Spell Checker

Hash Tables

For this assignment you will need to implement a spell checker. Our spell checker works by looking up every word from a text in a word list. When a word is not found in the list it will be reported as a possible spelling error. This is the easy part, and the `spell-checker.c` file that does this can be found on the course website. The tricky part, which you have to implement, is to make these lookups fast. You will be spell checking whole books, the longest of which is 565.000 words. And because the word list you will be using contains more than 600.000 words, you will have to perform these lookups very efficiently using a hash table.

A hash table is an efficient way to implement a dictionary. It allows you to store and lookup (key, value) pairs. It is a generalized version of an array. With an array the key is always an integer index directly into the array. With a hash table you can use any kind of key, and it still allows you to lookup a word nearly as fast as if you where indexing an ordinary array.

It works by taking the key and using a function to transform the key into an integer index. The index is then used to access the array to retrieve the value associated with the key. The function that takes the key and computes the index is called a hash function. A good hash function provides a uniform distribution of the hash values and should use all the key data for the index calculation. But is should also be quick to calculate. For your spell checker you will need to write a hash function that hashes strings (words) into integers.

The same string will always hash to the same array index. But there may be different strings that will hash to the same array index. This is called a hash collision. There are multiple ways to deal with hash collisions:

Hashing with collision chaining

With collision chaining the hash table contains pointers to key/value pairs, so the actual data is stored outside the hash table. When multiple keys hash to the same index these key/value pairs are chained together in a linked list. A lookup will have to check all the keys of the linked list at that index.

Hashing with open addressing

With open addressing the data is contained in the hash table itself. The addressing into the hash table is called open because it is not solely determined by the hashed key. For an insert into the table the index is checked first. When this location is already occupied the hash table is probed with a specific probe sequence until an empty spot to insert the new key/value pair is found. For a lookup the hash table is probed until the correct key is found. If the probe sequence hits an empty spot the key/value pair is not in the table. The three commonly used probe sequences are: linear probing, quadratic probing, and double hash probing.

Spell Checker files

The tar file on the course website includes the main spell checker file `spell-checker.c`, a header file `hash.h` and simple program that shows you how to time C code `test-timing.c`.

The file `spell-checker.c` is provided to save you time and let you focus on the hash table implementations. The only thing you will have to do is to add the code to perform the timings for your experiments. You are of course free to modify it as much as you want, or even write your own from scratch. The spell checker main() function does the following:

1. Creates a new hash table.
2. Reads a word list from file, and inserts every word into the hash table.
3. Reads the text file and looks up every word in the hash table. When a word is not found in the hash table it is counted as a spelling error.
4. Print some hash table statistics, and performs cleanup.

Because we just want to check if the hash table contains a word, we are not interested in the actual value that is associated with it. That is the reason that every word stores a pointer to the same placeholder token, which in our case is the character `'a'`. For this assignment you can also choose not to store any value because we only care if the key, the word we want to spell check, is in the hash table or not.

The header file `hash.h` declares the hash table data structure and the hash functions you will need to implement:

`hash_t* hash_table_new(unsigned int size)`

Returns a pointer to a new hash table of the specified size.

`void hash_table_destroy(hash_t* table)`

Frees the memory of all the keys (the dictionary words) and the hash table itself. The value that is stored in our hash table always the same token *shared* between all entries. So this data is freed *once* in main and not in this function.

`void hash_table_insert(hash_t* table, void* key, void* value)`

Inserts a new key and value into the hash table. Does not insert the new value if the key is already present. Neither the key nor the value is copied in the insert function, so a copy of the key is made first. As mentioned above the value is shared between all words in the table.

`void* hash_table_lookup(hash_t* table, void* key)`

Looks up the key and returns the value pointer. If the key is not found `NULL` is returned.

`unsigned int hash_table_size(hash_t* table)`

Returns the size of the hash table.

`unsigned int hash_table_fill(hash_t* table)`

Returns the number of elements in the hash table.

For this assignment you will need to implement the collision chaining hashing technique *and* at least one of the following probing techniques:

- Open addressing with linear probing.
- Open addressing with quadratic probing.
- Open addressing with double hash probing.

It is up to you to decide the organisation of the different implementations. You can choose to implement each hashing technique in its own C file or you could make the probing sequence a parameter of the implementation that can be set with a function. Or even make the hashing function itself a parameter of the hash table. This will make it easier to perform the timing experiments with different parameters later on, but could also make your code more complex.

