Assignment 3: Merging Spreadsheets

Everywhere you look, data are organized into **tables**. In sports, tables are used to summarize [athletes’ performance](https://cornellbigred.com/sports/mens-ice-hockey/stats/2022/pdf): rows correspond to players, and columns correspond to “stats” such as shots, penalties, and goals. In finance, bank statements are tables of transactions, each showing a debit/credit amount and the account’s remaining balance. In science labs, you might write down tables of measurements taken during an experiment.

Sometimes it would be useful to combine information that is spread across multiple tables. To take an example from economics, if we had a spreadsheet listing the Gross Domestic Product (GDP) of all 50 US states, and another spreadsheet containing median income of the residents of those states, it would be useful if we could easily generate a single spreadsheet containing both kinds of information for each state—whether or not the two tables list the states in the same order.

Spreadsheet programs like Microsoft Excel are used to accomplish that kind of table merge. Such programs tend to use complicated file formats, such as “.xlsx”. But spreadsheets can also be saved in a simple format called “CSV” that makes their data available to other programs, including your own.

In this assignment, you will create a program that merges information contained in CSV files.

Learning objectives

* Implement linked-list operations
* Read and write CSV data
* Implement a main program from scratch
* Create end-to-end test cases

Recommended schedule

**Start early.** Office hours and consulting hours are significantly quieter shortly after an assignment is released than closer to the deadline. Although there are fewer TODOs on this assignment than on A2, they each require more code to implement. We recommend spreading your work over *at least* 5 days. Here is an example of what that schedule could be:

* Day 1: Skim this entire handout. Download the release code and get it set up in IntelliJ. Make sure you can run the test suite LinkedSeqTest. Complete TODOs 0–1 in LinkedSeq and confirm that testToString() passes. Move on to TODOs 2–3, including writing and passing their corresponding tests in LinkedSeqTest.
* Day 2: Complete TODOs 4–7 in LinkedSeq, including writing and passing their corresponding tests in LinkedSeqTest.
* Day 3: Create class CsvJoin and implement static methods csvToList() and join(). Uncomment and run the corresponding test cases in CsvJoinTest.
* Day 4: Implement CsvJoin.main(), including any helper methods you decide to define. Perform manual testing (by tweaking program arguments and/or file contents) to verify desired behavior.
* Day 5: Create new input-tests cases. Re-read your code. Ensure that it conforms to our style guide and complies with all implementation constraints. Submit to CMSX, then confirm that you submitted the files you meant to (in particular, check that the contents of “input-tests.zip” look like the example listing in this handout).

Collaboration policy

On this assignment you may work together with one partner. Having a partner is not needed to complete the assignment: it is definitely do-able by one person. Nonetheless, working with another person is useful because it gives you a chance to bounce ideas off each other and to get their help with fixing faults in your shared code. If you do intend to work with a partner, you must review the [syllabus](https://www.cs.cornell.edu/courses/cs2110/2023sp/syllabus.html) policies pertaining to partners under “programming assignments” and “academic integrity.”

Partnerships must be declared by forming a group on CMSX before starting work. The deadline to form a CMS partnership is **Thursday, March 2, at 11:59 PM**. After that, CMSX will not allow you to form new partnerships on your own. You may still email your section TA (CCing your partner) to form a group late, but a 5 point penalty will be applied. This is to make sure you are working with your partner on the entire assignment, as required by the syllabus, rather than joining forces part way through.

As before, you may talk with others besides your partner to discuss Java syntax, debugging tips, or navigating the IntelliJ IDE, but you should refrain from discussing algorithms that might be used to solve the problems, and you must never show your in-progress or completed code to another student who is not your partner. Consulting hours are the best way to get individualized assistance at the source code level.

Frequently asked questions

If needed, there will be a pinned post on Ed where we will collect any clarifications for this assignment. Please review it before asking a new question in case your concern has already been addressed. You should also review the FAQ before submitting to see whether there are any new ideas that might help you improve your solution.

I. Assignment overview

To complete this assignment, you will need to do the following:

1. Complete the implementation of a linked list class. Employ **defensive programming practices** to ensure that preconditions are respected and that invariants are maintained.
2. Read tables from files in CSV format and represent them as lists.
3. Merge data from two tables according to a “join” operation.
4. Write tables in CSV format.
5. Construct example input and output files to verify end-to-end behavior.

