UvA / Datastructuren / 2015-2016
Plenair assignment

Yorick de Boer [10786015]

February 2016

1 Introduction

The subject of this assignment was to measure the performance of different data structures in Java. To make this possible I had to implement my own versions of an Open Addressing hashtable[3], Collision Chaining[2] hashtable and a Trie data structure. The data structures will only be tested on their recollection performance. This is done by matching a sample list of words with another larger (dictionary) list. The performance is measured by the elapsed time needed to check the whole list. All this had to be done without any imports other than file readers and file writers.

The main question of this project is: What is the performance difference among data structures when recollecting random words?
Subquestions are:

- What is the effect of several hashing functions in a hashtable structure?
- How does the size of a data structure effect the datastructure?

2 Implementation

The goal of this project was to implement all the requested datastructures. Since the programming language is Java a good object orientated programming style is required. Therefore all datastructures are in their own class and implement my own interface MyDatastructure. The whole projects design is based on the command pattern[1].

Java does not have a built-in tool to measure real CPU time so I had to use the system time. To measure the time I used $System.nanoTime()$. The function $System.currentTimeMillis()$ was not an option because the significance of the hashtable timer would result in only 1 digit.

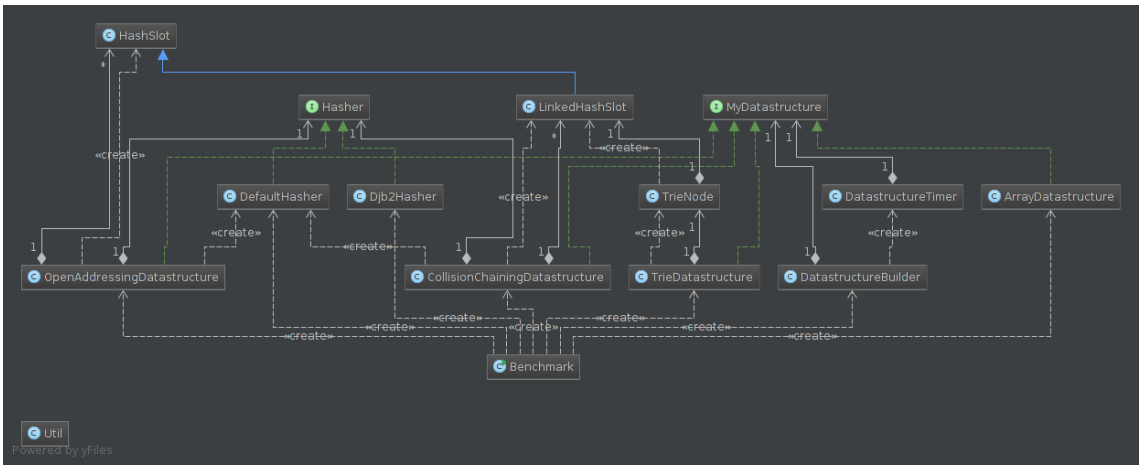

Figure 1: UML diagram of the whole project

2.1 Array data structure

The most simple type of data structure was an array data structure. Indeed as the name says, an array. The only thing I had to create was a wrapper for a string array which would implement MyDatastructure interface. It has the ability to dynamically resize so setting the exact size for the array is not necessary.

2.2 Open Addressing and Collision chaining data structure

A hashtable works by using a hash as the indexer for an array. I had to implement two types of hashing tables, an open addressing table and a collision chaining table. The difference between these two is the way it handles two different keys producing the same hash. Open addressing works by adding some number to the hash and see whether this produces a not yet occupied slot. The other approach is a collision chaining table. If a collision occurs it chains the keys with the same hash in the same slot.

There are multiple ways to optimize a hashing table. In the case of an open addressing table, there are multiple ways of probing for a new slot. For my implementation, I limited the probing to the most simple solution, that is linearly probing for the next open slot.

An other way to optimize a hashing table is the hashing function. The default Java hashing function can be used. But I also tried another function called the djb2-hashing[4] function.

Also relevant to the performance of a hashing table is the amount of free slots in the table. Intuitively the more slots are free, the faster a free slot is found. This is also available as an easy to change option.

2.3 Trie datastructure

The trie datastructure works by creating a node for every character of a word and connecting these nodes to construct the word. When putting a new word in a trie it splits the word into its characters. It then checks each character of the word if it is inside the trie. When it is not in the trie it creates a new node from the last character it does contain, so it is linked to the next character.

To test if a word is contained in the trie it follows all the character nodes until the word is formed. A side effect of this is that when a larger word is saved, and a shorter word is queried it returns true, the word is inside the trie. While the shorter word is not specifically added to the datastructure. For example when the the word *Strawberry* is saved and the word *Straw* is queried it returns true. To fix this, the whole word is saved in the last node from a word and checked when looking up the word from the datastructure.

3 Experiments

3.1 Setup

In order to test the performance of the different datastructures several tests were setup. All tests were run over all 30 sample files. To limit environmental factors the average words per nanosecond was then calculated over all these files. All tests were run on a HP Elitebook 8570w with only the experiment task running. The java version used was Java 8.

