Assignment 2
Spell Checker

M. Pfundstein (10452397) and T.M. Meijers (10647023)

February 16, 2015

1 Introduction

This assignment requires a spell checker to be built. The main focus of the assignment is, however, to implement a Hash Table and utilize different techniques for collision resolution. For a spell checker to work reasonably fast, we need a way to look up words very fast, as we can have dictionaries containing hundreds of thousands of words, and a hash table satisfies this requirement.

For this implementation one hash function was already implemented (*Division.java*), this function returns an index for the table based on the key (in the case of a spell checker, the key is a word). The problem with this is, that with an imperfect hashing function, different keys can generate the same index, which is where collision resolution comes into play.

Collision resolution techniques can be divided into a few classes of methods which are: chaining, open addressing and some hybrid approaches of the former two. For this assignment the focus lay on implementation of the first two techniques, especially linear probing, quadratic probing, double hashing and collision chaining.

2 Implementation

The goal was to implement all four mentioned collision resolution strategies in an object oriented programming style. While the implementation of the data structures will be discussed in the next subsection, we will first discuss the implementation of the used techniques which might require explanation.

For the chaining collision we built our own implementation of a linked list, which remembers the head node for accessing and the tail node for quick concatenation. For the implementation of quadratic probing for every iteration of finding the next index we add to the index $c_1 * n$ and $c_2 * n^2$, where c_1 and c_2 are user defined constants and n is the number of iterations. To prevent not finding a new index, the resize factor used should always be under 50%, which is also the case for double hashing, for all other strategies they are user defined (as long as it is between 0 and 1).[1] The second hashing function used is based on D.J. Bernstein's *"times 33"* algorithm (also called djb2), using this hashing function gives us the constraints that the maximum load factor cannot be higher than 0.5 and the minimum initial hash table length should be above 64.[2]

2.1 Design Choices

Note: For completeness, the graphical layout of the data structures used and its description can be found in Appendix A.

Our implementation of hash tables is not the most efficient one. This is due to the same class using different strategies with different methods, which caused the need for conditional statements which slow down the implementation. However, as the assignment was to time and compare these different implementations, they all (except chaining collision, which works very differently) suffer from this implementation and thus can still be effectively compared. Another choice made is that of separating the class for chaining collision, this was done because it needs a very different implementation (linked lists) and so doesn't benefit from having a Strategy object.

We also overloaded the *OpenHashtable* constructor, where one now allows the user to specify at which load factor the table needs to be increased with which factor (respectively *ResizeThreshold* and *ResizeFactor*), with some constraints for specific strategies as discussed before. Finally, the choice was made to store the actual words and not the placeholders, so we can realistically look up words and time the performance of this as well.

3 Experiments

Setup

In order to test the hash-table classes and the four strategies, we developed a small experiment script that can run the code, record its output and plot the results.

The SpellChecker program was configured in such a way, so that it can output the results for each hash-table in csv format. The script enumerates the sequence $\{2^8, 2^9, ..., 2^{26}\}$ and uses the enumeration value as hash-size input for the SpellChecker program which was checks for spelling errors in the novel War and Peace. It then runs for each value the script 10 times and stores the results into a directory called output. From there a python script uses the contents of the directory to create several plots from the average values of the 10 runs in a directory called figs. We recorded the following measurements: A) The build-time, the time it takes to store all words from War and Peace in the hash-table, b) The run-time, the time it takes to retrieve all words again, and C) the load-factor, the ratio of hash-table size and count of each hash-table implementation. The big advantage of that setup was that we could easily rerun the experiments when someone made a change in the program. If the reader wants to run the experiment for herself, then she can execute the shell script run-experiments.sh.

Results

When looking at the results of the hash-size vs. build-time performance (See Fig. 1) than we easily see that the CollisionChaining strategy is the most effective way when using a hash size smaller than 2^{20}. The DoubleHashProbing strategy doesn't perform very well in this case but performs very good when the hash size is bigger than 2^{20}. QuadraticProbing is stable all way through and LinearProbing also doesn't perform very bad. The peak at 2^{21} is very interesting and future research could address this issue. The same goes for the zigzag-pattern of CollisionChaining between 2^{22} and 2^{24}. That CollisionChaining performs better when a small hash-size is used is not surprising, as the Probing strategies must resize when no more space available and this is of order $O(n)$.

