Your work for this must be finished and shown to your lab TA by the end of this lab session or the start of your next session.

Note: You will need to use the department servers for this lab.

You won't be able to do Lab10 on your own machine, you need the Department's servers.

On on one of the department servers (or use the lab machine, or XManager... you know the drill).

1. Download the modified version of the AVL tree code you used in Lab8 (along with associated files) from the link on the Course Webpage ("Lab/Lecture Notes" > "Lab10" > "lab10.zip"), and extract the source code into cs221/lab10 (a new subdirectory on your undergrad account) – again, you know the drill.
   Doxygen-style documentation, similar to what you had access to in lab8, will be available Wednesday afternoon from a link near the "lab10.zip" link cited above, and on Piazza.
2. There is no Unit Class in this lab (so there are no unit tests), but there is a Timer Class, which is declared and defined in Timer.hpp and cpp. And the Makefile we're giving you is extremely versatile.

   In the avl.cpp program itself, there's a pound-defined variable called "REP" that has initial value of 1. It's the number of repetitions you'll run for the sequential/recursive and parallel/fork-join approaches to the problem. You'll want to play with this number a bit. One trial is not enough.

   A little further down, in the function prototypes, notice that findMostCommonAndReport() takes, as one of its parameters, the signature of a function!

   After that, except for the "pragma" statements you'll need for the fork and join, it's fairly straightforward code. I've replaced a couple of tricky bits that I'll go into more on a Piazza post. You don't need to know about them for this lab.

   The program will count the occurences of each word in a txt file. After the tree is loaded, balancing itself as it goes, findMostCommonAndReport() will find the most common word and report what that word was, and how long it took to find it.

3. The program is fully implemented except for the findMaxHelper(Node*,KType,VType) procedure which is called by findMax(Node*, KType, VType) in order to find the maximum "value" in the tree using divide-and-conquer fork/join style parallelism.

   Your solution should fork off tasks to handle subtrees until you can guarantee by looking at the root node of a subtree that it has at most 1100 nodes, at which point it should switch to the sequential/recursive version, findMaxSequential(Node*, KType, Vtype), to complete that portion of the search instead. Be prepared to briefly justify your cutoff to your lab TA.

4. Find a lightly used linXX.ugrad.cs.ubc.ca machine. What's a linXX?? Enter this on the command line:
   lin-uptime

   SSH into a machine (while already logged into your account, because the lin machines are only available behind the firewall). Time your solution on that machine using the sample text files that are in:

   ~cs221/assignments/parallel-lab-resources. (You should have access to them from you ugrad account.)

   (We provide the King James Bible because it's a big file available from Project Gutenberg and one tiny piece of Google's ngram corpus from its scanned collection of texts, because it's bigger.)

Your work for this must be finished and shown to your lab TA by the end of this lab session or the start of your next session.

5. **Note that you do not need to copy these files!** Just pass the path to one of the files as the command line argument to your avl program. What are the most common words in these files?
6. Finally, graph the relative runtimes of the sequential and parallel versions for a reasonably wide range of differing sequential cutoffs on the two sample files that we provide and explain the results that you find. (Work with the Bible first, as it's a much more manageable size. Then, spend a few minutes getting the data points you need for the Google ngrams file.)

7. For no bonus credit, but a substantial amound of street credit, create new sequential and parallel versions of a function that finds the top 20 most common words rather than just the single most common word. Comment on the relative performance gains for the parallel version with this task.
8. For even more street cred, figure out how we changed the Google data.