Compuer Science 62
Assignment 3

Due 11:59pm on Tuesday, February 16, 2010

All of the sorting algorithms we've looked at so far assume that the data is in memory and that we can swap data elements efficiently. Sometimes, because of the size of the data you cannot fit all of it in memory. In these situations, many of the traditional sorting algorithms fail miserably; the algorithms do not preserve data locality and end up accessing the disk frequently resulting in very slow running times.

For this assignment, you will be implementing an on-disk sorting algorithm that is designed to use the disk efficiently to sort large data sets. The sorting algorithm will work in two phases:

- First, your sorting algorithm breaks the data into reasonable sized chunks and sorts each of these individual chunks. This is accomplished by reading a chunk of data, sorting it, writing it to a file, then reading more data, etc. At the end of this phase, you will have a number of files on disk that are all sorted.
- Second, you will need to merge all of these files into one large file. This is accomplished by pair-wise merging of the files (very similar to the merge of MergeSort) and then writing out the result to a new, larger merged file. Eventually, all of the files will be merged to one large file. Note, this can be done very memory efficiently.

1 Getting started

As usual, create a new project in `Eclipse` and copy the files over from `/common/cs/cs062/assignments/assignment3/` in to the source directory of your newly created project.

2 On-disk sorting

I have provided you with a skeleton class that you will need to fill in the details for. I encourage you to add additional private methods, but **do not** change the names or parameters of the methods I have provided you. This will make our life much easier when we grade the assignment. As an aside, I have made some of the methods `protected` where normally I would have made them `private` to, again, assist us in grading.

You will need to fill in the following methods:

- `OnDiskSort`: the constructor for the class. Make sure that you understand what all of the parameters do. `maxSize` is the maximum number of Strings that can be read in to memory at any one time. You will need to created temporary files along the way (for example, to store the sorted chunks). This should be done in the `workingDirectory` directory (which is also just a `File`). Make sure you clear the working directory when you're done. `sorter` is the sorter that you should use to sort each chunk. `outputFile` will contain the final result of your sorting.

- `sort`: this is the public method that will be called when you want to sort new data. For this assignment, we will only be sorting String data (notice that `WordScanner` is an `Iterator<String>`). This method will read in the data a chunk at a time, sort it using the sorter and then call the `mergeFiles` method to merge all the sorted files.

- `mergeFiles`: takes an ArrayList of Files, each of which should contain sorted data and then uses the `merge` method below to eventually merge them into one large sorted file. Notice that the `merge` method only merges two files at a time. The easiest way to merge all of the n sorted files is to merge the first two files, then merge the third file with the result of merging the first two files, then the fourth in, etc. This is *not* the most efficient way of doing it, however, it will make your life easy (see the extra credit for doing it a better way). NOTE: you cannot read and write to a file at the same time, so you will need to use another temporary file to store your temporary results as you merge the data.

- `merge`: take two sorted files and merge them into one sorted file. This is very similar to the `merge` method of MergeSort. The main difference is that rather than merging from two arrays (or ArrayLists) you are merging two files. You **should not simply read in the data from both of these files and then use the merge method from MergeSort**. We are trying to be memory efficient and this would defeat the purpose. Instead, you should open `BufferedReader`s to both of the files and then, reading one line at a time, read either from the first file or the second and write that directly out the the output file, depending on the appropriate ordering. Besides the variables for doing the file I/0, you should only need **two** String variables to keep track of the data.

To assist you, I have also provided a few helper methods in the `OnDiskSort` class that you may find useful. If there is any confusion about what these methods do, please come talk to me. In addition, these helper methods may also help you understand basic Java file I/O.

3 Extra Credit

As I mention above, this is not an efficient way to merge all of the sorted chunks. Once you have things working above, for extra credit, implement a more efficient `mergeFiles` method. This is optional and you do not have to do it.

If you do this, I strongly suggest making a new method (i.e. don't delete your original `mergeFiles` method, just rename is to something like `mergeFilesLinear`). If you do the extra credit, put the phrase "EXTRA CREDIT DONE" in a line by itself in the class header under the @date tag, so the TAs know to look more closely at your method.

When You're Done

Submission

Follow the directions on the course web page for submitting. You will only need to submit your `OnDiskSort.java` class file (though it should be in a .jar file and in an appropriately named folder). Be sure that your code is clear, formatted properly and commented appropriately (using Javadoc... see the first assignment for details on what's expected for comments).

Grading

You will be graded based on the following criteria:

criterion	points
functionality/correctness	12
cleans up temporary files appropriately	1
appropriate comments (including JavaDoc)	3
appropriate use of generics	2
style and formatting	2
submitted correctly	1
extra credit	2

NOTE: Code that does not compile will not be accepted! Make sure that your code compiles before submitting it.

Some useful information

- File I/O in Java

  For those that haven't had any file I/O experience in java, I'll give a brief intro here, but also take a look at the course notes for cs51 from last semester about streams and file I/O (you can find a link to it off my home page) and you can also look up information about the classes seen in the code and discussed here online at java.sun.com. For most I/O, you'll need to `import java.io.*`.

  The two main classes you'll be concerned with when doing file I/O in java are `BufferedReader` for reading data and `PrintWriter` for writing data. To read data, you can create a new reader by:

  ```
  BufferedReader in = new BufferedReader(new FileReader(...))
  ```

  where "..." can be either replaced with a `String` or can be replaced with a `File` object.

  To write data, you can create a new writer by:

  ```
  PrinterWriter out = new PrintWriter(new FileOutputStream(...))
  ```

In both cases, you will need to surround these with a try-catch to handle the `IOException`.

- The file system

  The file system on these computers starts at the very base directory of '/'. Everything is then expanded out based on directories. For example "/home/dave/" is two directories starting from the base, first "home" then "dave". The '/' is called the file separator and is different depending on the operating system (e.g. it's '\' on windows computers). Filenames can be specified as *relative* filenames, where they are relative to the current location of the program (or user). Relative filenames do NOT start with a '/'. It can be confusing telling exactly where you program currently is when running it, so the best approach when writing programs is to use a *full* path which starts at the base directory. If you ever want to know where you are when you're in the `Terminal` is the `pwd` command (try it out, but just typing it and hitting return!).