Asgn7 Writeup by Derick Pan
Derick Pan - dpan7
Cse13s

Time Complexity

ADT/what i'm referring to:	Time Complexity:	Explanation:
Hash Tables	O(1) search times on average.	Within the hash table, we map keys to values which in turn gives us a O(1) look up time. We aren't iterating over values as we can find the key through hashing.
Bloom Filter	Insertion and Search will both take O(k) time. K for hash functions. Space of the actual data structure is O(M) time. M for bits.	Bloom filters are very fast and memory efficient. Given that, the interest we're looking at is the probability of False Positives. If more items are in a bloom filter, then the higher the chance of false positives. To resolve this we can increase the amount of hash functions to reduce false positives.
Doubly Linked lists	Insert: O(1) Search: O(n)	Insert has a time complexity of O(1) because in this assignment we are inserting at the head of the linked lists, without any need to iterate. Search has a time complexity of O(n) because we need to traverse the list from the head until we find the element we're searching for.

Do linked lists get Longer?

The better question is which linked lists get longer with respect to hash table or bloom size? The badspeak and oldspeak linked lists to keep track of which words had regressions that did not change in size. Although, a fact to note is that as you increase the size of Hash table or Bloom size, more storage is used, and in turn Time complexity could grow especially if MTF is not used.

Now how about the linked lists used by Hash tables to resolve hash collisions? If the hash filter size is too low, then more linked lists may be used but not necessarily longer. If the length of the linked list is too long, then the lookup time increases from O(n) to O(1) which would actually lose the entire purpose of a hash table. It is memory intensive, but faster and more efficient.

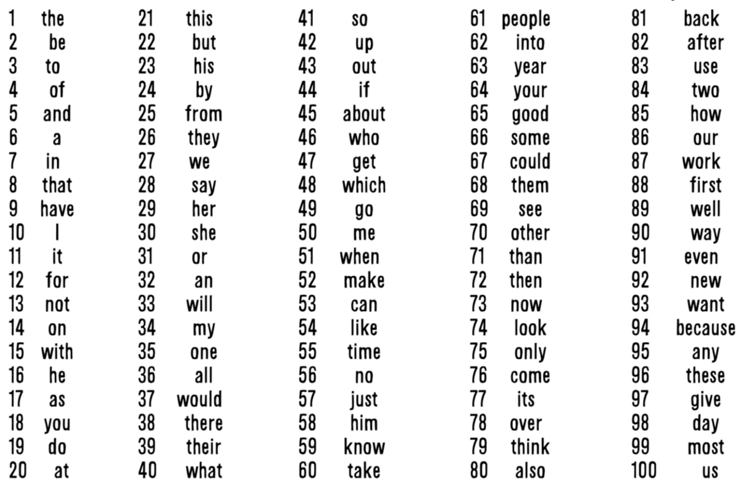
How does the number of links followed without using the move-to-front rule compare to the number of links followed using the rule?

Test file:	alice29.txt	bib	book1	book2	paper1
WITH move-to-front rule	76358 links	85931 links	616254 links	150336 links	17532 links
WITHOUT move-to-front rule	431630 links	394179 links	9591302 links	1659463 links	45028 links
Percentage increase in links from with to without MTF	465.271%	358.716%	1456.39%	1003.84%	156.833%

Without move-to-front (MTF)	With move-to-front (MTF)
The reason why the amount of links is SIGNIFICANTLY higher than with MTF is due to the sheer mass of data we need to look up. For example, Let's say we have a list of 10,000,000 elements. Now we want to look up the word "the." (The most commonly used word in the english language.) But that word is at the very end of the list. Each time we want to find that word, we need to increment the number of links by 9,999,999. Only if there was a way where we could move that word to the beginning of the list because we'll likely be looking it up a lot. AND THERE IS!!!! We use the MTF rule, where we move the elements we look up to the front of the list.	The number of links without MTF is significantly larger than with MTF. This is because MTF moves the item we "looked up" to the front of the list. According to Zipf's law *"the most frequent word will occur about twice as often as the second most frequent word, three times as often as the third most frequent word, etc."* - https://www.sciencedaily.com/releases/2017/08/170810082147.htm So because we're moving a frequently used word in the beginning of the list, we don't have to increase the time complexity and iterate through every word in the list until we find the word we're looking for. Even if we move up an uncommonly used word, it's still better than the scenario I stated to the left.

