CSE250, Spring 2022 Assignment 6 Due Sat. Apr. 23, 11:59pm

Lectures and Reading:

For next week, finish Chapter 16. Contrary to how I have skimmed over long code examples in the text before, study the code in section 16.3 (pages 506–510) closely. Similar code in the ISR framework would avoid such a hard separation of the notions of "key" and "value" and thus might be simpler in parts. Then skip Chapter 17 (which might support a compilers course), and go into Chapter 18. While reading 18, hark back to chapter 13, while noting that 18 gives the signature implementation of a priority queue. The topics we will emphasize regarding Chapter 18 are (i) how a heap enables compiling "top-ten (or so)" lists in less time than the order-$n \log n$ needed for sorting, and (ii) how to quickly fix up a heap if the priority value of one item suddenly changes. (I am not sure whether I will cover the topic of immutable trees in section 16.3.2; it could distract away from the flow to chapters 21 and 20.)

——– *Assignment 6, due Sat. Apr. 23 "midnight stretchy" on CSE Autograder* ——–

Brief Task Statement:

Redo the task of Assignment 4, but with a different strategy for handling words like `question` that are listed as multiple parts of speech with separate sets of synonyms. Instead of (i) creating separate tagged keys (as in the given answer code) or (ii) lumping the multiple synonym sets together under one key (which was also OK for this task), simply store multiple entries having the same key. The built-in Scala `Set` and `Map` data structures do not allow "duplicate-key items" but the `ISR` code does. Since the ISR implementations are sorted, the multiple items with the same key like `question` will be consecutive when one iterates through the data structure. Save the information about whether the key is a noun or verb or adjective (or etc.)—when Fallows gives it at all—in a separate field of the `SynonymEntry` class that is not part of the key. Demonstrate that your new code replicates the results of Assignment 4 while either (i) giving faster time than your previous strategy (i), or (ii) allowing you to preserve the information about whether the reciprocal synonyms were for the original key word being a noun or verb or adjective (or etc.) if you used something more like strategy (ii). Write one essay that includes comparing the results of timing code between the assignment 4 answer and your new one (report the average of ten runs for each) using either `BALBOA` or `BALBOADLL` as the underlying implementation of the `ISR` trait that is sandwiched and selected by changing one line in `Cardbox`. Write a second essay that compares the time against either the `SortedSLL` or `SortedDLL` implementation, summarizing why `BALBOA` is faster.

Points are: 20 for compiling under the ISR framework, 20 (only) for replicating the correct output from Assignment 4, 20 for the client code edits needed to employ "first-class iterators," and 18 for each essay, making 96 points total. A header comment with `/**` atop your `SynonymsISR.scala` that includes your name is now required.

Files to Submit. Please zip the following. All Scala code files can go together into `src/` or some other subfolder, but `output.txt` and `essay6.txt` must be in your project root folder. NOTE CHANGE (4/18/22, am): The dictionary file has been upgraded to

`Fallows1898fx.txt`, which has several hundred typo fixes and adds about 100 new entries overall. (It is still far from perfect but has all `KEY:` fields fixed, so as not to detract from the main point about handling like keys.) Your submissions must use this. The new file gives the same results on the tests in the Assignment 4 autograder, although the fixed typos produce 2 reciprocal pairs (`quantity` and `sum`, `quick` and `brisk`) plus 6 new non-reciprocating ones, increasing `output.txt` from 202 to 210 lines overall.

- `SynonymsISR.scala` Note that the file of this name in `.../cse250/DataStructures/` has been replaced by `SynonymsISR0.scala`. You are to modify that source file and rename it back.
- `ISR.scala`, `Cardbox.scala`, `BALBOADLL.scala`, and whichever others of the ISR implementation code you use. (Note, however: the grading script will overwrite them by the reference versions anyway. No problem submitting all of the repository code.)
- `output.txt`
- `essay6.txt`, with `two` essay answers (A) and (B).

Submitting `Fallows1898fx.txt` in your project root folder is optional, or having it read as `../Fallows1898fx.txt` (in which case, you won't submit it). Auxiliary code files are OK if you do not use `package` and put them in the same folder as `SynonymsISR.scala`. The repository code may be updated during the span of this assignment and a further one, but this comes with a promise that the code will work alike as visible to clients—this is a larger-world point of the whole exercise.