The hash-size vs. run-time performance (See Fig. 1) shows clearly that Collision Chaining needs a large hash-size in order to perform well. Hence if this strategy is used, the user must make the tradeoff between fast build-time and fast run-time. According to our tests, the best hash-size for this strategy is between 2^{19} and 2^{20}. The best performers here are QuadraticProbing and DoubleHashProbing. Both perform tremendously good independent from the initial hash-size. This is because they resize itself during insertion. Access is therefore

always of order $O(1)$ but it also explains why CollisionChaining performs better at build-time. This is in strong contrast to the CollisionChaining strategy, where a high load factor can lead to a worst-case retrieval complexity of $O(n)$. This can also be seen in figure 2, which shows that the load factor is enourmous in the beginning. For instance a load factor of 2000 means that per hash, 2000 items a stored.

4 Conclusion

Our conclusion is that the QuadraticProbing strategy seems to perform best. Its run-time *and* build-time performance is very stable and independent of the hash-size. CollisionChaining should be used if a lot of fast insertions are needed and if enough memory is available. But also LinearProbing and DoubleHashProbing are worth considering. We want to note that there are much parameters for each strategy to tune and that we were not able to do experiments on this. Please see figure 3 for the total running time of each hash-table (run-time + build-time).

References

[1] Steve Wolfman. When Keys Collide. `http://courses.cs.washington.edu/courses/cse326/00wi/handouts/lecture16/sld001.htm`, 2000. Accessed: 15-February-2015.

[2] Ozan Yigit. Hash Functions. `http://www.cse.yorku.ca/~oz/hash.html/`. Accessed: 15-February-2015.

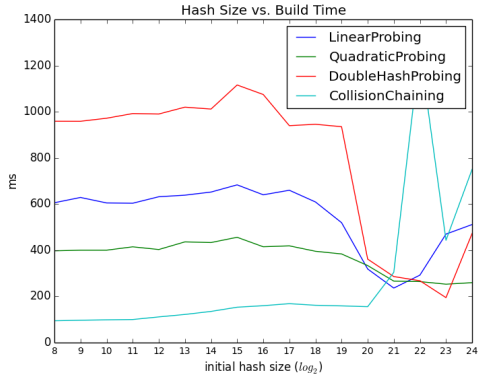
Appendices

A Data structure

Note: Refer to figure 4 for the visual layout of the data structure.

The implementation of hash tables, collision resolution strategies and hashing techniques uses two abstract classes (*AbstractHashtable*, *Strategy*) for the former two and one interface (*Hasher*) for the latter.

The two different implementation of hash tables, namely *OpenHashtable.va* and *ChainHashtable*, inherit from *AbstractHashtable*. *OpenHashtable* uses open addressing as it's collision resolution technique. These techniques are: *LinearProbing*, *QuadraticProbing* and *DoubleHashing*, these classes all inherit from the abstract class *Strategy*. *ChainHashtable* uses collision chaining, and the strategy is implemented by a linked list implementation (*LinkedHashList* which uses *HashNode*). Finally, all implementations require a hashing technique: *DivisionHasher* while the hash table implementation which utilizes double hashing uses *DJB2Hasher* as it's seconds hash function.



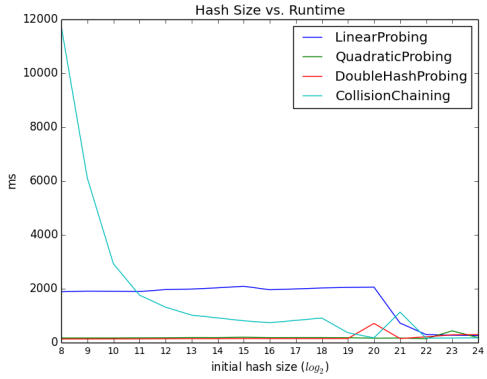
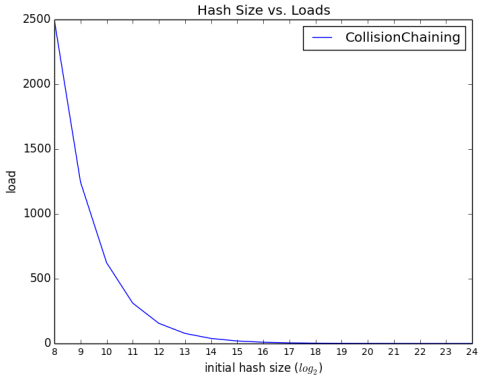


