Datastructures 2014

Spell Checker

This assignment is due on: Tuesday, February 18th 23.00.

Anything that is turned in after this deadline, will not *be checked.*

assistent(s) Auke Wiggers en Tim van Rossum
email tim.vanrossum@student.uva.nl en wiggers.auke@gmail.com

author(s) José Lagerberg

University of Amsterdam

Department of Computer Science

Faculty of Science

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 `SpellChecker.java` 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 565000 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 is quick and uses all the key data for the index calculation. 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 pair data, 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. When the index in the hash table is already occupied the hash table is probed with a specific probe sequence until the correct key (or an empty spot, if it is an insert) is found. 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 spell checker file `SpellChecker.java`, the hash function interface `Compressable.java` and the file `Division.java` which contains the code for computing the hashcode of a String (which could be negative) and the compression map to the length of the hash table.
The file `SpellChecker.java` is provided to save you time and let you focus on hash table implementations. In this implementation the HashTable from the Java library is used. You have to add your own hash table implementations on which you have to perform experiments.

The use of the following hash function interface;

```
Compressable function = new Division(hash_size);
```

allows you to pass the Compressable `function` as a parameter of the hash table constructor:

```
HashTableOpen table = new HashTableOpen(hash_size, function);
```

The SpellChecker main() method 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.

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 in the hash table is stored with the same value, which in our case is the String `"a"`.
The implementation of the hash table is only allowed to use primitive data types. This means that you can not use lists, vectors, etc. However you are of course allowed to use an array.
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 java file or you could make the probing sequence a parameter of the implementation that can be set with a function. The hashing function should be 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.

Benchmarks

To determine how good your hash table implementation is your are asked to perform some benchmarks. Before performing benchmarks you should determine what and how much data you are going to use. Secondly you will have to determine what part of the code you want to benchmark. Lastly, it is essential that you benchmark is reproducible by the reader of your report.
Some specific challenges lay in the benchmarking of Java programs. First, the Java virtual machine performs under the hood optimizations which may cause the same code to be faster the second time it is executed. This behaviour can be switched off with `-Xint`. Of course, this is not a problem if you do only one benchmark per program execution. Another Java specific feature of benchmarking is that the Java virtual machine exists in two flavors: the client `-client` and server `-server` mode. Try it out!

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:

```
public class Division implements Compressable {
    int table_length;
    int initial = 11;
    int multiplier = 31;

    Division(int length) {
        table_length = length;
    }
    int calcIndex(String key) {
        int index;

        index = Math.abs(hashCode(key)) % table_length;
        return index;
    }

    int hashCode(String key) {
        int h = initial;
        char[] val = key.toCharArray();
        int len = key.length();

        for (int i = 0; i < len; i++) {
            h = multiplier * h + val[i];
        }
        return h;
    }
}
```

You can try to improve performance by trying different values of `initial` and `multiplier`. Or you could try to improve performance by using a different hash function and a different compression function like MAD.

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

```
java SpellChaining british-english-insane origin-of-species-ascii.txt 932587
Selected table size: 932587
Hash table contains 611723 words
Hash table load factor 0.655942
Text contains 161656 words
typo's 439
Search in:              165 ms
```

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.
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, that much 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/).