## CS 367 Programming Assignment 3:  Memory Efficient Merging of Sorted Files

**DUE by 10:00 PM on Sunday, March 19th**  
  
[Announcements](https://canvas.wisc.edu/courses/23073/assignments/27571#Announcements) | [Overview](https://canvas.wisc.edu/courses/23073/assignments/27571#Overview) | [Specifications](https://canvas.wisc.edu/courses/23073/assignments/27571#Specifications) | [Javadocs](http://pages.cs.wisc.edu/~cs367-1/assignments/p3/javadocs/index.html) | [Submission](https://canvas.wisc.edu/courses/23073/assignments/27571#Submission)

## Announcements

Corrections, clarifications, and other announcements regarding this programming assignment will be found below.

## Overview:

In Program 3, you are tasked with implementing a relatively simple algorithm for merging (reducing) a series of (sorted) input files into a single output file.  The basic idea is to read in data from multiple files where each file contains some common identifier and some additional data for that identifier.  Based on the common identifier, merge the data from multiple files to form a single record (line) of data for the common identifier.

One way to solve this problem would be to read the entire contents of each file into memory, sort them using any method you like, and then write them to the output file in sorted order.  If the files are extremely large however, or if there is limited RAM available for this process, this method will fail.

The algorithm you are to implement for this assignment will use only a small amount of memory, regardless of the size of the input files. In order to to be able to test how general your implementation is, you will also be writing code to handle data from two application areas - weather data and thesaurus entries. 天气和词条

### Goals:

The goals of this assignment are to:

* Understand, implement, and use minimum priority queues
* Observe the generality and flexibility of the data structures and algorithms learned in this course.
* Gain experience with file I/O, interfaces, and abstract classes in Java.

### Background:

The information used within a program while it is running is almost always stored in Random Access Memory (RAM). In Java, it is possible to specify for a given instance of the Java Virtual Machine how much RAM it will be allowed. This value is generally 2-4 GB, and of course it must be less than the total RAM available to the physical machine. Similar limitations exist for other programming languages. Therefore, our input files do not need to be extremely large in order for a naive approach of merging them to fail. For example, suppose that we have 100 files, each of which is 200 MB. We would need 20 GB (100 \* 200 MB) just to read all of them into memory!

The algorithm you will write for this assignment needs only to hold in memory at most one line per file at any given time. In the example given above, suppose that each of the 100 files contains a million lines. Then, we would only need approximately 20 KB (each line is 200 MB/1 million = 200 bytes, and there are 100 files). As you can see, the total memory required grows linearly with the number of input files, independent of the actual size of the files. Our algorithm will make use of a priority queue to merge these files. The basic version of the algorithm works as follows:

First, take one entry from each file and insert them into the priority queue, keeping track of which entry came from which file. Then, as long as the priority queue is not empty: Remove the minimum entry e from the queue, write the entry e to the output file, and then get the next entry e' from the file from which e came (unless that file has no more entries, in which case do nothing), and insert e' into the queue.

For this assignment's applications, two different input files may have entries with the same key. For weather data:

* Suppose two files DEWP.txt and TEMP.txt are as given below:
  + (DEWP.txt): 726506,20170201,18.7
  + (TEMP.txt): 726506,20170201,25.8
* Since both of those lines have the same key (station ID and date), they should appear on the same line in the output file.

In the case of thesaurus data:

* Suppose two files 1.txt and 2.txt are as given below:
  + (1.txt): bad:lamentable,sorry,rotten
  + (2.txt): bad:naughty,unfavorable
* Since both of those lines have the same key ("bad"), their synonyms should appear on the same line in the output file.

With this in mind, you will be implementing a slight variation of the algorithm given above. Entries in the input file with the same key value should be merged into a single record before they are written in the output file. For weather data, this means writing all of the readings for a given Station-Date combination on a single line of the output file. For thesaurus data, this means merging the different synonym lists (removing any duplicates), sorting them, and writing the resulting list on a single line. Here is the algorithm you will be implementing:

1. Take one entry from each input file and insert them into the queue.
2. Create an empty record *r.*
3. Remove the minimum entry e from the queue and merge it with r.
4. While the queue is nonempty:

remove the minimum entry e from the queue and compare its key to the key associated with r.

If they are the same,

merge e with r.

Otherwise,

write r to the output file, clear r, and then merge e with r.

