Alex Crooks

Analyzing Data Structures

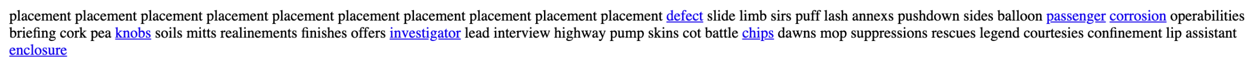
Goal

To determine how effective which data structure (a binary search tree, a splay tree, an AVL tree, and a hash table) is used when employed as a search engine.

Overview of driver code

We have a list of data structures (the four mentioned previously) that will be populated using the crawl function. By passing the same website in for each data structure at each depth, we are ensured that each data structure will be populated with the same KeywordEntry objects, so the comparisons can be made on the functionality of the data structures alone.

The website that I used for this experiment, http://compsci.mrreed.com, is arbitrary in its structure and thereby simple to navigate.



placement placement placement placement placement placement placement placement placement placement defect slide limb sirs puff lash annexs pushdown sides balloon passenger corrosion operabilities briefing cork pea knobs soils mitts realinements finishes offers investigator lead interview highway pump skins cot battle chips dawns mop suppressions rescues legend courtesies confinement lip assistant enclosure

If I was to run this function using http://foothill.edu, we could expect the time to crawl and store the list of words found on the website at each depth to be much higher. Even with this, the function will not be able to access websites that require log-in credentials. As such, we can conclude that terms that may have been excluded, for example, a slang term that might be used on a social media website, will not be excluded from this experiment.

Acknowledgments

The experiment did not always produce results such as these. It is my belief that these results best demonstrate the desired effect of this experiment.

Crawl and Store

	DEPTH 0	DEPTH 1	DEPTH 2	DEPTH 3
BST	0.14	1.16	15.12	123.83
SPLAY	0.11	1.09	17.14	124.34
AVL TREE	0.11	1.12	13.97	125.94
HASH TABLE	0.13	1.23	13.57	122.3
WORDS FOUND	38	598	3920	5298

Depth 0

As the values are too small to see significant differences between times, we can only make estimations. Because the first 9 words found on the website are the same, splay tree would perform well at this depth as it allows fast access to elements that were recently accessed.

Similarly, because the AVL tree is self-balancing, this will make it easier to traverse in order to find the desired object.

The Hash table may also take longer because of the fact it requires more code to find/insert a node into the table at a smaller size.

Depth 1

At depth 1, we see something of a continuation of the trend from Depth 0.

Splay now proves itself to be faster than the AVL tree. In my opinion, this is because only a few more links are found from the starting web page, and so the ability to recall recently used elements makes it the fastest data structure of the four.

In a similar vain, AVL Tree was the second fastest, because the self-balancing means that the shorter length of the data structure that is required at this depth allows the smaller amount of elements to be easily found or inserted.

Because there are now more values in the data structure, the inability for a BST to balance itself becomes an even larger factor in the appending and inserting of elements.

Hash table, because it had to be resized from it's original size of 98, took the longest to crawl and store. The rehashing will factor into the crawl and store time, but pays off in regards to the search (which we will revisit later), and higher depths.

Depth 2

Depth 2 is where various advantages of data structures that have been useful up until now become hinderances, and vice versa.

The perfect example of this is the splay tree. While it was the fastest until now, it is now the slowest data structure to crawl and store the elements. Splay trees have always been used for quickly sorting information, but with smaller packets of information. An example of this is a network router, which would quickly receive a packet and send it out based on a table. This example demonstrates how relying on splay tables for larger programs is not what it was built for.

Similarly, with the binary search tree, while it was shorter than the splay table, because it has no way of keeping itself balanced, will result in long loading times for the user.

With some acknowledgement of the AVL tree's consistently good performance relatively, the hash table shines in this scenario. While extra time needs to be taken to rehash for the larger number of elements, it is able to outperform the other data structures because it's average Big O time for search/insertion is O(1).

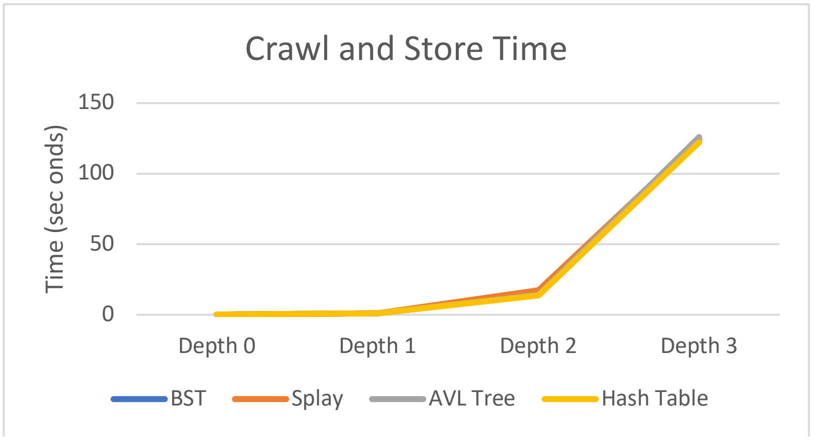
The relatively quick process of hashing and finding the bucket, or not, results in very fast runtimes in this case.

<u>Depth 3</u>

Depth 3 sees a continuation of hash table outperforming the other data structures, but some inconsistency with the results from the splay/BST/AVL trees. This may be due to a fault while the program was running.

This may have been due to the fact that the website that was being combed was relatively unrealisitic in it's layout. Were we to perform this experiment on a website such as foothill.edu, we would expect to see an increase in the amount of words that we had seen up to this point, around 10000% with each level of depth.