As usual, all public functionality must be covered by unit tests.

**Setup.** Download the release code from the CMSX assignment page; it is a ZIP file named “a3-release.zip”. Follow the same procedure as for A2 to extract the release files, open them as a project in IntelliJ, select a JDK, and add JUnit 5 as a test dependency by resolving import errors in “tests/cs2110/LinkedSeqTest.java”. Confirm that you can run the unit test suite. Test case testToString() should fail with an AssertionFailedError, but the other tests should pass.

II. Implement a Singly Linked List

In this part of the assignment, you will implement a singly linked list. Another word for “list” is “sequence”, and the files and classes you will work with in this assignment use the name Seq instead of List. That’s because Java’s Collections library already has several classes with List in their name, and we would like to avoid confusing error messages (or hints from IntelliJ) that could result if we used List as part of the assignment type names.

**Your tasks.** In “LinkedSeq.java” you will find eight tasks labeled TODO 0–7. Those ask you to implement a singly linked list data structure, which is an instance of the List ADT. The rest of this section of the handout gives you guidance on how to complete those TODOs.

**The List ADT.** A **list** is a collection, like a bag. Unlike a bag, a list maintains an ordering among its elements. That ordering can be 0-based or 1-based. In this assignment we’ll refer to a 1-based ordering as a *position*, and a 0-based ordering as an *index*. So the element at position 1 has index 0, as with arrays. A list’s size is *dynamic*: it can grow by inserting an element at the beginning, end, or even middle of the list (which will affect the positions of subsequent elements), and it can shrink by removing an element. Iterating over a list (such as with an enhanced for-loop) will yield each element in order. But lists also support a Get operation to retrieve an element given its position (DSAJ) or index (Java collections).

**The singly linked list data structure.** A singly linked list is implemented much like a linked bag. But here (and often elsewhere) a singly linked list will maintain a reference to the *tail* of the chain of nodes in addition to the *head*, so that elements can easily be appended to the end of the list as an alternative to prepending them to the beginning. This means that, whenever mutating the list, it is possible that the head pointer, tail pointer, and size will all need to be updated to maintain the class invariant.

**Restriction**: You must implement LinkedSeq using (only) these fields. You may not use any data structures from the Java Collections library in your implementation of LinkedSeq. The release code, however, imports some interfaces and exceptions from java.util, and that is fine.

**Testing your linked list.** As with A2, you should practice **test-driven development**, implementing and testing your methods incrementally. Each TODO specifies the test coverage for which you are responsible. See the unnumbered TODO in LinkedSeqTest for additional testing guidance. The given tests make use of helper functions to conveniently create lists of “corner case” sizes (empty, one element, two elements, more than two elements); you are welcome to use these helpers in your own tests. You are allowed to add helper methods to LinkedSeq, but they must be private and have thorough specifications.

**Defensive programming.** You must assert that preconditions are satisfied in any method with parameters. You must also assert that the class invariant is satisfied at the end of the constructor and any method that mutates the list.

**Efficiency requirement.** Do not call get() in a loop over indices. That pattern is inefficient, because each call to get() will traverse the list from the beginning instead of from where the previous call left off**. Use other patterns to iterate over the list (or its nodes) instead**, such as traversing next fields with a while loop.

**Iteration.** Seq extends Iterable and LinkedSeq has an iterator() method implemented for you. This makes it possible to use LinkedSeq in an **enhanced for-loop**, which will be helpful to client code such as your main program. Enhanced for-loops support easy iteration over arrays and other collections of data. For example, you can print all the integers in a Seq<Integer> like this:

Seq**<**Integer**>** list **=** **...**

**for** **(**Integer s **:** list**)** **{**

System**.**out**.**println**(**s**);**

**}**

**Advice on testing:**

* Make sure that you include @Test directly above each test method. Test methods that do not have @Test will not be run by JUnit, which might lead you to erroneously think that all your test cases are passing!
* Look at the lists of which test procedures were run in the “Run” window whenever you run your tests to ensure you’re testing all of the cases that you have written.
* Use JUnit assertions, not Java assert statements, in JUnit tests. Remember that the expected value is the *first* argument for assertEquals().
* Ensure you have the required number of test cases for each method. Think about what tests are likely to improve coverage in both a **black-box and a glass-box sense**.
* To **identify corner cases** that are more likely to catch bugs, think about the fields of LinkedSeq:
  + An empty list has a null head and a null tail.
  + A list of size 1 has head == tail.
  + A list of size 2 has head != tail, but no nodes in between.
  + A list of size 3+ has at least one node in between head and tail.

