CS231: Project 7

Trees v. Hashing

Parth Parth | CS231L-A | Dr. Alan Harper

Abstract

In the project, a Hash Table has been created. A Hash Table is a data structure that uses a hash function (which is a formula) that computes an index position called hash code to store data into an array of buckets. This allows for a time complexity to access that is O(1) in the ideal case. In my implementation, if two elements get the same hash code, they are linked together using a Linked List. Once a certain number of elements are reached, the array is expanded and all elements are add to it again to keep time complexity close to O(1).

The purpose of doing it in this project is to compare times for reading the same files between a Binary Search Tree Map and a Hash Table (Hash Map).

Analysis

The following are the times that were obtained by analyzing all the reddit comment files. The ones highlighted in yellow were dropped from the robust average calculation.

	2008	2009	2010	2011	2012	2013	2014	2015
1	2270.19	5476.40	7963.05	6847.66	5953.64	5758.27	6086.68	12364.39
2	2204.96	5672.04	7406.91	6541.13	5952.92	5924.40	6845.69	13260.27
3	2252.93	5648.62	6872.01	7675.01	5768.88	5622.57	6007.36	13074.57
4	2111.92	5661.21	6755.47	6252.35	6439.31	6189.60	6427.00	12919.31
5	2188.05	5714.87	6907.88	6792.63	6197.47	5685.50	5927.56	12663.41
Robust average	2215.31	5660.62	7062.26	6727.14	6034.67	5789.39	6173.68	12885.76

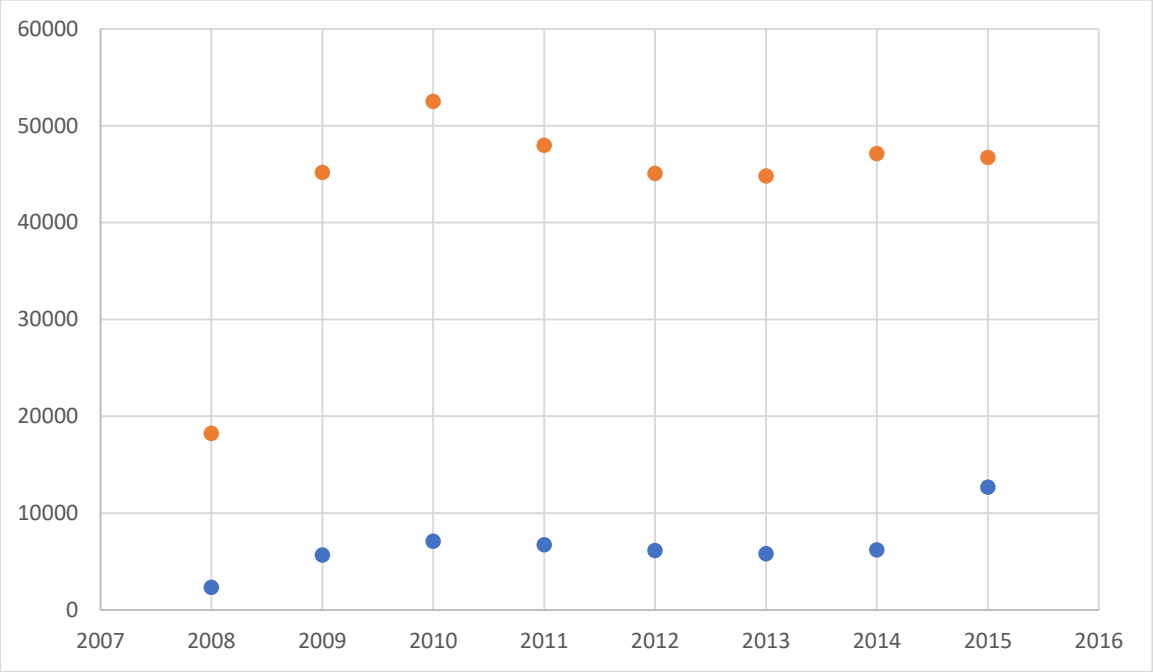
The following are the results obtained from running the analysis on the Binary Search Tree Map from last week:

	Run-Time	Total Words	Unique Words	Depth of Tree
2008	18240	16492769	191901	312
2009	45176	40263318	368559	500
2010	52505	46955203	428006	239
2011	47974	43372614	424995	303
2012	45075	40213302	431132	435
2013	44812	39236476	454010	288
2014	47132	41543298	491737	175
2015	46730	41151652	551608	156

On redoing the HashMap analysis, the following data was obtained:

