CSE/IT 122: Homework 6
Hashing

Download `hw6b.tar.gz` from Canvas. It contains a number of files you need for this homework.

Follow the Linux coding style and make sure you comment your code using Doxygen style comments.

Any dynamic memory allocation should be checked with valgrind for errors. Make sure you free memory correctly in all problems that use dynamic memory techniques.

Do not change any provided structures.

Follow any sample output exactly.

Problems

1. Hash the information in the file `postal` using a open addressing scheme with quadratic probing and a table size of 200. The file `state_hash.c` is provided to get you started but it is far from complete.

   You will use Bernstein's function to convert the strings into a numeric key and then hash it with a simple hash function $h(k) = k \ mod \ tablesize$ where k is the key. See the appendix for a discussion of Bernstein's hash function.

   You will store both the two digit state abbreviation (uppercase) and the name of the state in start case (every letter of every word is capitalized and the rest of the word is lowercase). Note there are more than 50 "states" in the file.

   After you initialize the hash table, write a menu to enter two digit state abbreviations and perform a lookup to find the state's name and print that to the user. Your output should look like this:

   ```
   Enter a two-digit state abbreviation (q to quit): FM
   Federated States Of Micronesia

   Enter a two-digit state abbreviation (q to quit): FX
   State not found
   ```

   The file won't parse easily with `strtok()` as it uses a random number of spaces as the separator. Read the manpage for `memcpy()`. You should be able to come with a solution

using `memcpy()` to split the line into the state name and its two digit abbreviation. Don't delete anything from the file, hash all 59 two digit US postal codes.

In addition to the menu option create a debug print function that prints the hash table. Skip over any index that contains NULL values. Your output should look like this (that is a tab between the abbreviation and the name and i is the index where the element is located):

```
i = 6
MA      Massachusetts

i = 7
SD      South Dakota
```

2. Replace the scrabble lookup to see if a word is in the scrabble dictionary with a hash table. Use chaining for the hash. As in the previous problem you will will use Bernstein's function to convert the dictionary word to a key and then hash the key as $h(k) = k \ mod \ tablesize$.

   You will pass in the table size as a command line argument.

   You can use singly linked lists for the lists. Use the given data structures in the file `scrabble_lookup.c`. `dict` is an array of pointers to the sentinel node.

   You only need to insert at the head of the list. Use the provided sentinel structure list_t and also keep track of the number of nodes in the list. There are no deletions so the list functions are minimal (create a node, insert into the list, print the list, delete the list, etc.).

   You need a means to dump the index and the linked list. The file `scrabble50000` provides sample output (follow this format) for the dump of the hash table.

   Here are the first and last few lines (head/tail -n 10 scrabble50000; some lines are truncated):

```
    0: freebases
    1: harpooners, endogenies, defilades, daman, brittanias
    2: viceregal, reasserted, intent, charros
    3: spelunkings, outcrow, listenable, illumined, czars
    4: tush
    5: serially, revivifications, foamflowers, fidos, damar, clausal
    6: semisweet
    7: tusk, mooncalves, knotholes, cellulars
    8:
    9: sporozoon, neglecter
...
49991: needleworker
49992: pahlavis, bubinga
49993: revest, bisque
49994: zymology, rhizoctonia, milnebs, lyrists, creminis
```

```
49995: pitsaws, neglected, cockerels
49996: wanion, vizslas, stagflation, prefabricated, pottering, outcrop
49997: unceasing, trolleybuses, decamps, castaway
49998: seducers
49999: unsplit, sargos, freebaser
table size 50000 load factor: 3.57
```

In this case, you are printing out every index value, whether there is a list or not. For empty lists just print the index, e.g. index 8.

Note the calculation of the load factor.

You will capture the output and load factor of running the hash for a table size of 30000, 50000, and 80000. Name the files scrable30000, etc. Just use redirection to capture these files.

Important: as we will use diff to compare your output with the solution, follow the format exactly. The index numbers are right justified with 5 spaces (%5d), followed by a colon, a space, and then the words in the linked list separated by commas except the last one which is terminated with a newline. And then the table size and load factor is printed to 2 decimal places.

You only need a menu option for the lookup of words.

Submission

Tar your source code, Makefile, and output files into a file named:

`cse122_firstname_lastname_hw6b.tar.gz`

and upload to Canvas before the due date.

Appendix - Bernstein's Hash

From `http://eternallyconfuzzled.com/tuts/algorithms/jsw_tut_hashing.aspx`

"Dan Bernstein created this algorithm and posted it in a newsgroup [comp.lang.c]. It is known by many as the Chris Torek hash because Chris went a long way toward popularizing it. Since then it has been used successfully by many, but despite that the algorithm itself is not very sound when it comes to avalanche and permutation of the internal state. It has proven very good for small character keys, where it can outperform algorithms that result in a more random distribution:

```
/* not the same as the code posted on the web page
 * a pointer version */
unsigned djb_hash(void *key)
```

```
{
        unsigned char *p = key;
        unsigned h = 0;

        while(*p) {
                h = 33 * h + *p;
                p++;
        }

        return h;
}
```

"Bernstein's hash should be used with caution. It performs very well in practice, for no apparently known reasons (much like how the constant 33 does better than more logical constants for no apparent reason), but in theory it is not up to snuff. Always test this function with sample data for every application to ensure that it does not encounter a degenerate case and cause excessive collisions."

And there is this comment regarding Bernstein's hash from Apache's Portable Runtime (APR) project. The comment is found in the file `apr_hash.c`:

```
/*
* This is the popular `times 33' hash algorithm which is used by
* perl and also appears in Berkeley DB. This is one of the best
* known hash functions for strings because it is both computed
* very fast and distributes very well.
*
* The originator may be Dan Bernstein but the code in Berkeley DB
* cites Chris Torek as the source. The best citation I have found
* is "Chris Torek, Hash function for text in C, Usenet message
* <27038@mimsy.umd.edu> in comp.lang.c , October, 1990." in Rich
* Salz's USENIX 1992 paper about INN which can be found at
* <http://citeseer.nj.nec.com/salz92internetnews.html>.
*
* The magic of number 33, i.e. why it works better than many other
* constants, prime or not, has never been adequately explained by
* anyone. So I try an explanation: if one experimentally tests all
* multipliers between 1 and 256 (as I did while writing a low-level
* data structure library some time ago) one detects that even
* numbers are not useable at all. The remaining 128 odd numbers
* (except for the number 1) work more or less all equally well.
* They all distribute in an acceptable way and this way fill a hash
* table with an average percent of approx. 86%.
*
```

```
* If one compares the chi^2 values of the variants (see
* Bob Jenkins ``Hashing Frequently Asked Questions'' at
* http://burtleburtle.net/bob/hash/hashfaq.html for a description
* of chi^2), the number 33 not even has the best value. But the
* number 33 and a few other equally good numbers like 17, 31, 63,
* 127 and 129 have nevertheless a great advantage to the remaining
* numbers in the large set of possible multipliers: their multiply
* operation can be replaced by a faster operation based on just one
* shift plus either a single addition or subtraction operation. And
* because a hash function has to both distribute good _and_ has to
* be very fast to compute, those few numbers should be preferred.
*
*   -- Ralf S. Engelschall <rse@engelschall.com>
*/
```