If your code works for some of these sizes but not others, thinking about these properties can help you pinpoint the bug.

* Since any helper methods of LinkedSeq must be private, you will not test those directly. Instead, you will test the public methods, thereby indirectly testing private helper methods.

III. Implement the CSV Join

Create a file named “CsvJoin.java” in your project’s “src/cs2110/” directory and declare a public class CsvJoin in package cs2110.

In this class you will be implementing and testing three methods, as described below. You are also welcome to declare additional helper methods, which must have thorough specifications.

There are no pre-written TODOs for these three methods, because we are turning more of the program development over to you in this phase of the assignment. Metaphorically, the “training wheels” are coming off.

Tables as lists

A table is organized into rows and columns. This is a two-dimensional arrangement, but it is possible to represent the structure using *nested* one-dimensional abstractions, like lists. For example, you could treat the table as a “list of rows,” where each row is itself a list of the values for that row in each column. This is called a **row-major** representation.

Using our Seq<T>, a table whose entries are strings would be a Seq<Seq<String>>, aka “a list of lists of strings.” If table is a variable of this type, then table.get(0).get(3) represents the value in the table on the first row and in the fourth column.

A *rectangular* table requires that every row have the same number of columns. But our list-of-lists representation permits *ragged* tables that are not rectangular—that is, some rows might contain a different number of columns than other rows. Sometimes this makes sense, but it would not be desirable in applications expecting a certain number of columns, so there are times when you might need to verify a table’s shape.

The CSV file format

One of the simplest and most common ways to represent tables is to save them as plain text files using the comma-separated values (CSV) format. These files can be read and written by any spreadsheet program. They are also easy to produce and consume from technical software platforms (like MATLAB and R) as well as from your own programs. In fact, many of your professors’ interactions with CMSX take place through CSV files.

Here is example of the contents of a CSV file:

State,Capital

New York,Albany

California,Sacramento

Florida,Tallahassee

Texas,Austin

Texas,Houston

Vermont,Montpelier

Each line represents a row, and columns on each row are separated by commas.

**Exercise.** To build your intuition for CSVs, try creating some small tables in a spreadsheet program like Microsoft Excel or Google Sheets and saving them in CSV format:

* For Microsoft Excel: File → Save As; File Format: Comma Separated Values (.csv)
* For Google Sheets: File → Download → Comma Separated Values (.csv)

Then do the reverse: using a text editor, create or modify a CSV file, then open it in a spreadsheet program to see the resulting table.

Simplified CSV format

Commas are used to separate columns in CSVs. Likewise newlines are used to separate rows. What if you want a cell to contain a comma or a newline? In this assignment, we disallow that. We call this restricted file format a *Simplified CSV*. One consequence is that spreadsheet programs will treat quotation marks and backslashes differently than this assignment, so those are best avoided in your testing. Note that spaces *are* accommodated in cell values, as in New York above.

For production applications, software needs to accommodate commas and newlines in data values, and this is done in a well-defined way using quotes and escapes as specified in [RFC 4180](https://www.ietf.org/rfc/rfc4180.txt). To get all the details right, it is usually best to rely on an existing library, like [Commons CSV](https://commons.apache.org/proper/commons-csv/), rather than parsing a format on your own. But for this assignment, you may only use Java’s standard I/O functions, hence our adoption of the Simplified CSV format.

Method 1: Convert a CSV to a List

Write a method named csvToList() with the following declaration:

*/\*\**

*\* Load a table from a Simplified CSV file and return a row-major list-of-lists representation.*

*\* The CSV file is assumed to be in the platform's default encoding. Throws an IOException if*

*\* there is a problem reading the file.*

*\*/*

**public** **static** Seq**<**Seq**<**String**>>** **csvToList(**String file**)** **throws** IOException

Implement this method with a FileReader, which will assume the platform’s default encoding, and a Scanner, as you learned in discussion section. It should read each line of the CSV, then it should separate each line into tokens delimited by commas using String.split(",", -1). The -1 argument will enable you to correctly handle empty columns at the end of a row.

