Datastructuren Plenary Assignments

Albert John Mwanjesa 10797874, Simon Verboom 10794468 - 19/02/16

Introduction

This assignment had as aim for us to consider the performance of our algorithms not only in correctness, but, very importantly, in speed. Different data structures have different complexities. For a specific, we as programmer, should take into consideration all of these complexities.

The specific case in this project was reading a text file and storing all words that are present in the file. After that, our program had to look up words from 30 others file in our set of words from the original file. The tasks were to be performed using an Array, Hash Tables and a Trie. The hash tables assignments consisted of two different types: a hash table with open addressing and a hash table that uses collision chaining. As to be seen in the .java-files, we were able to successfully store the data using the right method and look up words from the sample files, except the collision chaining algorithm. Unfortunately Simon wasn't able to properly finish this assignment before the deadline. In the table and graph on the next pages, those results are therefor left out.

Method

About our code, a few things should be mentioned in advance. Firstly, in our hash table with open addressing, it might be notable that we use prime numbers in our hash function: this resulted in a higher chance on unique indices. Also noticeable, in our trie file, it turns out a 'trie' in our case is actually just a node with 26 nodes as children. This means you start with 'root' at our object 'myTrie'. In the object, every time a word is being read, a node is created in one of the children, and so on. The code for our Array-store-algorithm seems pretty logical to us, so there seems no need for any more explanation. For more information en explanations, check the comments in our files.

As mentioned before, we couldn't manage to complete the second hash table assignment on time. There are a few reasons for this. The idea of chaining when collision occurred, was clear. It was also clear that we could use the same hash function as the previous assignment, and also pretty much the same reading-code in the DS.java file. Maybe except for a while loop to check every word in some linked lists in the hash table. We weren't able to successfully write error-free code for a linked list when collision would occur. The method to successfully implement this in a working hash table was too unclear and the time we had left was too little. Despite this, we had an idea how to fit this in the rest of the structure, partly the same structure as the other hash table assignment.

Results

This table below shows the results we got. On the next page, the corresponding graph is shown with the size of the file on y-axis and the file on the x-axis.

Striking is sample_24rQ[WUNIhd.txt, the amount of strange chars were surprisingly difficult for the hash table checkup function. The reason for this is unclear.

Array	Open Addressing	Trie	File	File Size
17260507993	681374685	22459533	sample_0OXg@T=T55.txt	103
16970156166	82801083	7421080	sample_18ywQoP^]I.txt	103
17450456499	907733652	7298770	sample_Do8D@6^MvZ.txt	103
18271238259	218830119	4673403	sample_uk4]KY7thf.txt	103
19196508719	66399269	5427357	sample_YMJh]w2k5P.txt	103
20314308147	901769162	5851162	sample__in]Ot6R79.txt	103
17179692081	84477486	5660428	sample_3B2_2GC8[M.txt	104
16363875384	63069983	6427211	sample_4s^U6a1^06.txt	104
18507870758	1232053180	3975899	sample_OtVTk]9Fe4.txt	104
19888023080	472171486	4601984	sample_]29pgao6lk.txt	104
35528629938	5771516448	12535821	sample_10wsogiLdW=.txt	206
35305973171	6798805905	19517701	sample_15ONfIJ1D=g.txt	206
36404231380	4921831265	10125563	sample_9TS_Fs4f17j.txt	206
34948418942	2600752072	8093352	sample_EB1^j@13=8c.txt	206
44678530359	5935229237	18032885	sample_H19_Uo81DhY.txt	206
37384679467	3225459610	10693488	sample_RoxM;UvI14@.txt	206
35605739800	3909174846	10073390	sample_B6JH16JohH9.txt	207
35735960013	3015772734	105917491	sample_L[11U3fYc_E.txt	207
37820945979	5546504164	8981589	sample_ocBe18XVd0x.txt	207
38142150048	3101066184	8036046	sample_o]_4D=12UBt.txt	207
55138590683	10395454637	23682623	sample_8Lg2uNPd22[.txt	308
58134297465	14799771547	14996970	sample_H2d^Fw29mjq.txt	308
54516527757	24361840605	23464519	sample_24rQ[WUNIhd.txt	309
59598663410	12756563099	14175447	sample_OONncoB728I.txt	309
57573661903	8280906661	14526123	sample_SC`S212i2Od.txt	309
54813231101	13559003207	18986554	sample_26Jdo5Z3d_0.txt	310
60087290166	9536664701	14805381	sample_SZj^s2L20a1.txt	310
66448245956	12169034516	19834592	sample_Xef3p25N9oS.txt	310
61831331129	9979904539	18903161	sample_[r23G7WAcjt.txt	310
60335932900	12596893350	20594533	sample_O4MBZ27gLgA.txt	311

