Spell Checker
Datastructuren week 2

Gijs van Horn & Jeroen Vranken
10070370 & 10658491

February 16, 2015

Introduction

The goal of the assignment is to create a spelling checker. The spelling checker uses two text files: one text file to check for spelling errors, and another text file to use as a dictionary. A spelling error is counted whenever a word in the source text is not found in the dictionary. The main focus of the assignment is to create a data structure called a hash table. A hash table uses a key to store values. The key is used to look up the values, which results in fast look up times. The key is generated based on the value to be stored, and key generation is done in many different ways. There are different hash table types, in this assignment the "Collision Chaining" and "Open Addressing" methods were implemented.

Collision Chaining

Collision chaining is a method where for each value a key is generated. The value is stored a the generated key index. Each key index is the start of a chain. Whenever an index is encountered where a value is already added, the current value is added to the end of the chain. This allows for more words than stored than possible key indexes.

Open Addressing

The basic idea of open addressing is simple. Create a key for a given value and put it in the table. When another value is encountered, simple put the value in the next available spot. This works well until the table starts to become full, when performance drops drastically. The table has to be increased in size when around 75% full. The table is doubled in size and new key indices are created for the existing values in the table. The values are then copied to the new table.

Implementation

The two different hash table implementations consist of four classes, the Classes `CollisionChainTable`, `LinearProbeTable`, `Entry`, and `Chain`. The Entry class was created for easy access to keys and values within a table and the Chain class is an implementation of a one-directional linked list. The following is an overview of the classes with their fields and methods. Moreover, we have created a GenericHashTable interface that defines the functions that each hash table must implement.

CollisionChainTable

This class is the implementation of a hash table with collision chaining, it has 4 fields; size, wordcount, table, and function. The table field is an array of Chains (our implementation of a linked list). Size is the length of the array and wordcount is the total number of words in the table. Function is a Compressable object and is the hashfunction for assigning indexes to entries (instances of the class `Entry`). The constructor for a CollisionChainTable sets the size and fills the table with empty Chains. The `put(String key, String value)` method creates an entry of the key-value pair and appends it to the list located at the index (computed by the hashfuntion) within the table. The `get(String key)` method retrieves the value associated with the key from the table, it does so by calculating the index, retrieving the Chain and comparing every element in the Chain with the the key to return the associated value. The last two methods are getHashSize() and getWordCount(), they return the relevant fields so this information is accessible from the main method for printing and calculating the load factor.

LinearProbeTable

This class is the implementation of a hash table with open adressing that uses the method linear probing for insertion of new elements. Its fields are size, wordcount, threshold, function and table. The size and wordcount do the same thing as for the CollisionChainTable function. The threshold is a double between 0 and 1, its function will be described later. The function field is a Compressable object which is the hash function and lastly the table field is an array of Entries. The constructor initializes all the fields with the given parameters. The `put(String key, String value)` method first checks whether the load factor of the table is higher than the threshold, in the case it is the `resize()` method is called. It then calculates the hash of the key and puts the entry created from the key and value in the first empty space in the table at or after that index. The `get(String key)` method calculates the hash of the given key and finds it at that index or after unless it finds an empty array element (null value) first. The `resize()` function makes a deep copy of the current array, replaces the old array with an array twice the size and fills this new array from the old one.

Experiments and results

Benchmarks were performed on collision chaining and linear probing. The used machine was an `HP EliteBook 8530w`, using `Java 1.7` with the `-Xint` flag.

Collision Chaining

The initial test hash size was specified using the command line. The load factor was calculated by dividing the number of words with the hash size. For each subsequent test the hash size was doubled. The results of this benchmark can be seen in Figure 1. The log base 2 was taken of the x-axis to make the distance between measure point evenly distributed.

Linear probing

The benchmarks for the linear probing hash table have been done on increasing hash sizes, starting with a size of 25.000 and increasing after each test with another 25.000 for 20 tests. The results can be found in Figure 2. The load factors in this figure are a little deceptive, they are the load factors calculated after all re-sizes of the tables have been performed. This causes them all to be below a threshold of .75 because we set this as the maximum load size before a re-size occurs. Results about the effect of the number of re-sizes have been omitted because no relation to timings was found.