Either way, take the next entry e'from the same file from which e came and insert it into the queue (again, if this file has been exhausted, do nothing).

1. Write r to the output file (this step is necessary because otherwise the last record would never get written).

## Specifications

### Running the program

The program can be invoked with either of the following:

java Reducer weather <directory> <output\_file>  
java Reducer thesaurus <directory> <output\_file>

Above, <directory> indicates the name of a directory: all .txt files in this directory should be considered input. They can be assumed to be well-formed:

* For weather data, each line will be in the format:
  + <station> (int), <date> (int), <data point> (double or int)
* For thesaurus files, each line will be in the format:
  + <word>:<synonym1>,<synonym2>,<synonym3>,...,<synonymN>

<output\_file> indicates the name of the file to which the output file should be written.

### Example runs

The files found in these examples are included in p3.zip.

#### **Weather data**

Assume that the following files are in the directory named weather\_data:

DEWP.txt TEMP.txt VISIB.txt WDSP.txt

-------------------------- -------------------------- -------------------------- --------------------------

726506,20170201,18.7 726506,20170201,25.8 726506,20170201,9.4 726506,20170201,7.7

726506,20170202,-3.5 726506,20170202,9.9 726506,20170202,10 726506,20170202,9.4

726506,20170203,-4.1 997340,20170201,33.4 726506,20170203,10 726506,20170203,7.3

726506,20170204,2.9 997340,20170202,17.6 726506,20170204,10 997340,20170203,12.3

997340,20170204,9999.9 997340,20170203,16.5 726506,20170205,10 997340,20170204,8.5

997340,20170205,9999.9 997340,20170204,22 997340,20170201,999.9 997340,20170205,10.8

997340,20170205,32.6 997340,20170202,999.9

997340,20170203,999.9

997340,20170204,999.9

997340,20170205,999.9

Then, upon running:

java Reducer weather weather\_data/ weather\_output.txt

weather C:\Users\Shaowen\workspace2017Spring\CS367\src\p3\sample\_data\weather\_data weather\_output.txt

weather\_output.txt should contain:

726506 20170201 18.7 25.8 9.4 7.7

726506 20170202 -3.5 9.9 10.0 9.4

726506 20170203 -4.1 - 10.0 7.3

726506 20170204 2.9 - 10.0 -

726506 20170205 - - 10.0 -

997340 20170201 - 33.4 999.9 -

997340 20170202 - 17.6 999.9 -

997340 20170203 - 16.5 999.9 12.3

997340 20170204 9999.9 22.0 999.9 8.5

997340 20170205 9999.9 32.6 999.9 10.8

**Note: we have used tabs above to help with visualizing the column structure of the data, but you should only use commas (',') to separate your fields in the output file, rather than tabs. So the actual output file will look like this:**

726506,20170201,18.7,25.8,9.4,7.7

726506,20170202,-3.5,9.9,10.0,9.4

726506,20170203,-4.1,-,10.0,7.3

726506,20170204,2.9,-,10.0,-

726506,20170205,-,-,10.0,-

997340,20170201,-,33.4,999.9,-

997340,20170202,-,17.6,999.9,-

997340,20170203,-,16.5,999.9,12.3

997340,20170204,9999.9,22.0,999.9,8.5

997340,20170205,9999.9,32.6,999.9,10.8

**Also notice that in every line, the different readings appear in the same order. This order is determined by the results of Java's**[**File.listFiles() (Links to an external site.)**](https://docs.oracle.com/javase/7/docs/api/java/io/File.html#listFiles())**method, which will be used to get every file in the directory which is to be merged.**

#### **Thesaurus files**

Assume that the following files are in the directory named weather\_data:

1.txt 2.txt 3.txt

------------------------------------ ------------------------------------ ------------------------------------

bad:lamentable,sorry,rotten bad:naughty,unfavorable good:satisfactory peachy,not bad

okay:fine good:acceptable,favorable,quality okay:alright

Upon running:

java Reducer thesaurus thesaurus\_data/ thesaurus\_output.txt

thesaurus C:\Users\Shaowen\workspace2017Spring\CS367\src\p3\sample\_data\thesaurus\_data thesaurus\_output.txt

thesaurus\_output.txt should contain:

bad:lamentable,naughty,rotten,sorry,unfavorable

good:acceptable,favorable,not bad,peachy,quality,satisfactory

okay:alright,fine

### Important Classes

#### **Reducer**

This is the driver class which parses its command-line arguments, constructs a Record object and performs the algorithm described above. The specifications for the command-line arguments are given above.