Figure 1: Performance measures with respect to initial hash-size



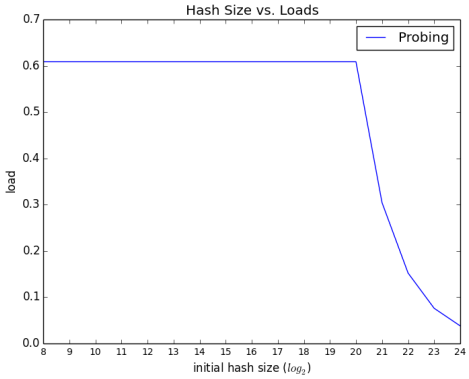


Figure 2: Load factor with respect to initial hash-size

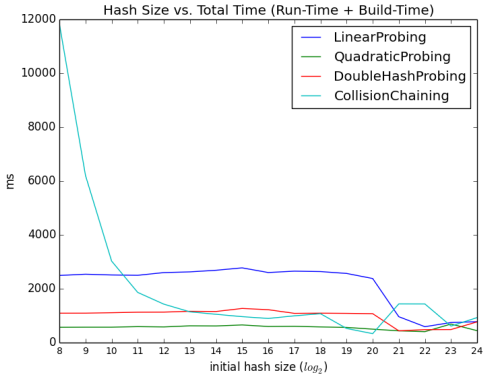


Figure 3: Total running time with respect to initial hash-size.

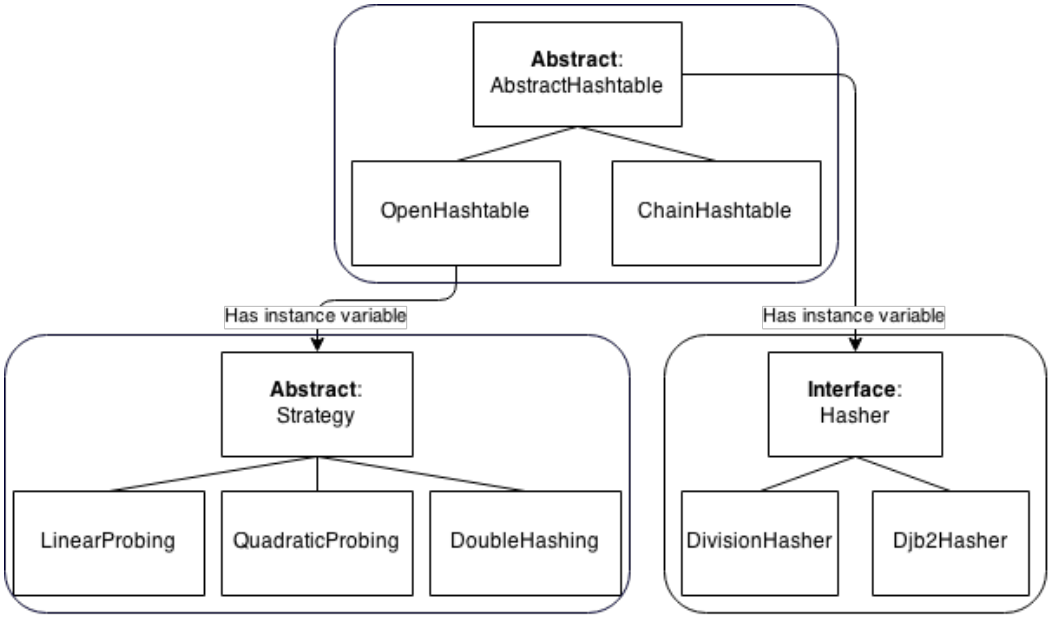


Figure 4: Graphical representation of the data structure