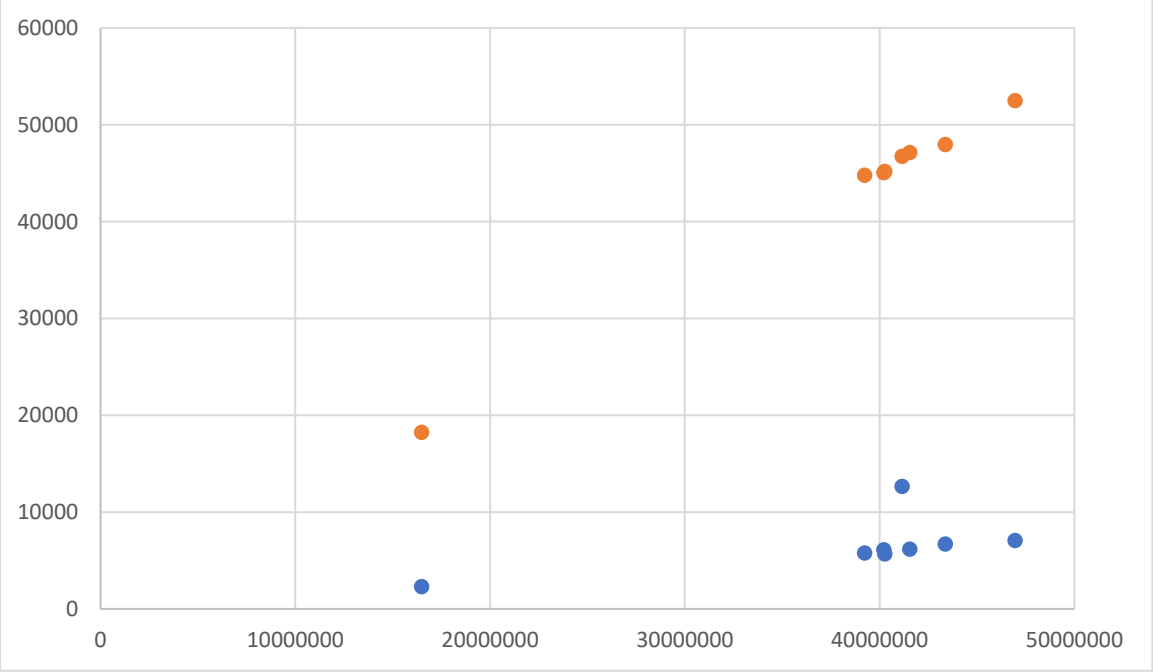
	Binary Search Tree						Hash Map	
	Run-Time	Total Words	Unique Words	Depth of Tree	File Size of original files (MB)	File Size of analysed files (MB)	Run-Time	Collisions
2008	18240	16492769	191901	312	91	2.1	2317.2	43709
2009	45176	40263318	368559	500	222	4.2	5677.3	82633
2010	52505	46955203	428006	239	255	4.8	7089.4	67961
2011	47974	43372614	424995	303	234	4.7	6729.3	67075
2012	45075	40213302	431132	435	216	4.7	6124.4	68852
2013	44812	39236476	454010	288	211	5.0	5788.9	75294
2014	47132	41543298	491737	175	224	5.6	6189.4	85917
2015	46730	41151652	551608	156	223	6.5	12667.3	103910

The first graph (Year versus run-time) is given below:



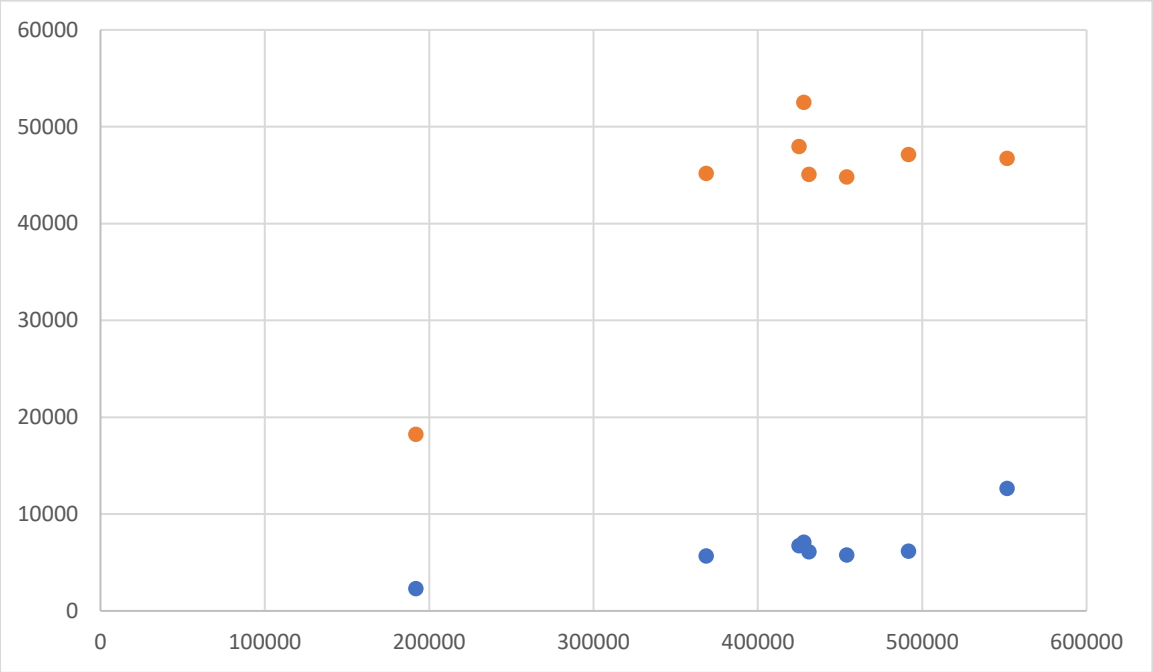
The run times for the Binary Search Tree are in orange while those for the HashMap are in blue. The run-time is on the Y axis while the years are on the X axis.