A graphical representation of the data shows as such:



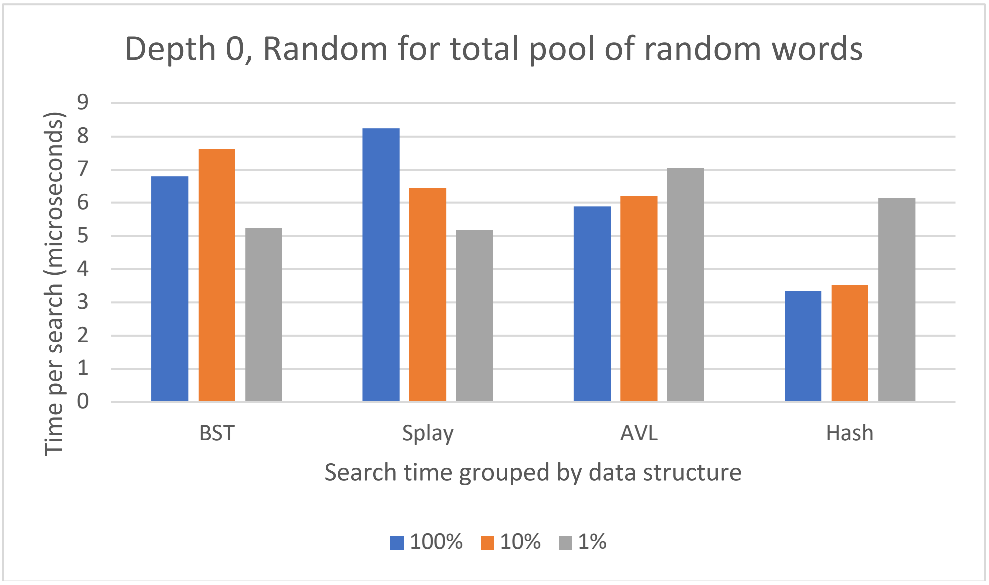
Search

For this section focusing on search, I will analyze results for data structures against each other for known and unknown words at each depth, and then compare the known and unknown results.

Random Words

Depth 0

RANDOM FOR TOTAL POOL OF RANDOM WORDS	38	3	1
BST	6.96	6.05	8.03
SPLAY	6.81	4.1	4.5
AVL	4.9	4.8	6.11
HASH	3.45	3.38	3.74

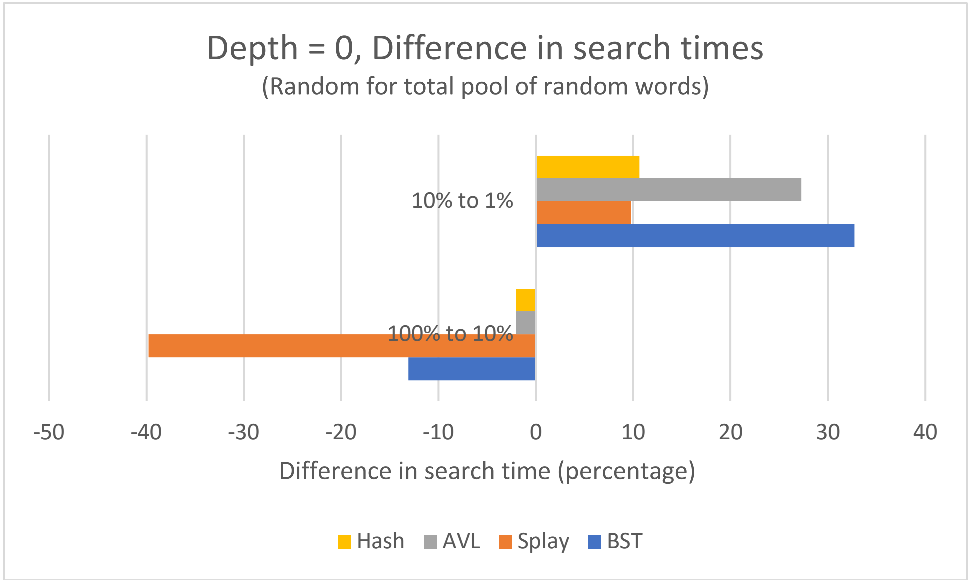


This graph demonstrates how the data structures fared when searching for smaller proportions of the words from the pool of random words.

To capture the difference in times properly, I felt the best way to express this would be through the change in search times as a percentage.

RANDOM FOR TOTAL POOL OF RANDOM WORDS	100% TO 10%	10% TO 1%

BST	-13.074713	32.72727273
SPLAY TREE	-39.79442	9.756097561
AVL	-2.0408163	27.29166667
HASH	-2.0289855	10.65088757



The splay table demonstrates the largest difference in search times when searching for 10% of the words compared to 100% of the words. The BST demonstrates the second highest swing, but it is less than half as great as the Splay table.