The hashing table tests were run multiple times with different parameters. For the table size of the open addressing table a range from the dictionary size to 2.5 the size of this dictionary was chosen. For the collision chaining I chose a range from 1 to 2.5 times the dictionary size. The other parameter is the type of hashing function. All measured data was written to a csv (comma separated values) file.

The analysis of the csv was done in Microsoft Excel, two plots were made from the data.

3.2 Results

When looking at the results it is clearly visible that all data structure outperform the simple array structure.

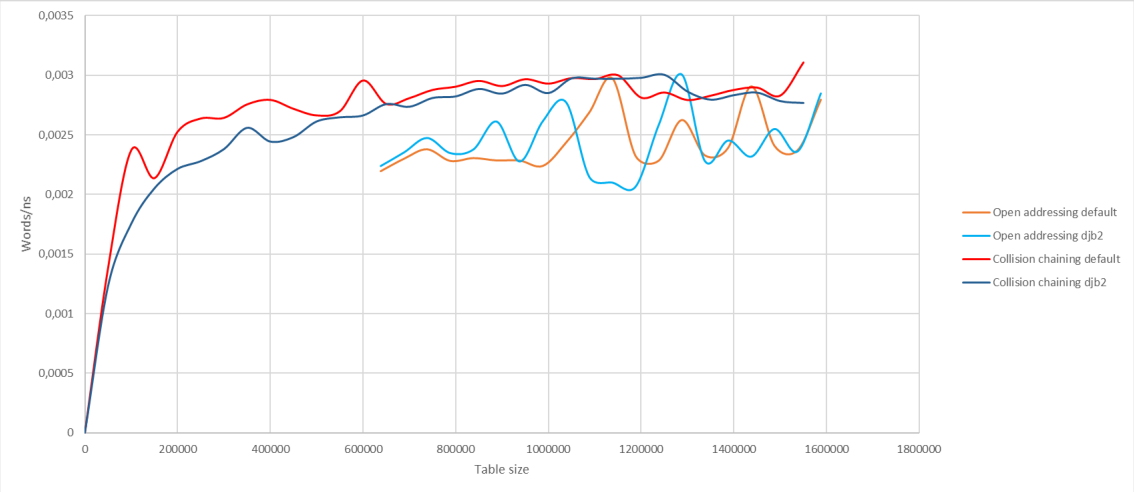

Figure 2: Table size vs words/ns

Collision chaining shows a logarithmic function for the table size vs words/ns. The Open addressing table seems to have a less clear function. It does however, show a slight linear trend.

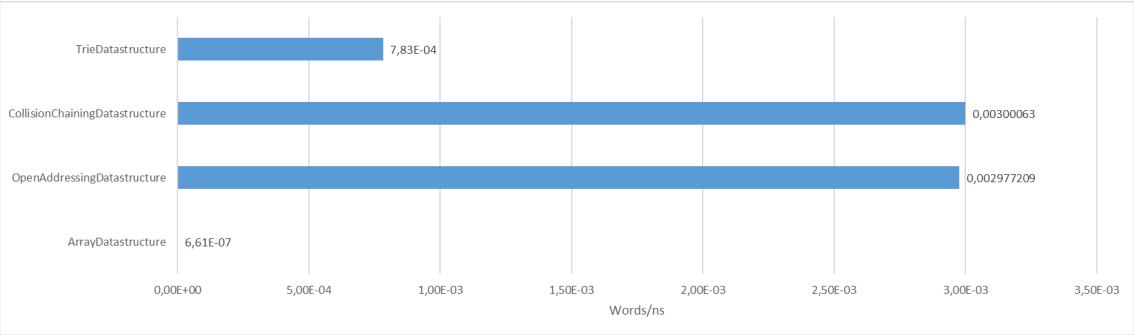

Figure 3: Quickest table size comparison

Figure 3 shows the performance of the different data structures compared to each other. For the hashtable measurements a hashtable size of approximately 1148400 possible entries was used. This is 1.8 times the dictionary size.

4 Discussion and Conclusion

The performance differences among the data structures can be explained by looking at the Big-O representation. All data structures have a worst case time complexity of O(n). But a large enough hashing table with a almost perfect hashing function would a have a high probability of complexity O(1). It is clear that the array data structure performs the worst and the hashtables definitely perform the fastest when looking up words. The relatively bad performance of the trie data structure can be explained by the way this structure is implemented. It is all done in arrays which means that for every character in a word a whole array has to be read. It would be better to use a hashtable for every n'th character. Then it might perform even better than a regular hash table.

To extend this research different types of probing for an open slot for the open addressing data structure could be examined. Alternatives are quadratic probing and double hashing. Another

point is the hashing function. Another point of improvement could be exploring different types of hashing function which improves the uniqueness of the hashing.

References

[1] Command pattern
http://www.tutorialspoint.com/design_pattern/command_pattern.htm

[2] Collision resolution
https://en.wikipedia.org/wiki/Hash_table#Collision_resolution

[3] Open Addressing
https://en.wikipedia.org/wiki/Open_addressing

[4] Hash Functions
http://www.cse.yorku.ca/ oz/hash.html