University of Toronto
csc228 — **File Structures and Data Management, Spring 2000**

Assignment 1 — Sorting a Large File

Part A Due: at the beginning of tutorial, Friday 21 January, 10 am.
Part B Due: at the beginning of tutorial, Friday 04 February, 10 am.
Total weight: 10% of your course grade. Part A is worth a small fraction of this total weight.

You will do the assignment in two parts. Part A is merely a step on the way to part B. It will be handed in for marks simply because I want to help everyone avoid procrastination. `:-)`

The purpose of this assignment is to introduce you to a classic file processing algorithm and to make sure you are comfortable with basic file input/output in C^{++}. It will also use some features of C^{++} that you may be somewhat unfamiliar with, including templates, and the standard template library. We will spend a little time on these topics in class, but be prepared to consult your C^{++} book for more information.

Introduction: External sorting

Suppose that we wish to sort the contents of a file that is so large its contents will not fit in memory. The familiar approach of reading the contents of the file into some internal data structure, performing a sort algorithm on the data structure, and writing the resulting sorted list back out to a file, is not possible. We need instead to do some kind of "external" sorting — sorting that doesn't take place entirely in memory.

Here is one general strategy for external sorting:

1. Break the file into "partitions", each of which is small enough to fit in memory.
2. Sort the partitions individually, writing each out to a file.
3. Merge the partitions into one sorted file.

This strategy side-steps the memory problem by sorting individual partitions, each of which is small enough to fit into memory. And the merging step doesn't introduce any memory problems because it can be done using only a constant amount of memory, no matter how big the partitions are or how many partitions exist. (Think about why.)

In this assignment, you will write a C^{++} program that performs external sorting using this general "sort-merge" strategy. See chapter 4 of Smith and Barnes for a detailed discussion of various sort-merge methods, as well as other methods of external sorting.

Assignment details

Input and output file format

The input file is a text file containing records, each on a separate line. The records have the following format:

- student number, a 9-digit integer
- name, a string of 20 characters
- grade-point average, a floating point number with one digit before the decimal point and two after it

For example, an input file might contain:

```
860293823Chris Beck          3.25
930048382Karen Cheng         2.90
880948640Stephan F. Jou      3.01
```

You can find a copy of this small sample file on the course web site. The final output file must follow this format, but also be sorted in non-decreasing order by student number.

Program interface

Your program should not be interactive. The name of the input file, the output file, and the partition size (*i.e.*, the number of records per partition) should be command-line arguments to your program — arguments that the user gives when calling your program.

The original input file must be left intact by your program, but your program must remove any temporary partition files that it creates.

You need not do a lot of error checking in your program, but you should specify in your report what error checks would ideally be done, and which function would be responsible for each.

Overall algorithm

Your overall algorithm should be:

- Read a partition-sized chunk of the input file into memory and sort it, using your own sort function[1]. Any algorithm that is $O(n \log n)$ in the worst case (where n is the size of the partition) is fine.
- Then print the sorted partition out to a temporary partition file. The C library function `tmpnam` will provide a name for your temporary partition file. See King, pages 483–4; Kernighan and Ritchie, page 243; your own C book; or `man tmpnam`.
- Store the size of this partition and the name of the file containing it in a priority queue (see below).
- Continue doing this until the entire input file has been broken up into sorted partitions. Note that the final partition may be smaller than the others.
- Now all of the partition files must be merged. Remove the two smallest files from the priority queue. Merge those two files, putting the output into a new, larger partition file. Again, use `tmpnam` to create a name for the file. Add information about this new partition file to the priority queue.
- Remove from disk the two temporary files that have just been merged. The C library function `remove` can be used to do this[2] See King, page 486; Kernighan and Ritchie, page 242; your own C book; or `man remove`.
- Repeat the merging process until there is only one file left. During the last merge phase, put the output into the final output file rather than a temporary file.

The priority queue

Your partition information is to be kept in a "priority queue". A priority queue is like an ordinary queue, except that when items are removed they come out in order according to their "priority" rather than according to which one entered the queue first. If there is a tie for top priority, the item that entered the queue first (among those that are tied) comes out first.