For example, This list shows the top 100 most used words in the english language.→

If the MOST commonly used word "the" was at the tail of our linked list. It would take us *O(n)* time to figure out where it is.
But since we're moving our lookup word to the front of the list, the chances are that we will search for it again. And this time it'll hopefully be less than *O*(*n*) time, in the majority of cases.

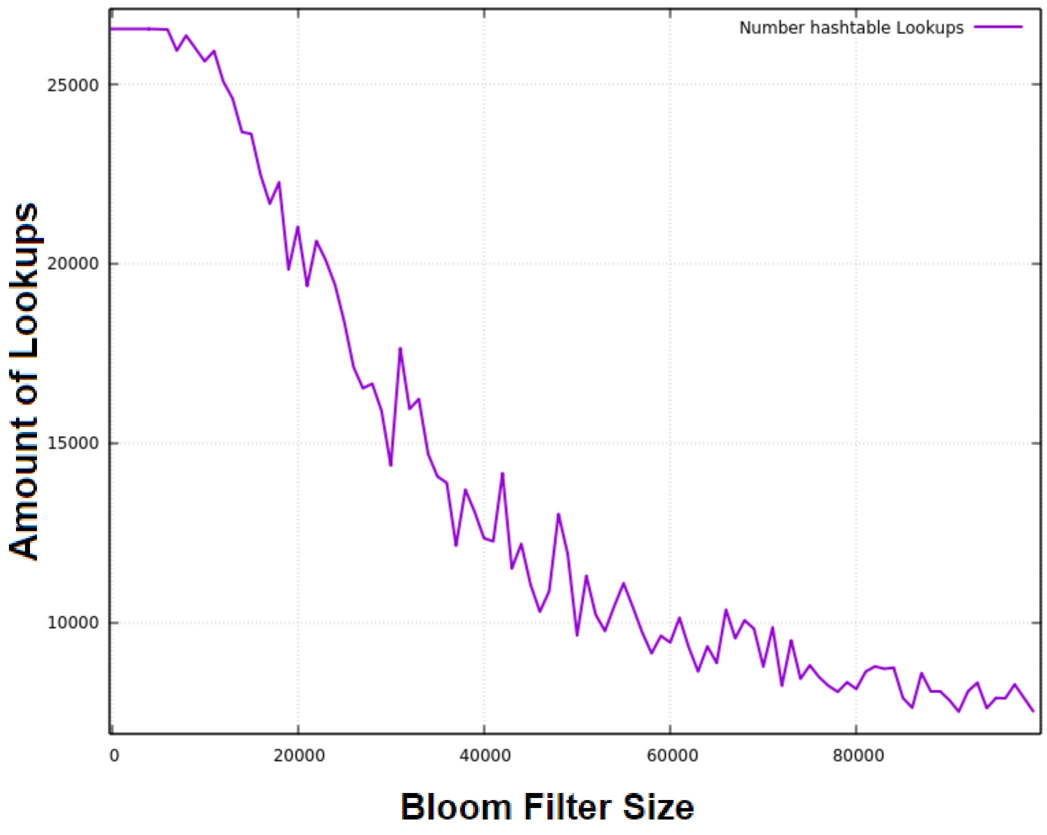


1	the	21	this	41	so	61	people	81	back
2	be	22	but	42	up	62	into	82	after
3	to	23	his	43	out	63	year	83	use
4	of	24	by	44	if	64	your	84	two
5	and	25	from	45	about	65	good	85	how
6	a	26	they	46	who	66	some	86	our
7	in	27	we	47	get	67	could	87	work
8	that	28	say	48	which	68	them	88	first
9	have	29	her	49	go	69	see	89	well
10	I	30	she	50	me	70	other	90	way
11	it	31	or	51	when	71	than	91	even
12	for	32	an	52	make	72	then	92	new
13	not	33	will	53	can	73	now	93	want
14	on	34	my	54	like	74	look	94	because
15	with	35	one	55	time	75	only	95	any
16	he	36	all	56	no	76	come	96	these
17	as	37	would	57	just	77	its	97	give
18	you	38	there	58	him	78	over	98	day
19	do	39	their	59	know	79	think	99	most
20	at	40	what	60	take	80	also	100	us

How does changing the Bloom filter size affect the number of lookups performed in the hash table?