Figure 1: Result table of the evaluation per file

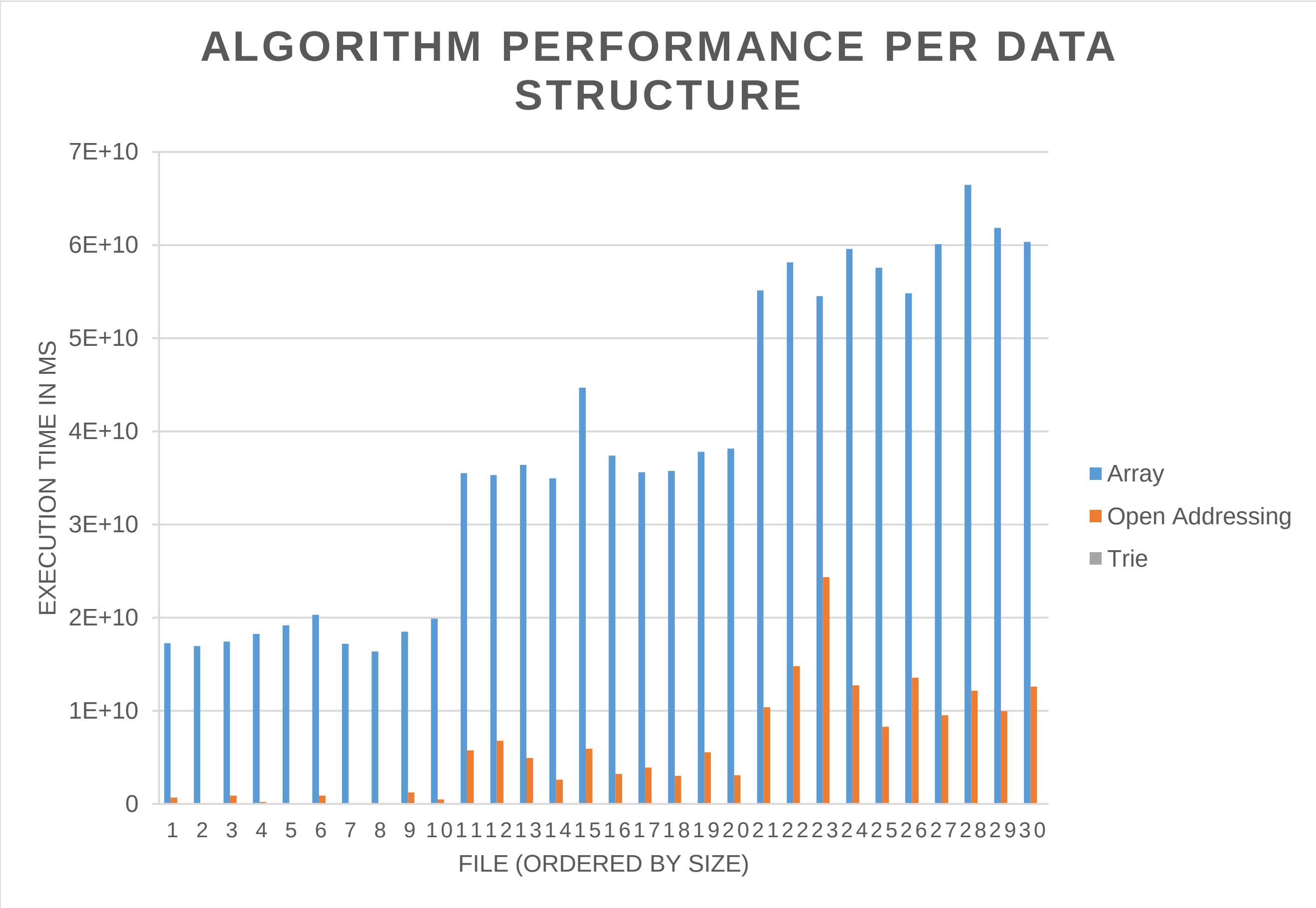

Figure 2: Graph as a the result of the evaluation

Otherwise the graph shows a unsurprising for the arrays, that the bigger the file, the longer is take to execute all the lookups in the data structures.

Conclusion

We can conclude from this project, that for insertion and lookup task, arrays are perform the least well of the three data structures. Tries are the fastest in this bunch followed by Hash Tables with open addressing. We cannot say that Tries are the fastest of all assign data structures, because we were not able to test the Hash Table with collision chaining.

We see in the results that Tries are very inconsistent in lookup time. Let's reason about this, between sample_L[11U3fYc_E.txt and sample_ocBe18XVd0x.txt, there is significant difference in lookup times even though both files are roughly the same size. The only reason, we could come up with, would be if sample_L[11U3fYc_E.txt contained on average more words that are longer than sample_ocBe18XVd0x.txt making the lookup time longer for the former.