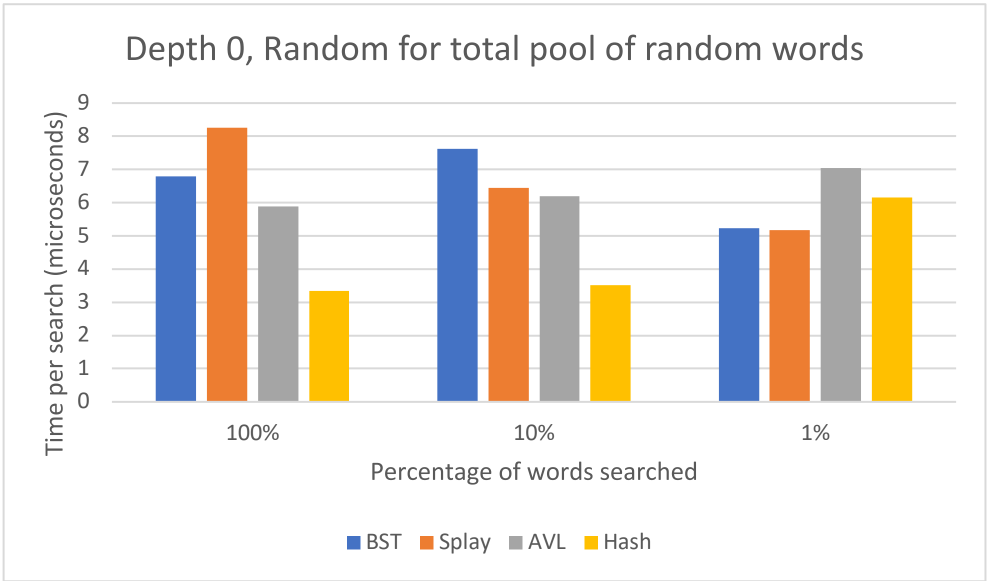
I believe this reaffirms the idea put forth in the crawl and store that Splay tables are better equipped to handle smaller operations. Another interesting note is that the

It is clear that the Hash table had the smallest difference between the different proportion searches, while the BST had the largest swing when narrowing the search from 10% to 1%.

This demonstrates that when finding a single value, it was the best equipped.

For a graph measuring how the data structures performed relative to each other, please look to the next page.

Interestingly, the hash table and AVL tree increases in time per search for each of the proportions, which we may contribute to program error, or the fact that because they are more meticulously sorted than the BST and splay table, means that their operations take longer on a smaller sample size.

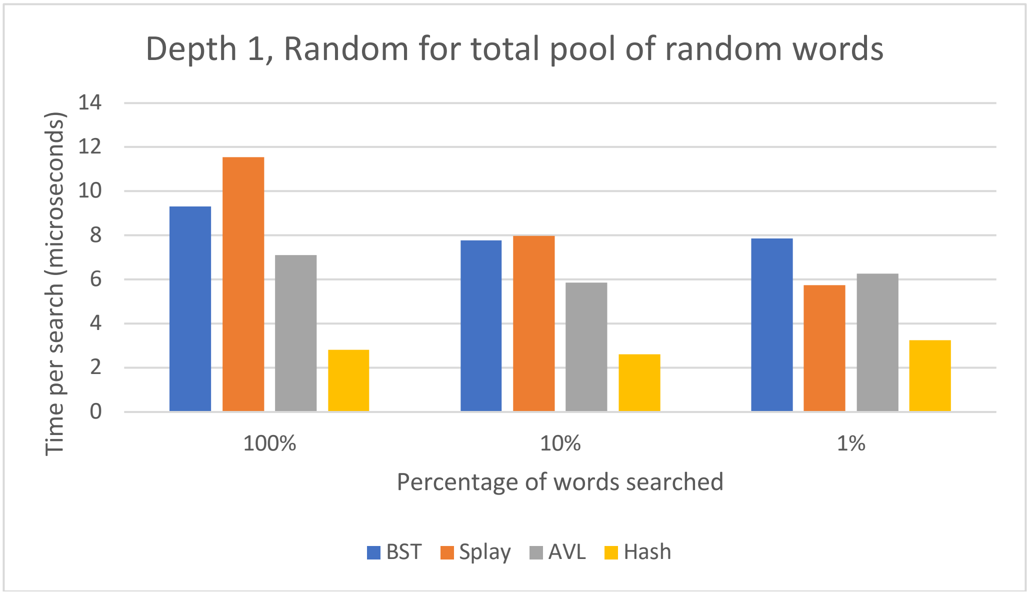


From this graph, it is clear that the value we would want to use at this level of depth, when searching for a single word, would be a splay table. Based on what was previously discussed, we knew that this would be the case.

A hash table is the data structure when searching for a large percentage of the words, which we already hypothesized would be the case based on how what we know about it's Big O runtime and ability to quickly search using the hashing method.

Depth 1

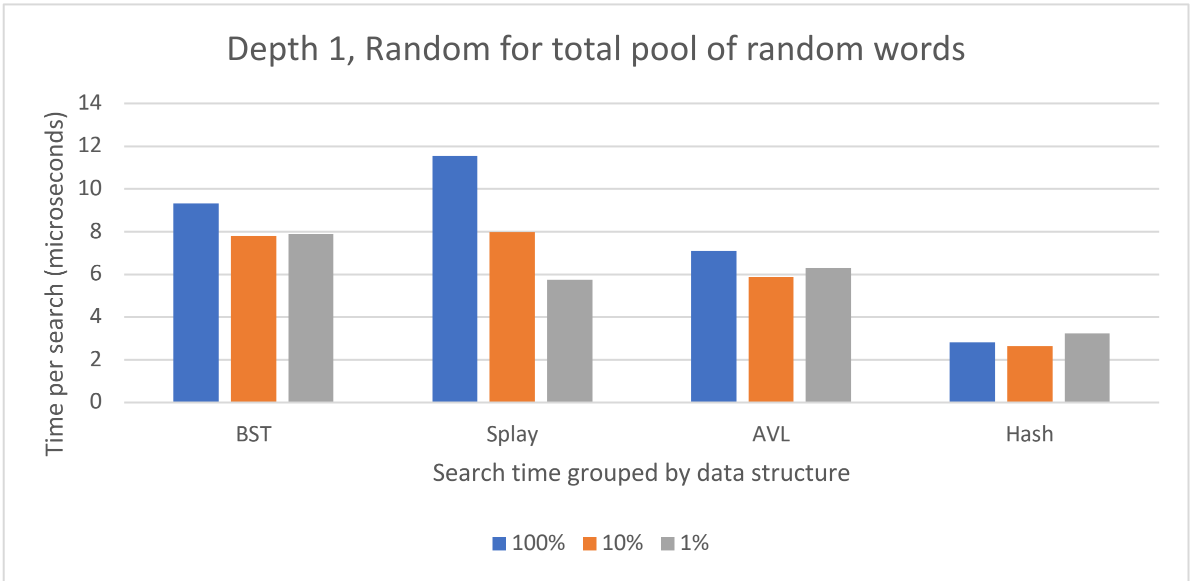
RANDOM FOR TOTAL POOL OF RANDOM WORDS	**100%**	**10%**	**1%**
BST	9.33	7.79	7.88
SPLAY	11.55	7.98	5.76
AVL	7.11	5.87	6.28
HASH	2.82	2.62	3.24



