Extra Credit Program 2: Parallelizing Maps using Pthreads

General Description: You, and optionally a partner or two, will build a map data structure (or two) to handle many insertion/deletions/lookups as fast and correctly as possible. A map (also called a Dictionary ADT) supports insertion, deletion, and lookup of (key,value) pairs. You may do this one of two ways (or both ways if your group has three members). Lone wolves will get graded somewhat more leniently but they will have more to do:

1) General Constraints:
   a) The map may hold as many as 2^{22} items *at one time*. The map stores (key,value) pairs.
   b) The key will be an int.
   c) The values will be some sort of C-style string, with no fixed/max length.
   d) The keys will be randomly chosen and unique. Each key may be
      i) Inserted once before deletion. An insertion to the same key, if already present, should fail unless there is a deletion in between.
      ii) Deleted. It is possible for a deletion to "arrive" before inserting if you're careless with multiple threads. Deletion should fail if the key does not exist
      iii) Lookup/search: You may lookup items in the map that exist or not.
   e) You may assume the keys are random.
   f) Your input will be a relative path to a (potentially large) text file having lines of the structure
```
N 4
I 34 “fred”
I 35 “is”
I 36 “happy”
I 36 “tired”
D 34
L 35
L 36
L 63
L 34
[…]
```
   g) Your output should be
```
Using 4 threads
OK
OK
OK
Fail
Ok
is
happy
No 63
No 34
```
   h) Your output must be identical to the single threaded version.
   i) Your performance (scaling) will be judged on input where there are not a lot of changes to the same key within some reasonable number of inputs (<33) but *must not break* in the pathological

cases where there are a large number of changes to the same key, e.g. alternating insertions and deletions to the same key. The latter case and cases like it will only be judged only for correctness, not speed.

j) You should consider a higher level queue of some sort for the threads to read from to ensure that the order of the map operations is correct. You may use one extra thread for this, if you like, perhaps the main thread, or let the new threads handle this. You may also consider writing the output while it is being generated.

k) Your program should compile to a **mapper** executable that takes its arguments as **mapper <input file relative path> <output file relative path>**

2) Using a Hash Table
   a) You do not have to deal with rehashing (resizing the hash table).
   b) You must assume a maximum number of elements (2^{20}) as a default. You may, of course, set it for testing purposes. Note that you may have to deal with up to 2^{22} items.
   c) You should do open addressing (each entry in the table is a possibly null pointer to a linked list). Closed addressing (where the items go "directly" into a hash table) will not work; see 1.a&1.b).
   d) You will probably want to extend the linked list code given in class and the book to handle deletion and lookup.
3) Using a Balanced Search Tree (BST)
   a) BSTs "resize" on demand.
   b) The keys are random so you we won't worry about sorted data. (1c)

Language options: C or C++, using Pthread locks, Condition variables, or Semaphores. It should run on the department Lab machines.

For groups, decide on a division of labor. You may just work together and get one grade (the default, if you don't tell us) or be divide up in to portions. Suggested division of labor:

1. Tester
   a. Develops the tests.
      i. Consider using GTest. If you do so, include the GTest library (so we don't have version issues).
      ii. If you do not use GTest, create a separate executable to run the tests. Score is based on how extensively you test.
      iii. For (Unit) Testing, either way, you probably want to have code that has examples of already parsed data in whatever form you want as it may be annoying to parse the command line arguments from the compiled program and somewhat violates the goal of unit testing – to test the individual "units" of the program.
      iv. You should time some tests, to demonstrate scaling. GTest has a basic version of this.
      v. Test for both correctness and speed.
   b. Performs the performance evaluation
   c. Principle person responsible for developing the main function code, which
      i. Loads and processes file
      ii. May manage a work queue of some kind
      iii. Outputs results to a file

2. Either BST map or Hash Table Map (for the other person). If you have a group of three, the other two people each pick one. For groups of three, I strongly suggest you agree on the interface early so the tester can duplicate tests for both versions easily and perhaps develop this together, to get a basic test going.

Deliverables:

1. A README(.md) file showing
    a. How to compile and run your code
    b. Describing your approach
    c. Describing your testing
    d. Categorize your performance on reasonable input
2. A makefile or CMakeLists.txt
    a. You may consider using GTest. If you do, it may be easier to use CMake.
3. The source, obviously.