Uncomment testCsvToList() in CsvJoinTest to test your implementation using some example CSV files included with the assignment.

A (Simplified) CSV file could represent a ragged table, or a table that has 0 columns (yet 1 or more rows). That is okay. Your implementation of csvToList() should not report any kind of error about these irregularities when you read CSV files, since a list-of-lists suffices to represent them. You will handle any such irregularities as part of main().

Joins

Our desired operation of merging two data tables is called a **join** in *relational database* jargon. Like a spreadsheet, a relational database consists of tables with rows and columns. A join combines two tables to produce a new table, based on information matching in one or more columns. There is more than one kind of join, but we are interested in computing a **left outer join** on the *first column* of each table.

For each row in the first table, the left outer join identifies every row in the second table with a matching value in the first column. Then, for each matching row, the new table contains a row that concatenates the row from the first table with the matching row, omitting the first column from the second table (since it is redundant). If there is no such row in the second table, a single row is added to the new table, but with empty entries for columns that would have come from the second table.

For example, consider the following two tables. The first one records some (past and present) capitals of states; the second has some population and economic data:

| **State** | | **Capital** | |
| --- | --- | --- | --- |
| New York | | Albany | |
| California | | Sacramento | |
| Florida | | Tallahassee | |
| Texas | | Austin | |
| Texas | | Houston | |
| Vermont | | Montpelier | |
| **State** | **Population** | | **GDP** |
| New York | 19.5 | | 1500 |
| Florida | 21.2 | | 1150 |
| California | 39.4 | | 3400 |
| Texas | 28.6 | | 2000 |
| North Dakota | 0.8 | | 56 |

The left outer join of these two tables is the following table:

| **State** | **Capital** | **Population** | **GDP** |
| --- | --- | --- | --- |
| New York | Albany | 19.5 | 1500 |
| California | Sacramento | 39.4 | 3400 |
| Florida | Tallahassee | 21.2 | 1150 |
| Texas | Austin | 28.6 | 2000 |
| Texas | Houston | 28.6 | 2000 |
| Vermont | Montpelier |  |  |

The order of rows in the result table always matches the order in the first table. There is an incomplete “Vermont” row because there is no “Vermont” row in the second table; there is no “North Dakota” row because “North Dakota” does not occur in the first table (despite occurring in the second). Note that if there had been two “Texas” rows in the second table, the result table would contain *four* “Texas” rows: two (Texas, Austin) rows followed by two (Texas, Houston) rows.

In this assignment, there is nothing special about the first row of a table. Although it will often be used as a header, containing labels for each column, our simplified join operation will always look for matches in the first column whether or not the first entries in those columns are the same.

Method 2: Join two tables

Write a method named join() with the following declaration:

*/\*\**

*\* Return the left outer join of tables `left` and `right`, joined on their first column. Result*

*\* will represent a rectangular table, with empty strings filling in any columns from `right`*

*\* when there is no match. Requires that `left` and `right` represent rectangular tables with at*

*\* least 1 column.*

*\*/*

**public** **static** Seq**<**Seq**<**String**>>** **join(**Seq**<**Seq**<**String**>>** left**,** Seq**<**Seq**<**String**>>** right**);**

As usual, you should **assert that preconditions are satisfied**. That means you will now need to implement a method to check for validity of tables according to the precondition. This method will also be useful to callers of join() so that they can avoid violating the preconditions in the first place (remember that asserts are for catching programming bugs, not user errors).

It wouldn’t hurt to assert that the result is rectangular too, though this is not required.

Uncomment testJoin() in CsvJoinTest to test your implementation using the two example cases included with the assignment.

Method 3: Main

Write a main() method that merges two CSV files using a left outer join, and outputs the resulting CSV.

**Input.** Your method should expect exactly 2 program arguments, which are the filenames of the two CSV files to join. The first corresponds to the “left” table and the second corresponds to the “right” table. Those can be set in the **IntelliJ Run Configuration**. For example, if you set the program arguments to file1.csv file2.csv, the strings "file1.csv" and "file2.csv" will then be available in the args array passed to the cs2110.CsvJoin.main() method at positions 0 and 1. If there are any problems reading the files, or if the resulting tables do not meet the preconditions for join(), print an appropriate helpful message for the user to **System.err**and exit the program with a status code of 1 using **System.exit(1).** A stack trace is not considered a helpful error message.