Exploring the time taken for each search with a given proportion of words, it is clear that the hash table outperformed all of the other data structures. Interestingly, the splay table demonstrates that it's performance continues the same trend of almost having it's runtime when narrowing the search for each proportion.

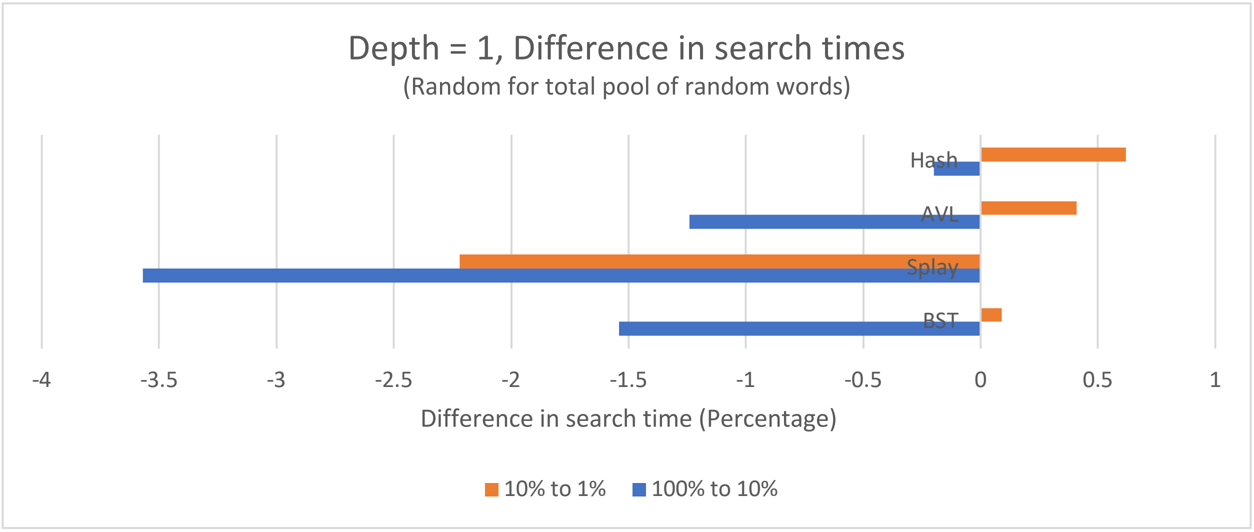
The AVL tree and hash table also perform similarly at this depth, performing faster when searching for 10% of the words but then slower when searching for 1%.

However, it is clear from the following tables that they are faster, the hash table dramatically slow, in a relative test against the other data structures.
Also of note is the splay tree, which outperforms the AVL tree at 1%. This reaffirms the hypothesis put forth in the crawl and store section that it is suited to quick searches.



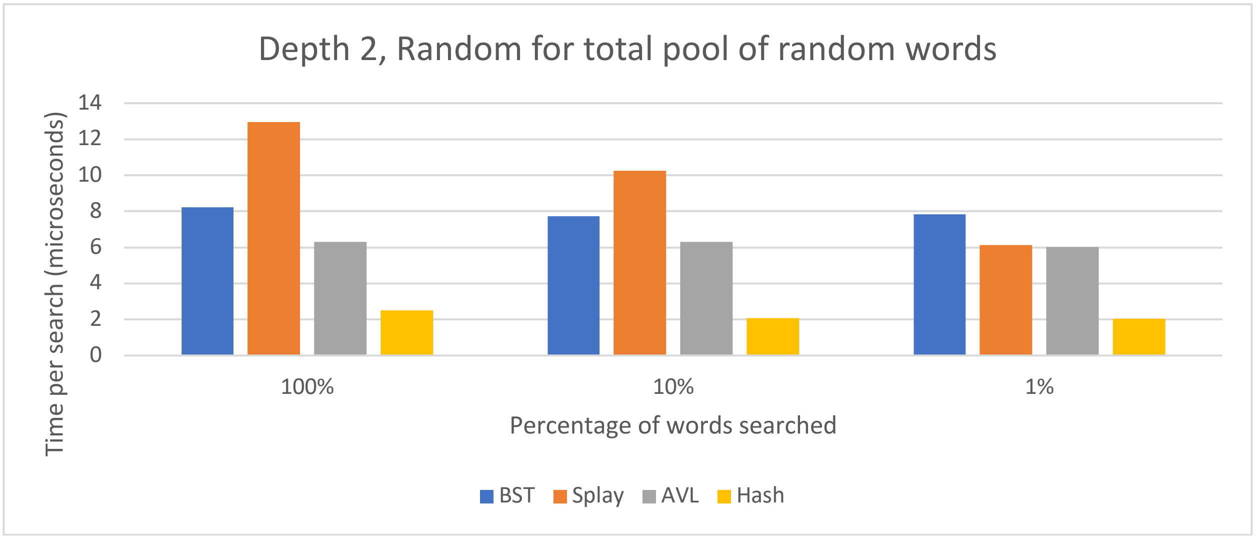
When observing the difference in times, it is clear the splay table consistently builds upon it's performance when dealing with a smaller search size, while the hash table enjoys the smallest swing amongst the data structures.