Graph is based off of alice29.txt
Badspeak and Newspeak words are from the asgn7 supplied documents

The amount of lookups performed on the hashtable (Y-Axis) decreases as the Bloom Filter size (X-Axis) increases.



Increasing the Bloom filter size decreases the number of lookups performed on the hash table. But why is this?

We can see from the graph that a Bloom Size filter of 10,000 has about 26,000 lookups, whereas the Bloom Filter size of 80,000 only has 5,000 lookups. The formula to the right shows the Probability of False Positives. Where M is the number of bits in the array, K is the number of hash functions, and N is the number of inserted elements. We can reduce this false positive rate by increasing the length of the bloom filter's bit vector. (and by adding more hash functions, but we're not adding more hash functions today.)

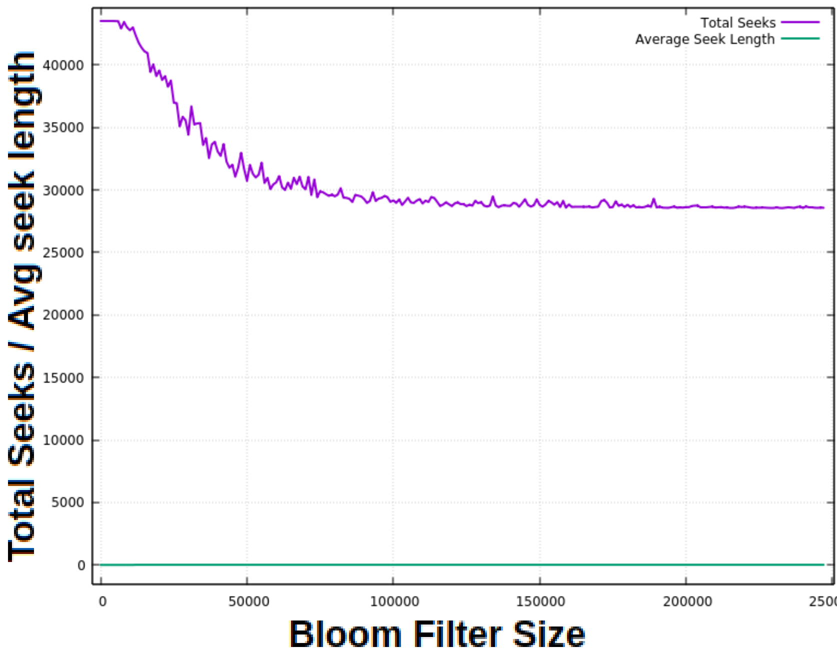
$$p = (1 - e^{-\frac{K \times N}{M}})^K$$

We started off with a small bloom filter, our code gave us more false positives and believed that the words we were reading were already in the bloom filter, because all the bits in the bit vector were likely all set. Therefore, requiring more hash table lookups.

But as we increased the Bloom Filter size, we received more accurate results on whether or not a word was in the bloom filter. So we didn't need to do as many hash table lookups. The downsides to this increased Bloom filter size and decreased Lookup count is the cost of time and space. With many fast acting systems in 2021, we can't always do this because things would run a lot slower.

Increasing Bloom Filter Size

The Total amount of Seeks and Average Seek Length when I'm increasing the bloom filter size from 1 to 30,000 for alice29.txt produces this graph. The Total Seeks has a very noticeable decrease as the bloom filter size increases. The Average seek length has minimal to no change, starting from average of 10, then very slowly increasing the average seek length by 1.



Why does the Total Seeks decrease and the Average seek length hardly change whatsoever?

Similarly to the previous page, increasing the bloom filter size reduces false positives. Inside of our banhammer code, if word is in bloom filter then no action needs to be taken. Else, we first check if a word is in the hash table, (might increase the total seeks), otherwise we will have to insert a word into a linked lists (Increases total seeks). With less false positives, we receive more accurate results, and with less false positives, we don't need to seek as often.

But why does the Average seek length stay so steady when the Total seeks has a very characteristic decline?