The next graph (Total number of words versus run-time) is:



Again, the run times for the Binary Search Tree are in orange while those for the HashMap are in blue. The run-time is on the Y axis while the total number of words are on the X axis.

The last graph (Unique words versus run-time) is as follows:



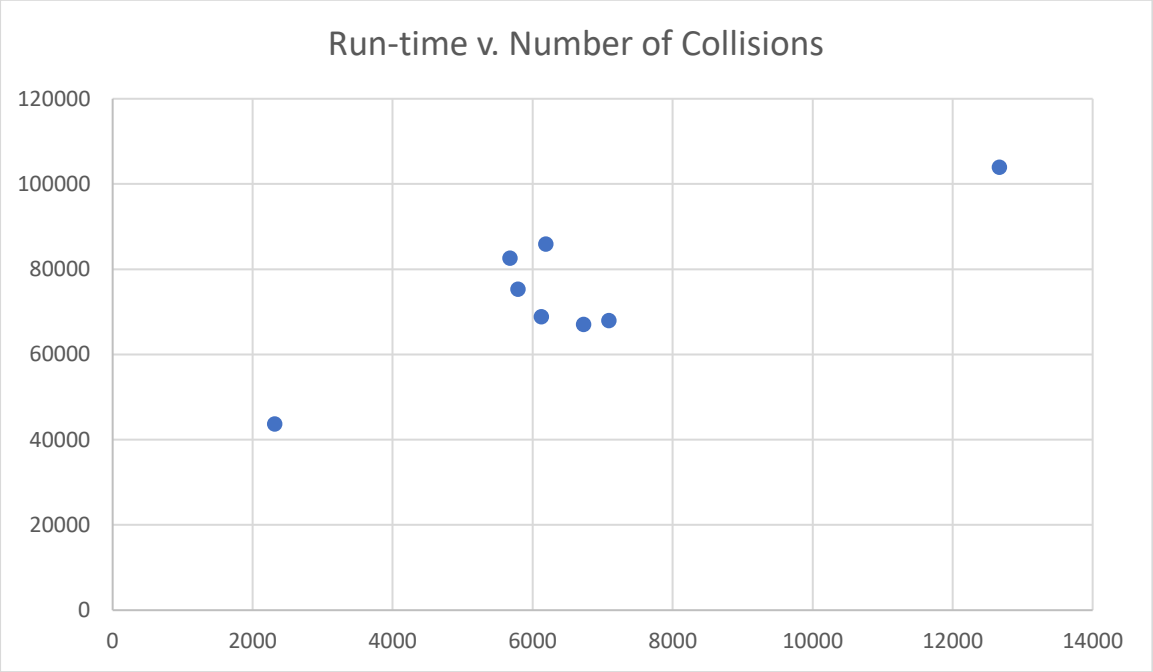
The run times for the Binary Search Tree are, again, in orange while those for the HashMap are in blue. The run-time is on the Y axis while the number of unique words are on the X axis.

Conclusions and Results

All three plots make sense. All three plots are a clear indication to the fact that the HashMap is faster. The second and third plots show that while a HashMap tracks somewhat with the trend of time taken vs total/unique words, it is much faster.