RANDOM FOR TOTAL POOL OF RANDOM WORDS	100% TO 10%	10% TO 1%
BST	-1.54	0.09
SPLAY	-3.57	-2.22
AVL	-1.24	0.41
HASH	-0.2	0.62



Depth 2

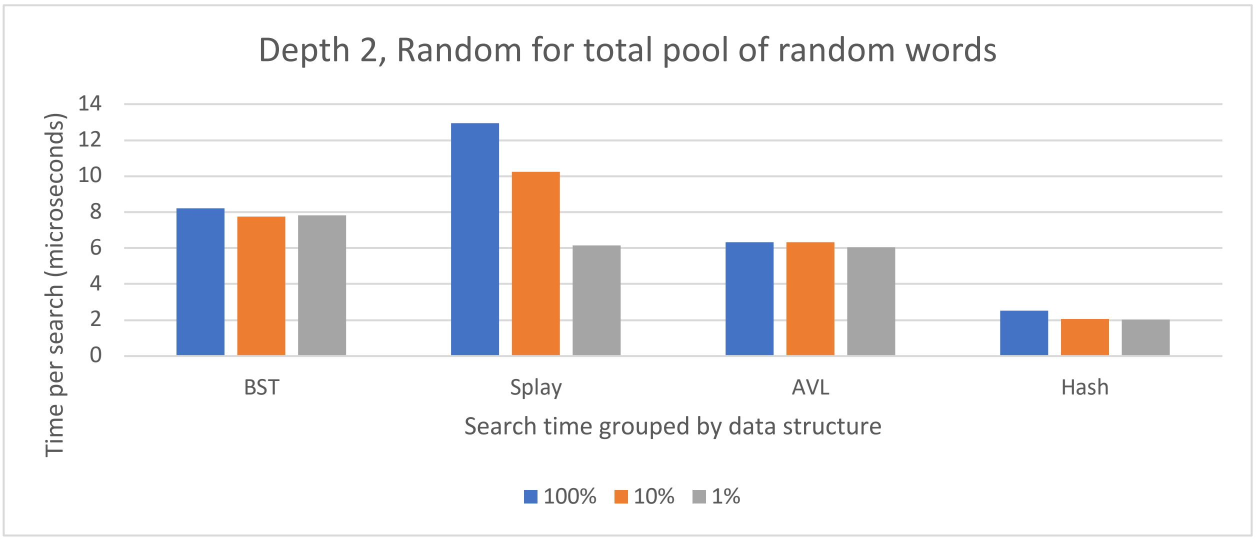
RANDOM FOR TOTAL POOL OF RANDOM WORDS	100%	10%	1%
BST	8.21	7.74	7.82
SPLAY	12.94	10.24	6.14
AVL	6.32	6.32	6.03
HASH	2.52	2.07	2.03



At this depth, the hash table is again the highest performer, with the AVL tree in second place. This may suggest that our results for the previous depth were an outlier, or something happened while the program was running in order to throw the results.

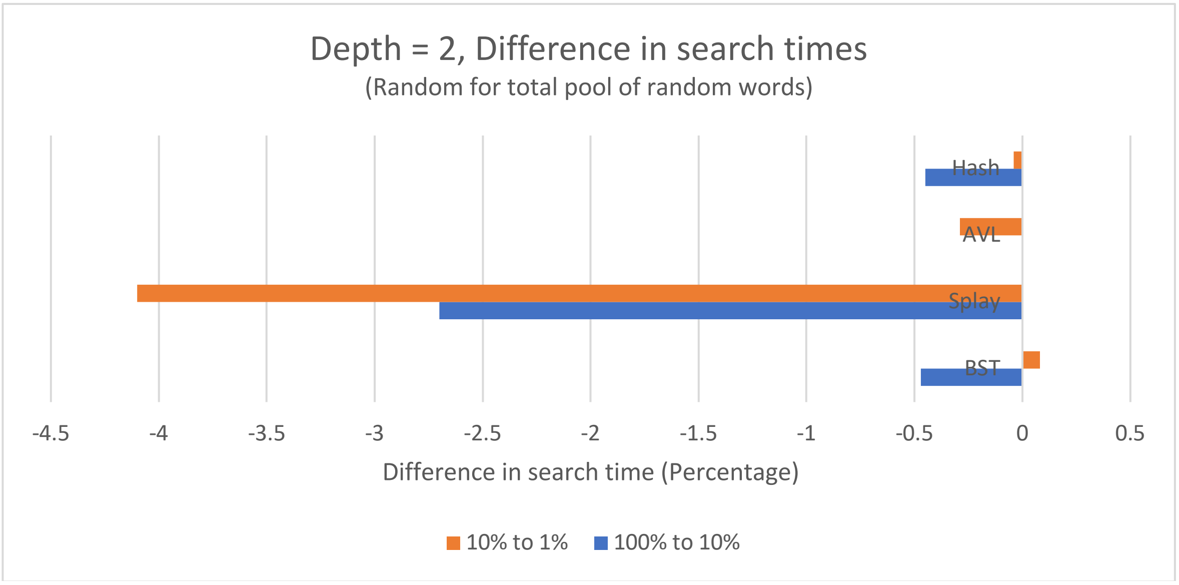
At such a short run-time, this would have a great effect on our results.

With the number of words now almost 6x larger than the previous depth, the splay tree is unable to surpass the AVL tree at one percent, although it does enjoy a large swing, again, from 100% through 1% searches.