Examples of sufficiently helpful messages for this assignment include:

* User does not provide exactly two program arguments:
  + Usage: cs2110.CsvJoin <left\_table.csv> <right\_table.csv>
* A problem occurred when reading a CSV file:
  + Error: Could not read input tables.  
    java.io.FileNotFoundException: missing.csv (No such file or directory)
* Tables in CSV files cannot be joined:
  + Error: Input tables are not rectangular or have no columns.

These are merely examples; you do *not* need to reproduce them exactly—any similarly specific and readable message is fine. Note that printing an exception’s message (but not stack trace) is okay if context is provided.

**Output.** Your method should output the resulting table to System.out. This should be the *only* output to System.out. The output must be in the format of a valid Simplified CSV file. That means you should not try to use your linked list’s toString() method to generate output, because it does not produce output in the format of a CSV file. Instead, you should implement a method to output in Simplified CSV format.

**Testing.** In the “input-tests” folder of the project are example inputs and outputs for the program that can be used for **end-to-end testing**. These are the same files used by testJoin(). Run your program with the input files as arguments, then compare your program’s output to the contents of the output files. For example, modify the Run Configuration for CsvJoin to set the program arguments to input-tests/example/input1.csv input-tests/example/input2.csv. Then run the program and compare what it prints to what’s in “input-tests/example/output.csv”.

As implied by the TODO in CsvJoinTest, you must prepare at least two additional end-to-end test cases. To add a new case, create a new folder under “input-tests” and give it a name. The folder name must not contain any spaces. Then create two tables in CSV format whose first columns correspond to the same attribute and save them in your new folder as “input1.csv” and “input2.csv”. (Tip: use a spreadsheet program to prepare the tables, then save to CSV as described above). Next, edit your program’s Run Configuration to pass these two files in as inputs. Run your program and copy and paste its output into a file named “output.csv” in your new folder. Open the output in a spreadsheet program to confirm that it looks correct.

With your example inputs and outputs thus prepared, you want to add a JUnit test case to ensure that your code keeps producing the same output, given the same inputs—this is known as **regression testing**. Modify testJoin() to add new calls to testJoinHelper(), passing the names of the directories you created as arguments. Make sure the test still passes!

In order to submit your new test cases, create a ZIP archive named “input-tests.zip” containing your project’s “input-tests” folder. Double-check the contents and organization of your ZIP file before submitting it. For example, if your test folders were named “bball\_stats” and “pokemon”, then your ZIP file’s contents should look like this:

input-tests/

├── bball\_stats

│   ├── input1.csv

│   ├── input2.csv

│   └── output.csv

├── example

│   ├── input1.csv

│   ├── input2.csv

│   └── output.csv

├── pokemon

│   ├── input1.csv

│   ├── input2.csv

│   └── output.csv

└── states

├── input1.csv

├── input2.csv

└── output.csv

IV. Submission

Please make sure to fill in the metadata in the comment at the top of “LinkedSeq.java”. Then upload your files “LinkedSeq.java”, “LinkedSeqTest.java˝, “CsvJoin.java”, “CsvJoinTest.java”, and “input-tests.zip” to **Assignment 3** on CMSX before the deadline.

If you forgot where your project is saved on your computer, you can right-click on “LinkedSeq.java” in IntelliJ’s project browser and select “Open In”, then your file explorer (e.g. “Explorer” for Windows, “Finder” for Mac). Be careful to only submit “.java” files, not files with other extensions (e.g. “.class”). Note that your test suite will be under “tests/cs2110/”, while your other files will be under “src/cs2110/”.

After you submit, CMSX will automatically send your submission to a *smoketester*, which is a separate system that runs your solution against the same tests that we provided to you in the release code. The purpose of the smoketester is to give you confidence that you submitted correctly. You should receive an email from the smoketester shortly after submitting. Read it carefully, and if it doesn’t match your expectations, confirm that you uploaded the intended version of your file (it will be attached to the smoketester feedback). Be aware that these emails occasionally get misclassified as spam, so check your spam folder. It is also possible that the smoketester may fall behind when lots of students are submitting at once. Remember that the smoketester is just running the same tests that you are running in IntelliJ yourself, so don’t panic if its report gets lost—we will grade all work that is submitted to CMSX, whether or not you receive the email.