#### **FileIterator**

Returns the lines of a file one by one.

#### **FileLine**

Represents the string data from a single line in a file, as well as a reference to the FileIterator that produced it.

#### **Record**

 An abstract class which WeatherRecord and ThesaurusRecord must extend. Its methods are described below:

* void join(FileLine l): A Record must be able to accept a FileLine and "join" it, meaning incorporate it into its current data structures.
* String toString(): This method should return the String representation of the Record.
* void clear(): This method should (re)initialize the data structures used by the Record.
* Comparator<FileLine> getComparator(): This method should return a Comparator<FileLine>, i.e. a class which implements Comparator<FileLine>.

#### **WeatherRecord/ThesaurusRecord**

These classes should extend Record.

* WeatherRecord
  + Each time we write a line to the final output file, it will contain:
    - Station ID
    - Date
    - Comma-separated list of measurements
* ThesaurusRecord
  + Each time we write a line to the final output file, it will contain:
    - Word
    - Comma-separated list of synonyms
      * **Note:** these should be in sorted order, and not contain any duplicates. We will learn more about sorting later in the semester. For the time being, you can just use Java's Collection.sort(List) method, as shown in [this example file](http://pages.cs.wisc.edu/~deppeler/cs367/assignments/p3/files/SortExample.java).
* Think about what the above imply about the data structure(s) you will need to initialize/update in the join/clear methods.

#### **MinPriorityQueueADT/FileLinePriorityQueue**

The key to the above algorithm is having a (minimum) priority queue that stores FileLines. At each step in the algorithm, a FileLine is removed from the queue and potentially replaced.

It’s your job to create the queue such that it maintains an array-based minimum priority queue (see the notes on Priority Queues) of these FileLines. The class must implement MinPriorityQueueADT<FileLine>:

   public FileLine removeMin() throws PriorityQueueEmptyException;  
    public void insert(FileLine fl) throws PriorityQueueFullException;  
    public boolean isEmpty();

**Note**: You must also implement the following constructor:

public FileLinePriorityQueue(int initialSize, Comparator<FileLine> cmp);

What makes a FileLine a minimum element? In order to maintain the **order property** of the queue, you **must** use the compare(FileLine a, FileLine b) method of the Comparator<FileLine> passed in the constructor.

**Notes:**

* For **removeMin()** you must throw a **PriorityQueueEmptyException** if the queue is empty.
* For **insert()**, you must throw a **PriorityQueueFullException** if the queue is full.

### Documentation

The javadocs for all classes provided to you are [here](http://pages.cs.wisc.edu/~cs367-1/assignments/p3/javadocs/index.html). You are expected to complete the sections marked as "**// TODO**".

* You may **not** add any other public methods than those listed in the provided files.
* You may **not** modify any class in any way except where noted as "**// TODO**".

## Steps

After you have read this program page and given thought to the problem we suggest the following steps:

1. **Download this**[**p3.zip**](http://pages.cs.wisc.edu/~cs367-1/assignments/p3/files/p3.zip)**file to a programming assignment p3 folder that you make.** Unzip all the files in your p3 program folder.
2. Individually, draft a task list like the one used in program 2. You can use this list as a starting point.
3. Make a list of each of the classes you are given and identify their main purpose.
4. Draw diagrams of the data stored by each class to help you understand which classes are responsible for which parts of the problem.
5. Identify which classes and methods need to be completed.
6. Identify sub tasks and tests that can help ensure each part works as intended.
7. **Incrementally** implement the methods and thoroughly test methods and classes.
8. During development, hand in partially complete versions of your program as a backup in case your computer crashes and work is lost.
9. and then run your program using java as in:
   * java Reducer thesaurus thesaurus\_data/ thesaurus\_output.txt

## Submitting Your Work

* Make sure your code follows the [style](https://canvas.wisc.edu/courses/23073/pages/program-style-guide) and [commenting](https://canvas.wisc.edu/courses/23073/pages/program-commenting-guide) standards.
* All submitted classes must belong to the default package. (No package declaration at top of class).
* **Late work is not accepted.**
* Please submit only the following files:
  + FileLinePriorityQueue.java
  + Reducer.java
  + ThesaurusRecord.java
  + WeatherRecord.java