RANDOM FOR TOTAL POOL OF RANDOM WORDS	100% TO 10%	10% TO 1%
BST	-0.47	0.08
SPLAY	-2.7	-4.1
AVL	0	-0.29

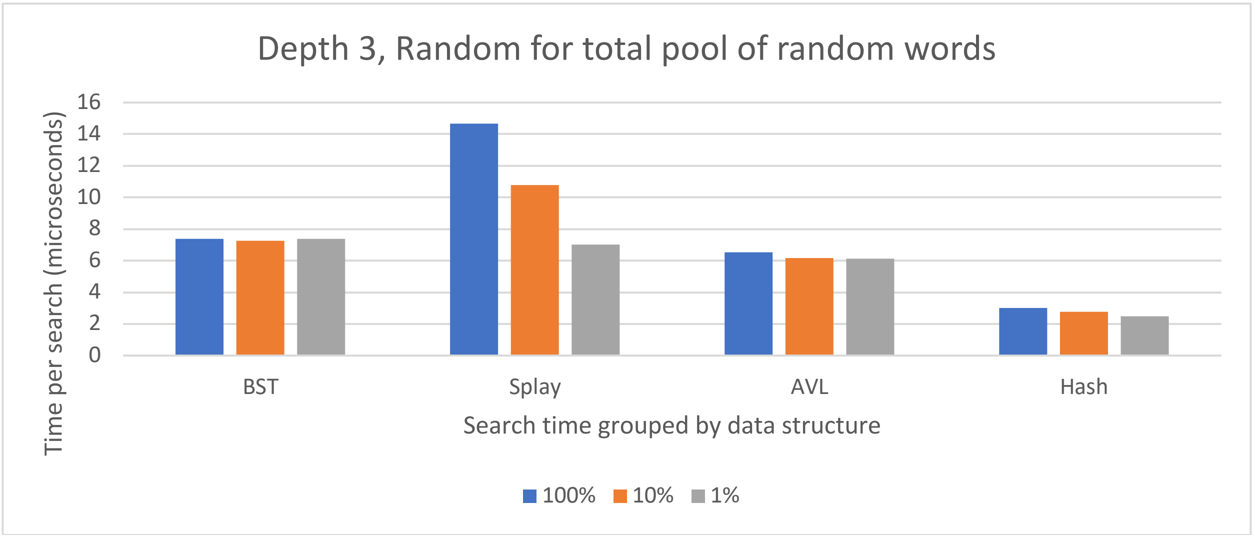
HASH	-0.45	-0.04



With this graph, we start to see some trends develop from the previous depth to this one.

<u>Depth 3</u>

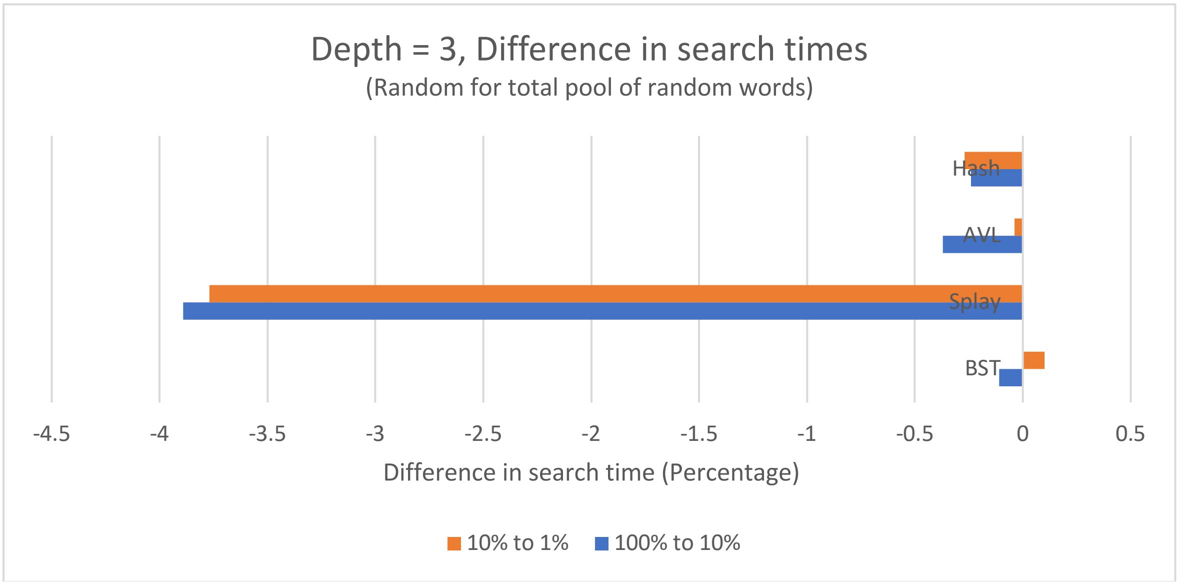
	100%	**10%**	**1%**
BST	7.38	7.27	7.37
SPLAY	14.66	10.77	7
AVL	6.54	6.17	6.13
HASH	2.99	2.75	2.48



This depth continues the trend we saw from depth 2. The BST will quicken when searching for 10% of the results, and then

	100% TO 10%	10% TO 1%
BST	-0.11	0.1
SPLAY	-3.89	-3.77
AVL	-0.37	-0.04
HASH	-0.24	-0.27

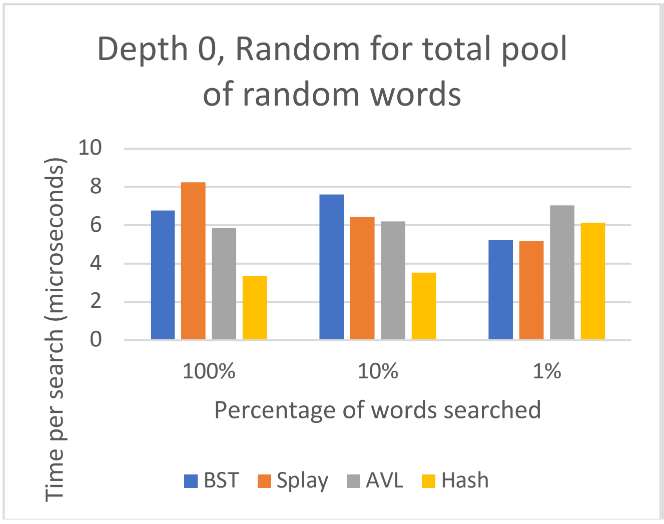
The swing in times is interesting at this depth, as it demonstrates the search times consistently get better no matter the proportion of words being searched.

