSuggestor.

Now that you know what NGramToNextChoicesMap is supposed to do, implement the following two methods:

```
public void seenWordAfterNGram(NGram ngram, String word)
```

```
Increments the number of times that word has been seen after ngram
```

```
public Item<String, Integer>[] getCountsAfter(NGram ngram)
```

```
Returns an array of Items representing words and the number of times each word was seen after ngram. There is no guarantee on the ordering of the array.
```

There is a third method relevant to word suggestion called getWordsAfter which we have partially implemented for you, but, for now, you should not implement it.

We recommend testing your implementation by using HashTrieMap since you already have one that works.

(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 extra credit), 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. We will not discuss this data structure in class.

MoveToFrontList relies on *equality* testing of elements. In Java, we deal with this by defining an equals method. If you look in BString (the class that AlphabeticString and NGram both inherit from), it relies on CircularArrayFIFOQueue having a reasonable definition of equality. Before MoveToFrontList will work, you will need to define the equals method for CircularArrayFIFOQueue. 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.

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.

Part 2: Implementing The Remaining Dictionary Classes and Sorts

(3) AVLTree: Another Another Dictionary

In this part, you will implement AVLTree. We recommend waiting to do this until we have discussed it in lecture. Just like before, you do not have to implement delete. Your AVLTree should be a sub-class of BinarySearchTree which we have written for you. Be careful to not duplicate code in rotation. You should use an array implementation of left and right children as in BinarySearchTree. Additionally, if your rotation code is repetitive, you will lose a substantial amount of points.

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(key, 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 verify-avl 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.

The [Eclipse visualizer plugin \(part 2 of the Eclipse document\)](#) can help while running through simple

test cases of single/double-rotation methods to check that there are no incorrect sizes, stray pointers, etc. visually. The [Eclipse display window](#) can help by interactively build a tree while debugging, and thus visualize the results.

(4) ChainingHashTable: Another Another Another Dictionary

In this part, you will implement `ChainingHashTable`. We recommend waiting to do this until we have discussed it in lecture. 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 always 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. Do not hard code every prime up to 200,000 - just pick a reasonable range of prime numbers. After this point, you should continue to resize your table using some other mechanism. Note that you **MUST GROW** the table past 200,000. It's ok if you just double the size of your table when you re-hash!

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.

(5) HashTrieMap: Full Circle!

Now that you have written your own hash map, replace the dependency on Java's `HashMap` with your `ChainingHashTable`!

You will want to look at [the `SimpleEntry` javadoc](#).

You will likely have to add or change an iterator method to make this work correctly. This is not only okay, it's a great example of unexpected refactoring. This may set off a chain reaction where you also have to edit other code. Remember that you may edit any class that is not in a `cse332.*` package.

You will need to fix `AutocompleteTrie.java` as part of your refactor in part 5.

While implementing the `HashTrieMap`, 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 the sort of 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.

(6) MinFourHeap (Again?) and The Sorts

The MinFourHeap you wrote in p1 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 edit your MinFourHeap to use a comparator. You should edit the constructor to take a Comparator<E> and the rest of your code to use that comparator in place of compareTo. This is necessary to make the sorts (below) work.

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

- HeapSort: Consists of two steps:
 - (1) Insert each element to be sorted into a heap (MinFourHeap)
 - (2) 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 $\mathcal{O}(n \lg n)$. However, your implementation should have $\mathcal{O}(n \lg k)$ runtime, assuming k is less than or equal to n . Efficiently tracking the k largest will require a different comparator than what you used in HeapSort. 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 $A = \text{quicksort}(B)$ for some array B , then: $\text{topKSort}(k, A) = [A[n - k], A[n - (k - 1)], \dots, A[n - 1], \text{null}, \text{null}, \dots, \text{null}]$.

Notice that inside NGramToNextChoicesMap, when you use TopKSort you will have to use a different comparator than you used in HeapSort and you will need to modify the result returned from the sort.

(Hint: Use a heap, but never put more than k elements into it. Think about why this gives $\mathcal{O}(n \lg k)$ runtime bound).

Part 3: The Write-Up

(7) Write-Up

Approximately half of your grade will be based on your write-up. The analysis part of this project is incredibly important, and we expect you to spend an entire week's worth of work on it.

Some of the write-up questions will ask you to design and run some experiments to determine which implementations are faster for various inputs. Answering these questions will require writing additional code to run the experiments, collecting operation counting or timing information and producing result tables and graphs, together with relatively long answers. Do not wait until the last minute! We will post more information about the difference between operation counting or timing.

Insert tables and graphs into your write-up as appropriate, and be sure to give each one a title and label the axes for the graphs. **IMPORTANT: Place all your operation counting or timing code into the package experiment.** Be careful not to leave any write-up related code in the normal files. To prevent losing points due to the modifications made for the write-up experiments, you should copy all files that need to be modified for the experiments into the package experiment, and start working from there. Files in different packages can have the same name, but when editing be sure to check if you are using the correct file! If your code does not compile because you did not follow these instructions, you will receive a 0 for all automated tests.

You will need to write a second hashing function. To exaggerate the difference between the two hash functions, you will want to compare a very simple hash function with a decent one (the one used in Part

2). For all experimental results, we would like to see a detailed interpretation, especially when the results do not match your expectations.

(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 write-up) 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: When using uMessage, our course policy requires that you use your CSE or UWNNetID as your username. Using the system with a pseudonym or anything other than your netID is grounds for failing the assignment. 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 university authorities.

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

- Make sure JavaFX is installed correctly. If you are in the labs, it should work on all machines, but you might need to follow extra instructions. Ask on Piazza. If it doesn't work on your personal machine, you will want to (1) make sure you are using Java 8, and (2) attempt to use a lab machine if it still isn't working.
- Increase the allowed heap size in Eclipse. In particular, uMessage runs significantly more smoothly if you give it **6GBs** of memory. To do this, read the ["out of memory" handout](#) on the course website.
- Make sure your computer is plugged in. (Yes, this will make a difference.)
- Finish the `getWordsAfter` method in `NGramToNextChoicesMap`. 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.

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.

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`.

Above and Beyond

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

- *Completing the ADT*: Implement the delete methods for *all* of the Dictionary classes.
- *Alternate Hashing Strategies*: Implement both closed and open addressing and perform experimentation to compare performance. Also, design additional hashing functions and determine which affects performance more: hashing cost, collision-avoidance cost, or your addressing strategy.
- *Introspective Sort*: Introspective sort is an unstable quicksort variant which switches to heapsort for inputs which would result in a $\mathcal{O}(n^2)$ running-time for normal quicksort. Thus, it has an average-case and a worst-case runtime of $\mathcal{O}(n \lg n)$, but generally runs faster than heapsort even in the worst case. Implement `IntrospectiveSort`, and give a sample input which would result in a quadratic runtime for normal quicksort (using a median-of-3 partitioning scheme).
- *Alternate Text Generation Models*: The n -gram model is relatively simple and has some major drawbacks. You can do more interesting things instead. For example, you might use a part-of-speech tagger to get the sentences to at least always be coherent. Research more interesting text generation strategies, implement them, and discuss your results.