This is an optimization and data scalability problem.  Assume you’re tasked with building a system to track entries, where the number of entries will be in the millions.  Each individual entry could be of any length, from 1B to 1MB.  Assume the total data size is small enough to fit in one physical system of 1 TB RAM.  The data is fully memory resident (ie. No Disk IO necessary).

Consider the “word” to be equivalent to entry and “dictionary” to be equivalent to solution. Where “word” can be any collection of ASCII characters, (0-254). For example, any and all combinations of letters, numbers, and symbols representable in standard ASCII format.

There are many ways to implement this program.  The focus is on how data structures are used to solve the problem and which algorithms are combined to achieve an optimized solution.  The solution must take {time, space} complexity into account.  Thus be able to describe the cost (O-Notation in space and time) of each, and tradeoffs in design/implementation. Also consider abstract data types and libraries for modularity, and how this can be used to facilitate long term serviceability, maintainability, and diagnosis.

Create a userspace program in C to support the following services.  The program does not need to be proven at 1TB scale, but the design must take said scale into account.

* 1. int word\_add( char \*p\_word )
     1. adds a word to the dictionary
     2. duplicate entries possible thus track with references
  2. int word\_remove( char \*p\_word )
     1. removes a word from the dictionary or decrements reference (yes, duplicate entries are possible).
     2. returns -1 if word does not exist
  3. bool word\_find( char \*p\_word )
     1. returns True/False if word exists
  4. struct \*stats word\_stats( \*p\_word )
     1. returns current entry count (ie. Duplicates)
     2. returns max entry count ever (ie. Duplicates)
  5. struct dictionary\_stats( void )
     1. returns current longest word (ie. in dictionary)
     2. returns longest word ever (ie. in dictionary)
  6. struct \*words word\_get\_anagrams( char \*p\_word )
     1. returns a list of all words on the dictionary which are anagrams of the input word.
     2. returns NULL if no anagrams present

Caveat here is that primitives such as Trees, Tries, HashTables, Stacks, Queues, Lists, etc. are not available in standard C. You’d have to build these out and associated validation. I suggest you focus on the design and algorithms can simply stub the calls to the primitives. Bonus points for implementing them if you decide to go that far.

Feel free to send over any questions you might have as you work through the problem.