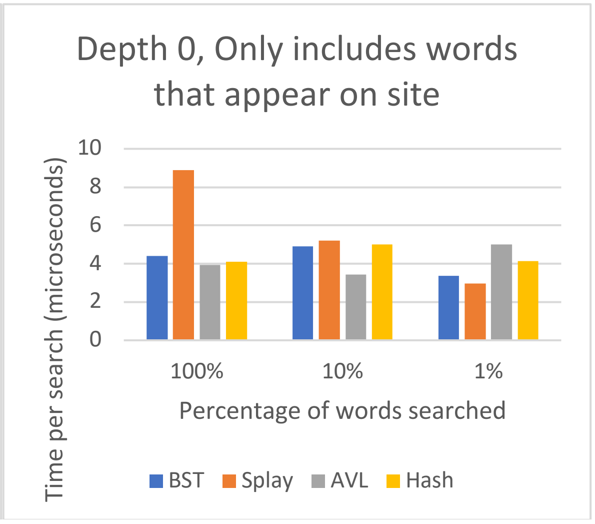


<u>Only includes words that appear on site</u>

<u>Depth 0</u>

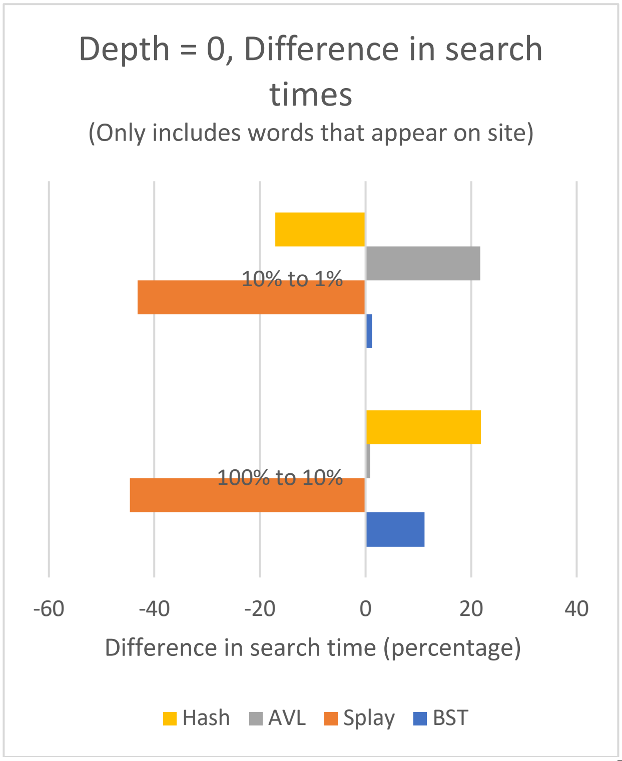
ONLY INCLUDES WORDS THAT APPEAR ON SITE	100%	10%	1%
BST	4.4	4.92	3.37
SPLAY	8.88	5.21	2.97
AVL	3.93	3.45	5
HASH	4.11	5.01	4.15

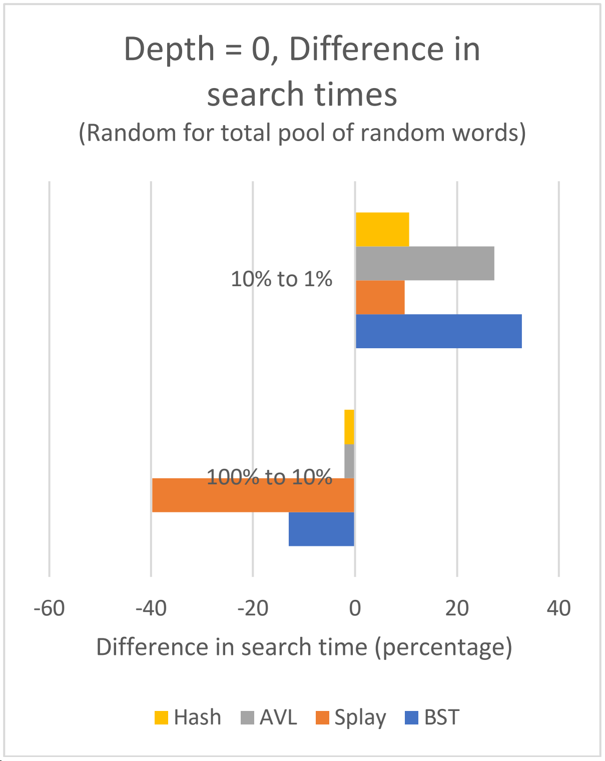




Comparing these results to the previous section, it is interesting to observe how the different data structures behaved at each level. The AVL tree appeared to perform the best in two for the words that appear on the site, and the splay tree when searching for 1%. This compared to the hash table performing the best for the random words.

This could be attributed to how the data structures behave. We see a similarly different set of results for the performance of the search times.

