CSE 332: Data Structures and Parallelism — Autumn 2016

P2: `uMessage`

Checkpoint 1: Due Tue, Oct 25
Checkpoint 2: Due Tue, Nov 01
P2 Due Date: Due Tue, Nov 08

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 `ADT`s 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 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 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.

`NGram`s 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

After you port these files over, some of them won't compile. You will need to modify (slightly) many of the classes you ported over from P1. In particular, CircularArrayFIFOQueue, HashTrieMap, and HashTrieSet all have a type parameter either called E or A. To implement compareTo, you need to insist that this type be Comparable<A>. That is, if the type parameter is "A", you should replace it with "A extends Comparable<A>" in the class header. An example is with the class header of MinFourHeap.

You will also need to add a compareTo method stub. For now, you should not implement this method; you will need to implement it correctly as part of (3).

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.
  - 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 5-minute time slot during which you will meet with Ruth or a TA 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), (3)	**Tue, Oct 25**
Checkpoint 2: (4), (5), (6)	**Tue, Nov 01**
P2 Due Date: (7), (8)	**Tue, Nov 08**

Part 1: A `Dictionary` Client & two new `Dictionary` classes

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 line of the corpus*. "SOL", which stands for "start of line", is inserted by `uMessage` so that individual pieces of the corpus do not get mushed together.

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 `Item`s 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 `CircularArrayQueue`. 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 to do research on the Java `instanceof` operator, as it will be a part of your solution.

(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. Additionally, if your rotation code is repetitive, you will lose a substantial amount of points.

Recall that all BSTs rely on a reasonable definition of comparison. Just like you needed to define `equals` for

`MoveToFrontList` (because it needed equality), you will need to define `compareTo` in `CircularArrayFIFOQueue` for `BinarySearchTree` and `AVLTree` to work (because they need comparison). `compareTo` on `CircularArrayFifoQueue` should work similar to how comparison on strings works. However, you may not use `toString` to implement `compareTo`; we expect you to build it from scratch.

Part 2: Implementing The Remaining `Dictionary` Classes and Sorts

(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 grow to at least 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.

(5) `HashTrieMap`: Full Circle!

Now that you have written your own hash map, replace the dependency on Java's `HashMap` with your `ChainingHashTable`! Warning! This part is harder than it looks. Please ask if you have questions about Java, although the first thing we will tell you is to take a look at the `SimpleEntry` javadoc.

In fact, you have now written pretty much all of the data structures that you've used from Java's library! WHOA! You now understand all the magic under the hood! Take a minute to bask in the glory that is data structures nirvana. Go for a walk, have a snack, talk to a friend, play with a cat/dog, call your mom - choose your own form of celebration. Yay you! (and your partner)! Bravo! (Notice how confetti and fireworks are sprouting from this document as you read this.)

(6) `MinFourHeap` (Again?) and The Sorts

O.k. back to work. 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 $A = \texttt{quicksort}(B)$ for some array B, then: $\texttt{topKSort}(k, A) = [\texttt{A}[n-k], \texttt{A}[n-(k-1)], \ldots, \texttt{A}[n-1], \texttt{null}, \texttt{null}, \ldots, \texttt{null}]$.

  Notice that you will have to modify the result returned from `TopKSort` when using it inside `NGramToNextChoicesMap`; you should do this *outside of* your `TopKSort` code.

  (**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

A very large fraction 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*. Really!

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 timing information and producing result tables and graphs, together with relatively long answers. Do not wait until the last minute!

Insert tables and graphs into your repository as appropriate, and be sure to give each one a title and label the axes for the graphs. Place all your 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, we recommend that you 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!

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.

We will treat all *proprietary formats* as unreadable files. Please do not give us xls, xlsx, doc, docx, odt, etc. Directly embedding the data in the markdown file is your best option; images for graphs are fine; csv files are fine. It is important to realize that other people will not necessarily have the programs you do–and, perhaps more importantly, they might be running on a machine with no GUI.

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

- Make sure `JavaFX` is installed correctly on whatever machine you are using. On the windows machines in the labs this will work automatically. On the Linux machines in the labs you will need to type: `. ojdk` (that is a dot, followed by a space, followed by ojdk) at a command prompt before running uMessage. 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's a cool application that uses all of the code you wrote. *Please do not spend a significant amount of time trying to get uMessage to work.*

Above and Beyond

- *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.