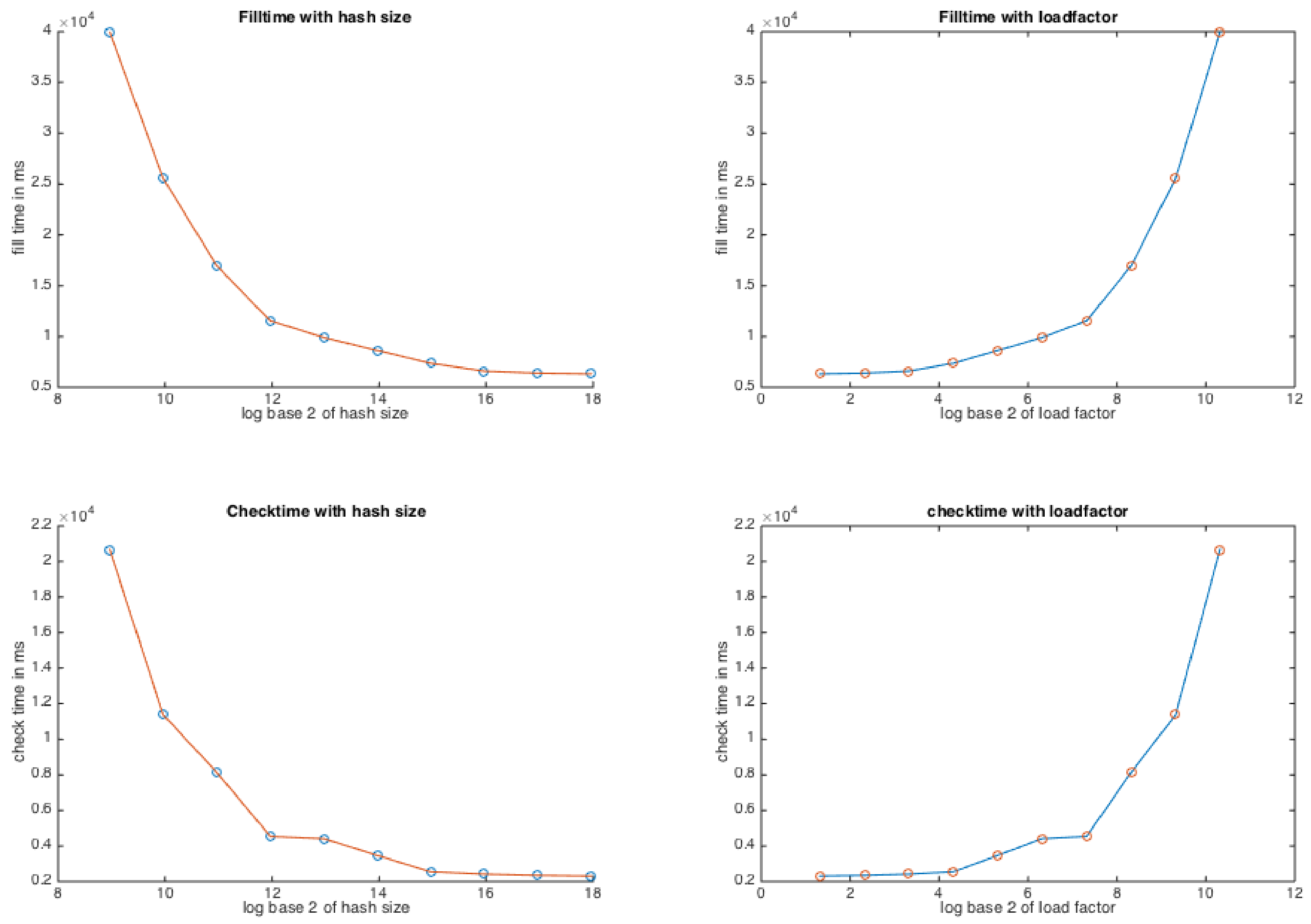

Figure 1: Collision chaining benchmarks

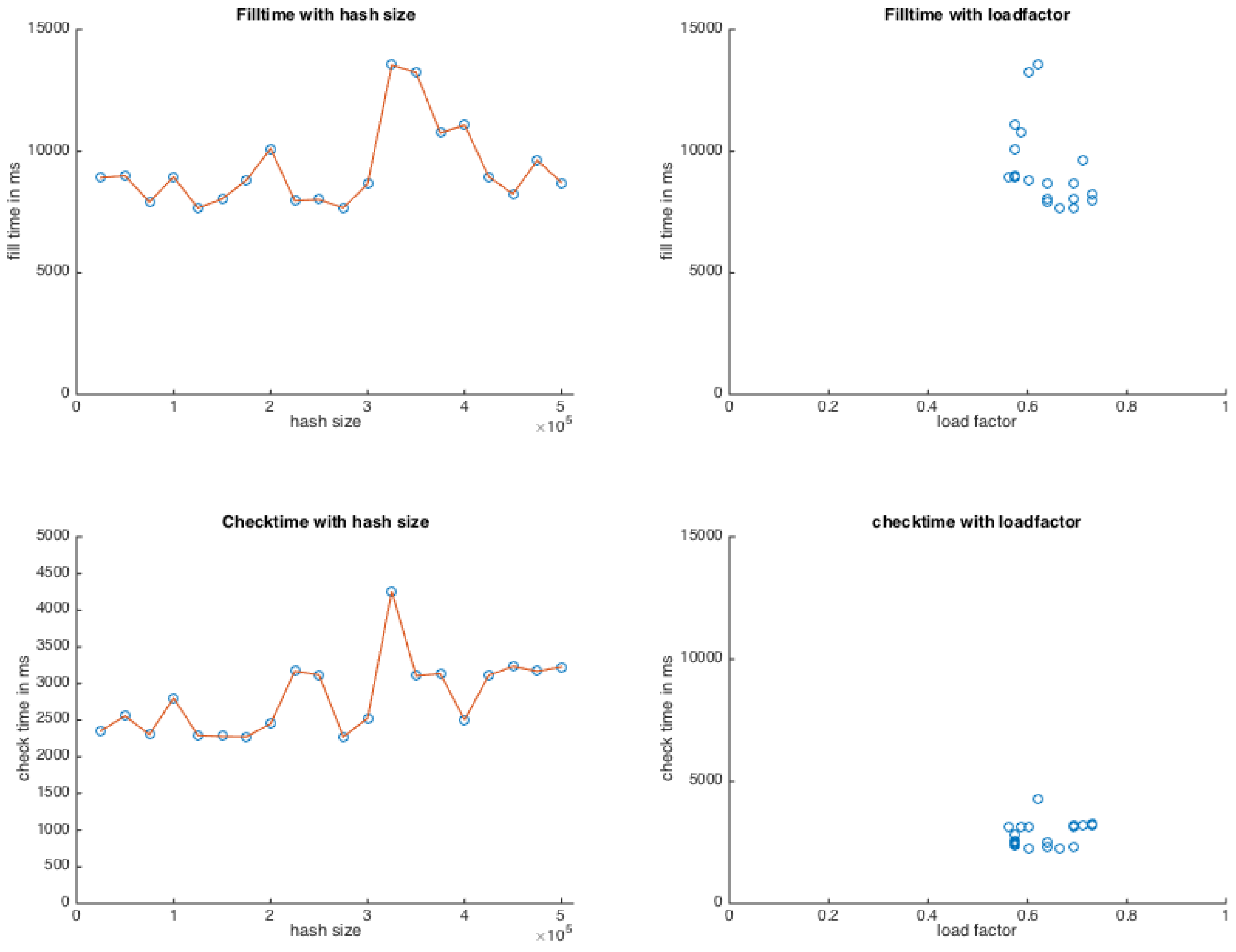

Figure 2: Linear probing with threshold at 0.75

Conclusion

The hash size and the fill time are inversely proportional to each other in collision chaining. Smaller hash sizes result in longer fill times. This is because small hash sizes need long chains to store the values. Searching in these chains takes long because a comparison needs to be made on every node.

In open addressing it is plain that the initial hash size has no impact on fill or check times, any variation is likely caused by other computer activity.

Comparing the two methods on fill time, collision chaining is significantly faster, unless the hash size is very small compared to the number of words to be stored. Linear probing stays constant.
Linear probing is faster initially on check time, and stays constant. However, on smaller load factors collision chaining will become faster.

In conclusion, if you know in advance how large you table will be, collision chaining is the faster option. If you expect your table to grow a lot, then linear probing is the better option because is maintains a constant fill and check time.