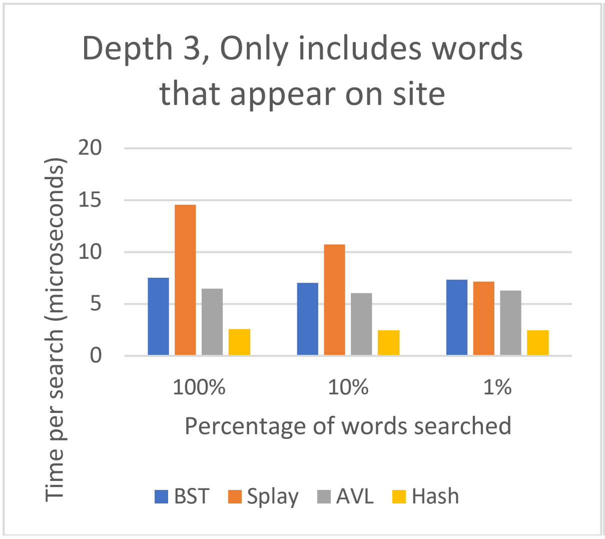


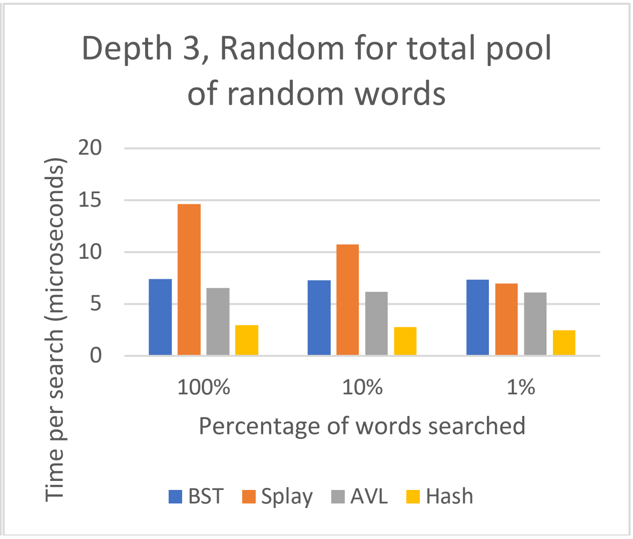


Whereas the random words performance dropped off, the difference was not as marked for the random words.

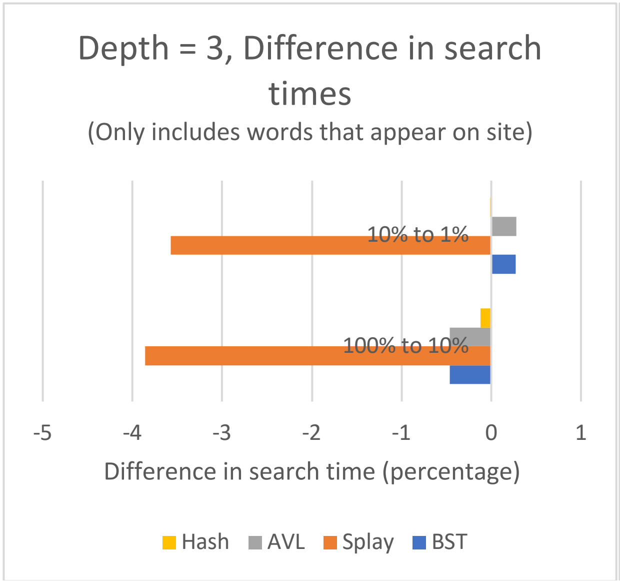
To see the true swing, we look to how the data structures perform comparatively at Depth 3:

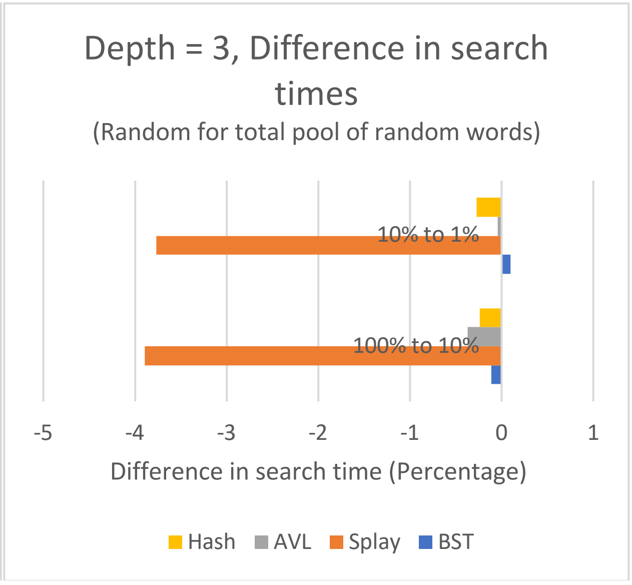
We now see that when dealing with only words found on the site, the two types of searches are almost exactly the same. This would make sense as the results have consistently proven that when dealing with larger programs, the data structures will act in a manner we would expect.





When comparing the difference in search times at depth 3, it is striking that the hash table and the splay tree both perform better with each level of search, whereas the AVL tree and the BST drop off in performance for words that appear on the site.





Conclusion

It is clear from the results of our experiment that the hash table is by far the best data structure to use in this case.

From previous lessons, we know that the hash method does not employ any kind of binary logic to narrow down a set of results, but rather uses the hashing algorithm to create a unique index that cuts down time for removing, inserting, and searching for elements.

While the runtime for the crawl and store method may fall short at the beginning when only faced with under 1000 words, with the goal of eventually creating a search engine, the hash table demonstrates time and time again that is able to crawl and store the data in a relatively quick amount of time.

This fact, coupled with the far outperformance against the other data structures, makes it the clear favorite when performing this kind of operation.