Homework 3

MPCS 51100

due: Nov 22, 2016 @5pm

Please package your assignment in a directory called hw3 with subdirectories p1, p2, and p3. At the top level, under directory hw3, include a makefile with targets p1,p2,p3,all, and clean.

1. Implement a dictionary similiar to hw2 using a hashmap rather than a BST. Your dictionary must support:

   - %find [key]
     - print value (definition) associated with key
   - %delete [key]
     - removes key and associated value (definition) from dictionary
   - %insert [key] [value]
     - add word/definition pair to the dictionary
   - %find [key1] [key2]
     - print word/definition for each word between key1 and key2, inclusive
   - %print
     - print all word/definition pairs in alphabetical order of words
   - %read [filename]
     - read in a plain text file of words and definitions and add them to the dictionary. Assume that each distinct word and definition pair is on a separate line, the word is separated from the definition by a single space and the definition is enclosed in double quotes i.e. word this is the definition ; word2 this is the second definition

   You can assume no word will be longer than 50 characters, and no definition will be longer than 500 characters.

   Since hashmaps do not preserve key order, you will need to think about how to implement interval find and in order print functions. It is understood that your solution will not be efficient for hashing, but it is expected to work correctly.

   Have your code create a log file (write to a text file) that records the load factor and occupancy after each operation.

   Implement three different hashing algorithms h(k), chosen either from our class examples or from the course readings. Be clear about the chosen value of f in g that make up h – ie h(k) = f(g(k)).

   The hashtable must support rehashing to potentially execute in O(1). You must choose a rehashing strategy and explain the choice.

   Benchmark your implementation against the test cases supplied and provide an overview of results.

   Note that the provided tests are not necessarily the test files used for grading.

2. Write a function that uses hashing to efficiently remove duplicates from an array of objects.

```
void uniq(void *f, int n, int sz, int (*equals)(void *, void *))
```

where f is the original array of n objects each of size sz, equals returns true when two objects are equal, and f is the array with duplicates removed upon return from uniq. Include a test driver that demonstrates uniq with any object of your choice.

3. This problem builds off of Problem 6 from the previous assignment. Consider a fourth data structure and lookup method that can be used to locate the bounding indices of the key q in each of the sorted arrays $\{L_1, L_2, \ldots, L_k\}$. This method uses a technique similar to hashing to narrow the bounds of the binary search in each array. The data structure is created in the following way:

   - Determine the bounds of the problem, i.e., find the minimum and maximum values over all arrays, x_{min} and x_{max}.
   - Choose a value for the number of bins m. The grid U will then be defined as the grid with m bins with uniform spacing between x_{min} and x_{max}.
   - For each bin in U, determine and store for each array the bounding indices b_1 and b_2.

   A given search then starts by "hashing" the search key into an integer bin using integer division (since the bins are divided equally by energy value). Since the bin indices are pre-computed, the search is then immediately narrowed to the given bin rather than taking place over the entire grid. The smaller the bin the fewer values need to be searched, but the larger the memory overhead. Write your code so that the binsize can be adjusted to experimentally find an optimal value.

   (a) Implement this data structure and a lookup kernel.

   (b) Using the data provided in the previous assignment, perform a study to determine the optimal number of bins.

   (c) Compare the performance of this method to the third method from the previous assignment.