Specific Directions: The file `SynonymsISR0.scala` has some directions inside it. Major points:

- Directions in ALL CAPS indicate things to change—and you can then remove those comment lines.
- Names that end in `0` should be changed to names without the `0` when you modify the code.
- The code for `apply` in the `SynonymBox` class (name after you remove the `0`) will not be used and can be removed.
- Note that `ISR` containers are being used on two levels: for the overall dictionary of synonyms entries, and for the `synonyms` within an entry. Both will use the same implementation—whichever one you select inside `Cardbox.scala`.
- You should use `contains` only with the `synonyms` within an entry. For the large outer container holding the 6,000+ synonyms entries, however, you will use the `find` method inherited from the `ISR` trait and then further use the `Iter`ator it returns.
- When you change the implementation line in `Cardbox`, both `Cardbox.scala` and then the client (i.e., `SynonymsISR.scala`) need to be recompiled. Your IDE should be able to recognize and do this automatically—at least it works for IntelliJ—but from the command line you must follow this sequence.

- The timing code is already provided. To report times, you must do an average of 10 runs. It is OK to leave the line `val allowPrintWhenTiming = true` set true when timing; the main essay points are not affected by the small differences from allowing screen and/or file output. Indeed, you must leave it set true in tour final submission, so we can check your output.

Motivations:

It is often advantageous to manage data with a more flexible notion of what is *key* and what is *value* than the standard `Map[K,V]` implementations provide. This includes allowing items with different values to share the same key. The *ANSI C++ Standard Template Library* (STL) provides `multiset` and `multimap` at the same primary level as its `set` and `map` containers (which work the same as Scala's `Set` and `Map`). The `multiset` (also called "bag") is especially useful, and is what the `ISR` trait allows.

A *sorted* data structure will guarantee that items with the same key will be consecutive in the iteration order. (We will see that some kinds of hash tables can give this as well.) If we can rapidly find the first such item, then we need only do consecutive iteration steps to read them all. This can be quicker than a regime where we extend a key `k` into distinct keys `k1, k2, k3, ...` for inclusion into a `Map` object, say `mymap`. The reason is that the calls `mymap(k1)`, `mymap(k2)`, and `mymap(k3)` require separate evaluations of the `Map`, which can be relatively expensive. Note that these evaluations are the most natural thing to do with a `Map` ("that's what its code is for") and yet we can replace them by simpler steps involving an explicit iterator that are less repetitive and hence quicker.

For this, it helps to have a somewhat more flexible environment for managing iterators than the standard Scala/Java-based methods `hasNext` and `next()` provide. For one thing, it is nice to be able to read the current data point of an iterator `itr` without having to advance it to the next item as `itr.next()` does. This is much the same motivation as our text's making `peek` as well as `pop()` a top-level operation of stacks and queues in chapter 7. The ISR framework does this by coding an `apply` method, which Scala automatically makes involve just parentheses by themselves. So we can get the value by `itr()` and the iterator `itr` stays put. (It might be nice if we could back up `itr` to the previous item, too. The "ISR" repo stops short of such a `prev` operation, but you could choose to add the functionality. In any event, the whole approach is toward making iterators be "First-Class Objects" in the code. Being able to quickly get an iterator to the desired location, rather than having to create `iterator` in the first position and traverse the whole data once as in the text, is the other key point.)

The iterators also allow a different way of handling an unsuccessful item lookup. Note at https://docs.scala-lang.org/overviews/collections/maps.html that the standard Scala `Map` class offers two possibilities. If the key `k0` is not present in `mymap`, the simple call `mymap(k0)` will throw a `java.util.NoSuchElementException`. The call `(mymap get k0)` will return `None`, as opposed to `Some(v)` when the key has the associated value `v`. This involves either the hazard of a runtime error or an extra level of `None/Some` wrapper syntax around the data point. The way exemplified in the `ISR` trait and its conformant implementations, where `mybox` is the whole data structure and `i0` is an item playing the role of the key `k0`, is that a call to `mybox.find(i0)` returns an iterator `itr`. If the item is not found, then `itr.hasNext` (which is always a quick and simple call) is false. If the item is found, then `itr.hasNext` is true and `itr()` returns the found item—which may have extra value information compared to a "dummy item" `i0`. The whole process is visually like a web form where if the initial name or e-mail info you type in is found, then the form fills in the values of the rest. Anyway, the lookup call itself will never bomb—you just have to remember to test `hasNext` (or equality to the `end` iterator) on the result before trying to read the value.