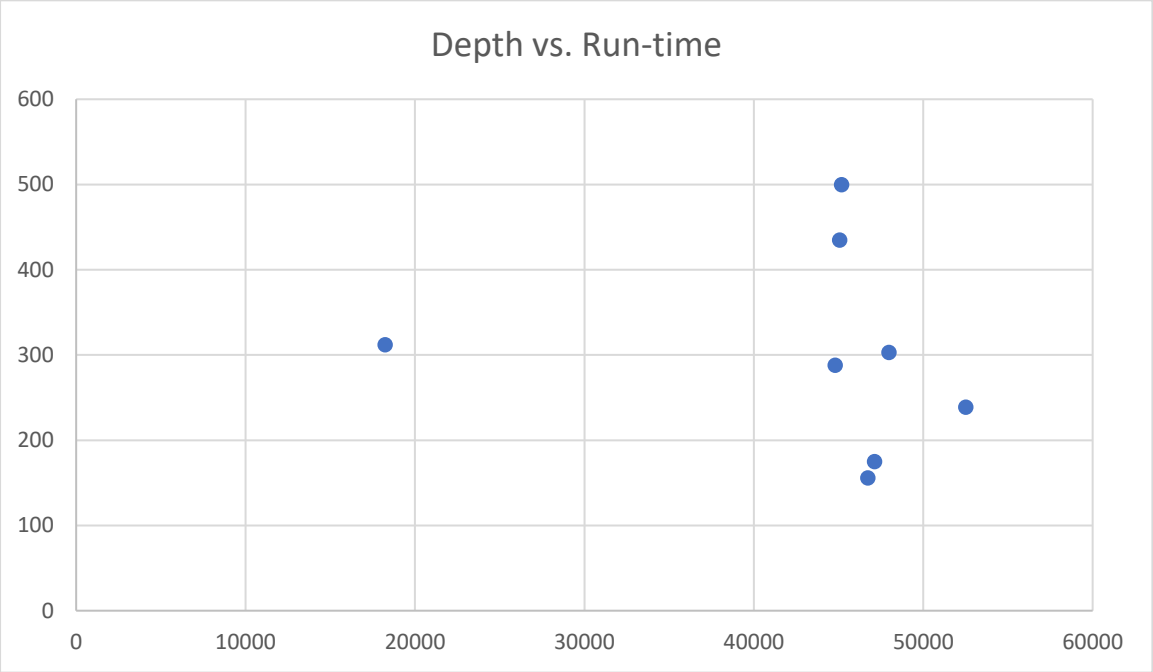
HashMap is probably faster because, as noted in the abstract, it has a time complexity of adding very close to O(1) i.e. constant time. (Clearly, from the graphs, it is not exactly O(1).) A Binary Search Tree, however, has an average time complexity of O(n*log(n)) which is much slower than the Hash Map.

The following graph shows a relationship between the run-time for the HashMap and number of collisions:



Clearly, as the number of collisions increases, the time increases too.

The next graph shows the relationship between the depth of the tree and run-time:



It is evident that while run-time does increase with depth, at some point the time becomes constant for depths.

Reflection

I learnt by doing this project that using a HashMap is faster than a Binary Search Tree Map for this specific case and pretty much all cases where the data is randomly and evenly distributed. This is due to the reasons of time complexity discussed above.

Experimentation / Extension 0

In the main project, the HashMap expands in size when its size reaches 0.75 times the max-size. In this experiment, I have changed it to be so that it expands only when max-size is reached.

The run-times for the original condition have been copied from the analysis done above:

	Old condition	New Condition
2008	2317.2	1958.5
2009	5677.3	5505.8
2010	7089.4	5861.0
2011	6729.3	5193.6
2012	6124.4	4746.5
2013	5788.9	4849.6
2014	6189.4	5118.1
2015	12667.3	5198.7

Clearly, this condition is more efficient. This is most likely because in the old condition, the time to rehash the map was also being taken into account which was increasing execution time.

Extension 1

In this extension, I have implemented custom hash code calculation paired with chaining. This is done by returning the sum of the UNICODE values for each character in the given key.

The times have been compared below:

	Original method	New method
2008	2317.2	7771.7
2009	5677.3	24245.3
2010	7089.4	28448.2
2011	6729.3	26320.7
2012	6124.4	25292.7
2013	5788.9	30726.5
2014	6189.4	37103.4
2015	12667.3	38810.5

Clearly, this implementation is slower than the original implementation.

Extension 2

For this extension, I've gone through all methods in all classes and added comments stating what is the last line where an object is used or when it is cleaned out of memory.

In general:

1. Objects that are created in constructors are usually removed from memory when the method in which the instance of their class was created.
    a. The exception to this is if the object reference is assigned to the fields or local variables of any other method. In that case, they are removed when the object of the other class(es) is/are removed. In case of a local variable, they are removed from memory when that method ends.
2. Objects referenced using local reference variables are cleaned out of the memory when the method in which the variables exist ends.

Extension 3

For this extension, I've used a Binary Search Tree instead of a Linked List to implement chaining. The times obtained are as follows:

	Linked List	Binary Search Tree
2008	1958.5	2874.0
2009	5505.8	8349.7
2010	5861.0	8202.6
2011	5193.6	7392.6
2012	4746.5	6970.8
2013	4849.6	6758.4
2014	5118.1	7458.6
2015	5198.7	7371.2

Clearly, the Binary Search Tree is slower than using chaining via a Linked List.

References / Acknowledgements

I did not make use of any reference material to complete this project.