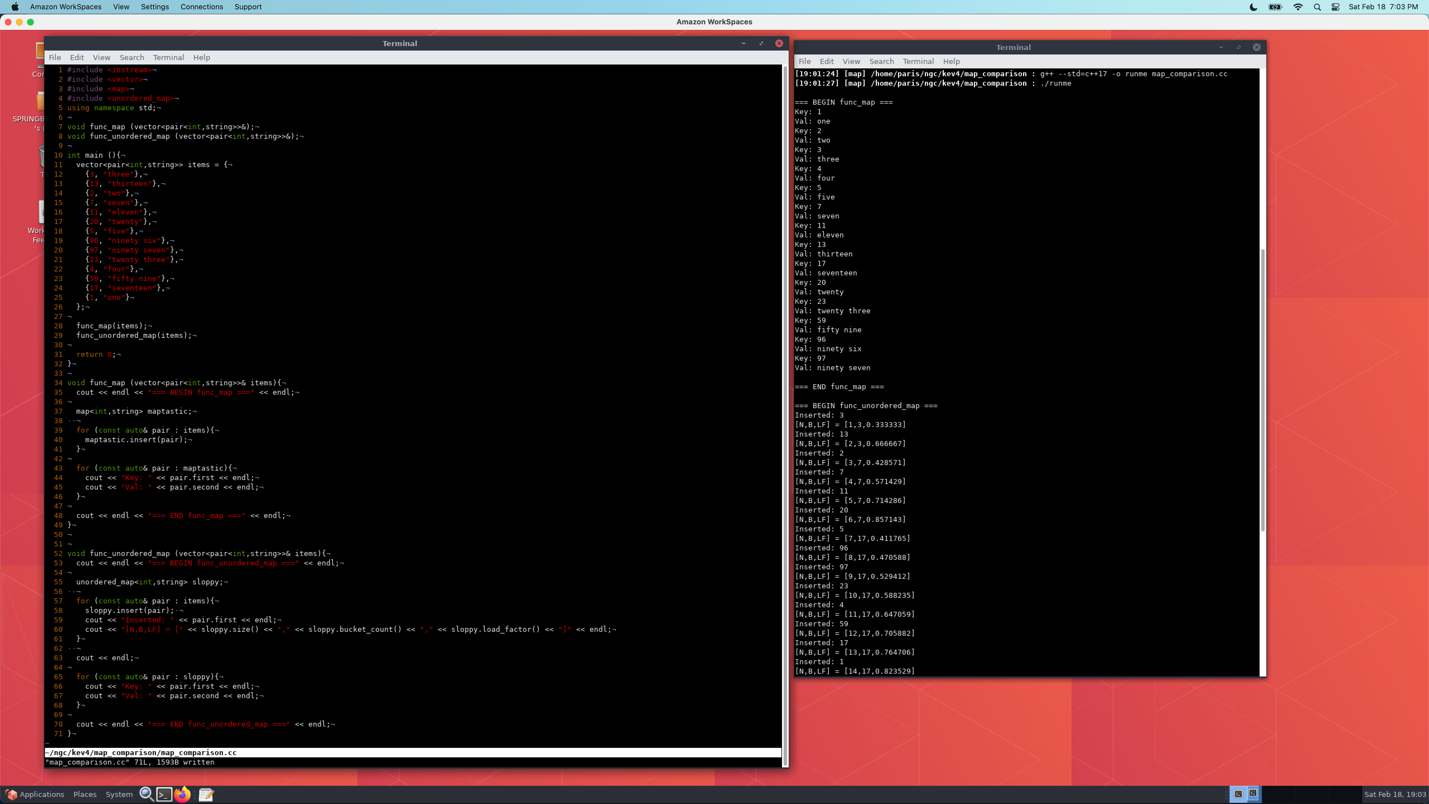
*Exercise 2: Maps(map\_comparison.cc)*

***Explained***

* This exercise asked us to compare maps vs unordered maps in C++.
* Initially I noticed that, syntactically, these were similar except that regular maps (aka ordered maps) do not have “buckets” or “load factors”.
* Side note: These maps are also extremely similar to the vector pair provided in main.
* After doing a little digging, ordered pairs are just like they sound. They’re essentially hash maps (dictionaries) that automatically sort the data as long as it can be compared (even if the key is a string). Apparently, these have a O(logN) retrieval time, but this is dependent upon the size of the data! One should probably not use these for super large datasets.
* When there is need for such a large data that doesn’t need to be ordered, the classical HashMap (unordered map) is the best bet. This provides a constant look up time and does so by distributing its values into buckets.



* The pair vector provided was not in order, yet the ordered map still managed to sort this for us without having to explicitly tell it to (the unordered map did not do the same).
* The bucket size of the unordered map seems to increase only once it is filled/overflows (which makes sense). Meanwhile the load factor’s behavior is not immediately obvious to me…
* Apparently the “load factor” of an unordered map in C++ is a measure of how full the hash table is displayed as a floating-point value. Well why is it displayed as a floating-point value Paris?
* The reason it is displayed as a floating-point value is because it is calculated by taking the # of elements in the hash table and dividing them by the number of buckets in the hash table.
* This explains why this pattern seems to increase as the bucket fills up, then when it is time to move to a new (much emptier) bucket the value drops back down.