We have included a hash table implementation in `hash-glib.c` that contains wrapper functions to the glib hash table (http://developer.gnome.org/glib/stable). The included Makefile will compile this source file into a library and link it with the spell checker to create a spell-checker called `spell-checker-glib`. You can use this implementation to check if your hash table is functionally correct. You can also use it compare the execution speed. Can you explain why this comparison is not completely fair?

Although you can make changes to the header file, you need to use a single `hash.h` header file for all the different implementations. The main interface to the hash table should not depend on the underlying implementation. If you do choose to add functions to the interface header file that are implementation dependent, you can use dummy functions for the implementations that don't need that specific function.

The file `test-timing.c` shows how to time C code. It uses two different functions, `clock()` and `getrusage()` to time three different sections of code. These functions measure CPU time, not wall clock time. The `getrusage()` function also splits the CPU time into user CPU time and system CPU time, but that distinction is not needed for this assignment. Run the program and make sure you understand the timings that are produced. And as always check the manual pages of these functions.

You will need to submit a Makefile with your assignment that will generate all your different spell checker implementations automatically.

Experiments

Speed is what makes hash tables attractive data structures to store dictionaries. Hash tables can be very fast if they are implemented correctly, but if you use a badly designed hash function, a flawed probe sequence or just store to many elements, performance can be badly degraded. This means that the timing experiments are an important part of the assignment.

The tar file also contains the word list file and two text files. The word list file is called `british-english-insane`, where although it contains some crazy words, insane is just an indication of its size. The text files you will be spell checking are two books from Project Gutenberg (http://www.gutenberg.org/): "The origin of species" by Charles Darwin and "War and Peace" by Leo Tolstoy.

The ratio of the number of elements and the size of the hash table is called the load or fill factor. Because a probing hash table stores all elements in the table itself it cannot have a load factor higher than 1. The four implementations perform differently when the load factor increases, so timing experiments with different hash table sizes will be interesting. Other aspects that you could experiment with are different hash functions. When the key is a string a common choice for a hash function is:

```
unsigned int hash_func(char* key) {
    unsigned int hash = initial;

    for (; *key; ++key)
        hash = multiplier*hash + *key;
    return hash % table_size;
}
```

You can try to improve performance by trying different values of the constants `initial` and `multiplier`. Or you could try to improve performance by using a different hash function entirely.

A sample run spell checking "The origin of species" using a collision chaining hash table should look like this:

```
./spell-checker-chaining input/british-english-insane \
        input/origin-of-species-ascii.txt 932587
Selected table size: 932587
Hash table contains 611723 words
Hash table load factor 0.655942
bytes read 965836
words read 161656
typo's 439
seconds:              1.510000
```

The first argument is the dictionary, the second argument is the text file to check for spelling errors and the third argument is the size of the hash table.

Report

For this assignment the report counts for 25% of the grade. So don't create it as an afterthought. Your report should contain the following sections:

- Introduction of the assignment (formal, no copy paste).
- Discussion of your implementation.
- Explanation of the design choices you made.

- Description and presentation of the experiments you performed. Small timing sets can be presented in tables. But if you would want to, for example, show the influence of the load factor on the different hash implementations a graph would be better suited.
- Conclusions: What conclusions can be made from the timings you performed. Try to explain what you have observed. What problems did you encounter, and how did you solve them.

Not counting tables and graphs the report should be around 2 pages long.

Tips

Quadratic probing and double hash probing are the most difficult hashing techniques. There are some constraints on the hashing functions and the table size that should be met to make the probing work correctly. Implement the other hashing techniques first, and make sure you have time left to perform the timing experiments and write up your report. Spend your time wisely, it is better to have two working hash table implementations and some timings results, than four buggy ones and no timings.

Turn on the compiler optimisations for the timing experiments and mention the flags that you used in your report.

Create a small word list file and a text file that contain just a couple of words, and print the "spelling" errors to make sure every hash implementation works correctly.

Don't print the spelling errors that you find on timing runs, as the I/O will influence the timing. Just count the spelling errors that you encounter.

References

Wikipedia has some good articles on hashing: http://en.wikipedia.org/wiki/Hash_table, http://en.wikipedia.org/wiki/Hash_function. You will also find a good explanation of hashing in the excellent book: "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein (http://mitpress.mit.edu/algorithms/).