A final point is that this protocol for handling search results—combined with protocols for inserting or removing an item—has rules that are independent of any particular implementation. Scala reflects this in that `Set` and `Map` come in several varieties: you can specify `SortedSet` or `SortedMap` or alternatively `HashSet` and `HashMap`. (The latter seem to be the same as the default forms.) This allows you to compare the efficiency and stability of different implementations. The file `Cardbox.scala` allows changing the implementation by commenting-in just one line of code.

These themes are reflected in this assignment in the following ways:

1. We have keys that are single words, but the part of speech (`a` for Adjective, `n` for Noun, `v` for Verb) also matters. Some key words like `question` have different data for different forms. The word `mean` not only has all three forms, there are two entries for the adjective form in which the original 1898 dictionary gives no lexical distinction at all—so they are duplicate items even under a regime that would make (say) `mean_a` the key. Even if our task allows a shortcut of lumping all the lists for `mean` together into one record, we would still want to know which form of the word `mean` was matched. (We might also wish to know between the two adjective forms, but Fallows gives no further distinction, so we won't either.)

2. So we make a separate field in our simple `SynonymEntry` class to hold the part of speech—so we can print it out later if and when we need to. (If the part isn't given, as for most Fallows entries, store `u` for Unknown there.) The simple `keyComp` function does not involve this field.

3. If we wanted to, we could rebuild the data structure with a different `keyComp` function that could use the idea `a < n < v` to give a "second level of sorting." But this is extra work. The same caveat applies to the idea of sticking with the standard Scala `Map` but creating the two-level sort—so that items with the same first-level key would come adjacent and one could iterate rather than evaluate the `Map` multiple times.

4. Having a single item type `A` in `Cardbox[A]` rather than splitting into "`K`" and "`V`," avoiding the creation of a "Frankenkey" with the part-of-speech appended, and iterating over just the desired sub-range, yields code that is both simpler and quicker.

Set-Up For This Assignment:

The first thing you should do is download the whole code in `/.../cse250/DataStructures/` into your IDE. If you've already done this with the original `FlowerCardbox` test client, note that the source files were updated when the Assignment 4 key was posted on 4/11 and `AIOLI.scala` (which you will not need to use) has reappeared. Your IDE (at least in my own IntelliJ setup) will automatically be able to figure out the chains of dependence, in particular:

`ISR.scala` ← `SortedDLL.scala` ← `BalboaDLL.scala` ← `Cardbox.scala` ← `SynonymsISR0.scala`

with all the files in the same folder, without needing organization into `package`(s). (At the command line, one needs to recompile dependent classes after any change; in the C/C++ world, this is managed by a `Makefile`.) Try changing the implementation line in `Cardbox.scala` from `BALBOADLL` (or however it is set) to `SortedDLL` directly, and then to `SortedArray`. Observe the changes in running time, as well as that the code still works.

Options—And Not

It is recommended that you work from `SortedISR0.scala` as provide, at least initially. There are no academic integrity issues in doing so, and submitting your revision of this code will not bring any deficit in points. You may, however, try converting your own previous client to the ISR framework, as would be good experience for later. One of the points of the way `SortedISR0.scala` is given, compared to the official answer key `A4Key/SynonymsKey.scala` (the version with `2` handles the alternate form of the Fallows file, as does `SortedISR0.scala`), is that the initial conversion can be done with minimal changes in syntax. To carry over the timing code, be aware that the step of reading the file to create the database is timed separately from the task of finding reciprocal synonyms, and this assignment focuses only on the latter. Once you work on your `SortedISR.scala` proper, you will make further changes that take advantage of features of the `ISR` trait—and you may find that the code overall gets *shorter*.

One option that is not provided is using Fallows's cross-references. When Fallows writes e.g.`Quagmire, [See MARSH]" the intent is that the synonyms of `marsh` are also synonyms of `quagmire`. Unfortunately, this becomes an unexpected quagmire especially toward the end of the alphabet, where the Project Gutenberg file has myriad cases of cross-references being put on the wrong entries, off-by-one or worse. I do not understand how a digitized file—the source was evidently this—leads to this kind of error. In a released file, this is inexcusable (SYN: Unmitigated, unpardonable, indefensible, unjustifiable). My version `Fallows1898fx.txt` fixed many of these in the course of sanitizing every `KEY:` field, but there are many more. Thus we have to shut down this possible option. (It was possible only when Fallows gives a single cross-reference anyway, and even some of those just go to the antonym. He was of course writing for processing by human beings, not computers.)

FAQ and Clarifications