"Priority" can be anything according to which the items can be ordered. For example, you could have a priority queue of customer purchases, in non-increasing order by price. This would mean that when you remove something from the priority queue, the purchase with the highest price would come out first. If several purchases have the same highest price, the oldest of them would come out first.

You should write a class that implements a priority queue, and use an instance of the class in your sort-merge program. Your priority queue class should allow one to create a priority queue of strings, or of integers, or of partition-information, or anything else. Use a template to make it this general.

You must implement your priority queue using a linked list, kept sorted by priority.

[1] The ultimate way to sort a partition would be to use the C++ function `sort`. However, it involves familiarity with some advanced features of the language, and so you are not required to use it.

[2] In order to use `tmpnam` and `remove`, you will have to be able to convert between a C++ `string` and old-fashioned, C-style string (*i.e.*, an array of characters with a 0 terminator.)

Other design decisions

Any other decisions about the data structures and algorithms are left to you. Be sure to document your choices in your report.

If you notice a problem with the specifications, or if you feel you need more guidance, contact me.

Developing your program

I recommend that you implement the program in phases, testing your program at each phase before moving on. Below is a suggestion for one possible strategy. Of course, you should feel free to devise your own; my point here really is to encourage you to have *some* strategy.

It's probably best to first lay out the object hierarchy and the interface for each class. You can even compile this skeleton code.

When you are ready to put some meat on the bones, you might begin with the priority queue. Since templates are new to you, you could write your priority queue class without them, for example, making a priority queue of integers. Once you have that working, you can generalize the code with templates.

When you move on to the sort-merge program (which will use your priority queue), you could tackle it one function at a time. Another strategy, which you might not have thought of, is to write a program that solves a simplified version of the problem, and to add more details to it until it does the entire thing. For example, you could:

1. Write a program that simply grabs the command-line arguments that you are to accept, and prints out their values.
2. Extend the program so that it opens the input file and simply prints out its contents, one record at a time.
3. Extend the program so that, instead of printing the contents of the input file, it breaks the file into partitions, writing each one out to a temporary file.
4. Extend the program so that it sorts each partition before writing it out.
5. Extend the program so that it inserts appropriate information about each new partition into a priority queue.
6. Extend the program so that it prints out the names and sizes of the two smallest files in the priority queue, and then deletes them from it.
7. Add to your program an appropriate call to the merge function, to create a new partition, and appropriate updates to the priority queue,
8. Extend the program so it repeats the merging process until there is only one file left.

What to hand in for part A

There is no specific requirement for Part A; It is merely a step on the way to part B. You should have made a good start on the program, and have some components working. You may need to create simple driver programs to test your code at each phase of development. Hand in a printout of your code (including any driver programs), and some test runs that demonstrate what works. Your code needn't work flawlessly for you to get some marks on part A. Do not hand in a report with part A.

If you have designed and implemented the priority queue, or completed step 4 above for the sort-merge program, then you will be in good shape for part B and will get a good mark on Part A. Do not despair if you don't get this far. The marking scheme for Part A will be fairly generous, and it is not worth a great deal of marks.

Your tutor will evaluate your work on this part by looking almost exclusively at your test runs, so be sure that they demonstrate what you have working. Explain your test results with brief, hand-written comments on your printout.

What to hand in for part B

For part B, you are to hand in your assignment in the form of a report, as described in the 228 Course Guide. The style of a 228 report is different from what you may be used to from earlier courses, so be sure to re-read the relevant sections of the Guide before getting far into the assignment. Your report must include all of your code and all relevant input and output test files.

Remember to justify the decisions you made in designing your program. This is a very important part of your report.

Your report should be thorough, well-written, concise, and honest. It should be no more than about five pages long, not including code and tests. Make sure that you run your report through a spelling checker.

You are to devise your own testing for this assignment. Although the program is designed to handle a HUGE input file, you can test it on relatively small ones. You will be marked for testing according to the standard set by the document on “Software Testing”, available on the course web site. Also note that if your testing is weak, you will lose marks not only for testing but also for correctness; the only way we will know that you program works is if you *demonstrate* that through your testing.