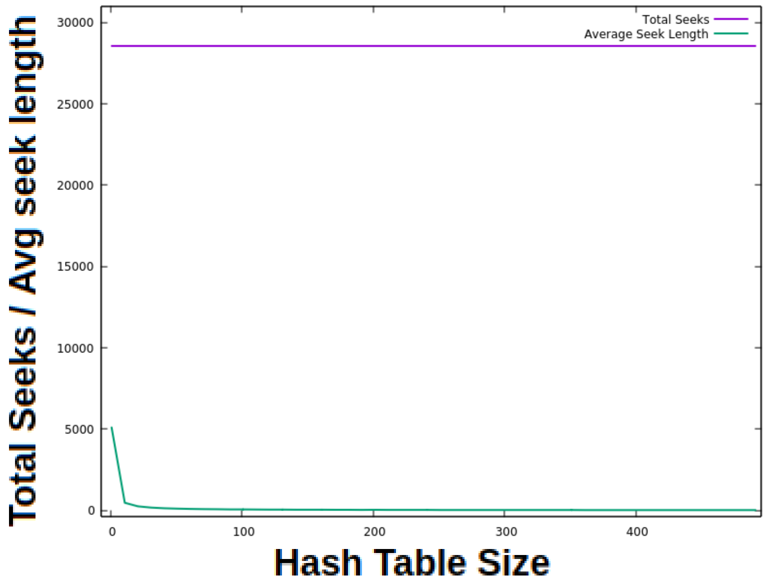
It doesn't have anything to do with MTF. If MTF was activated, then the Average seek length would only have a smaller value. The average seek length stays fairly constant because the amount of links with respect to seeks grows together, so there's always a constant average.

Increasing Hash Table Size

In contrast from the previous page, the Total Seeks has little to no change and The Average Seek Length has a dramatic decrease in the beginning, then a VERY SLOW decrease. with respect to the text file alice29.txt

A similar graph is produced for other files like book1.

I only made this graph go up to 500, because with this text file, anything greater than that just looks like a constant value.



Why is this the case?

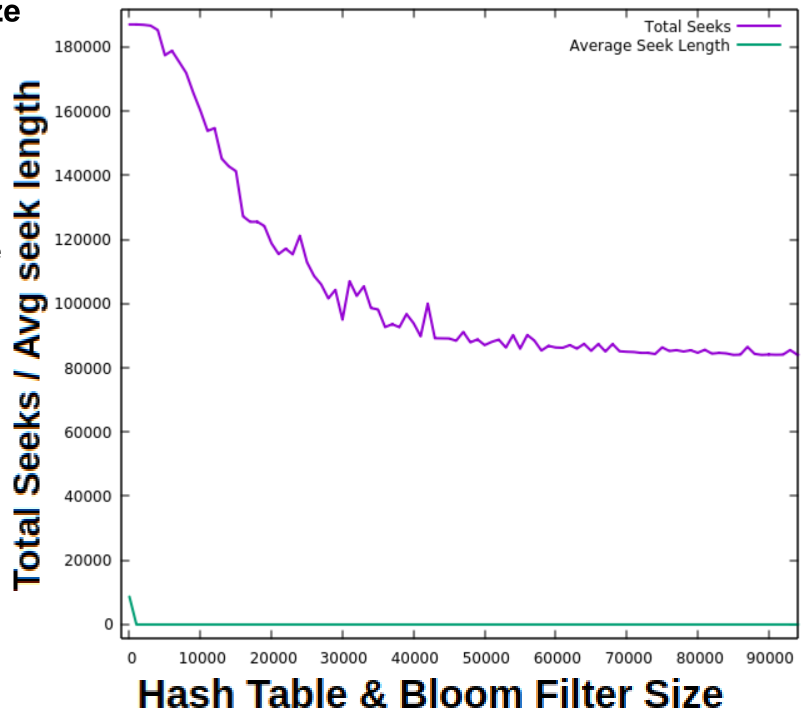
The total seeks hardly changes value because within this code there's still a need to perform a lookup in the hash table, which also performs a linked lists lookup. If per say we were to add more hash values, the graph may look a lot more different.

The Average Seek Length decreases a lot in the beginning because there was a far greater number of links as compared to seeks. With a small hash table, and a standard Bloom filter Size of 2^20, there's a significant amount of elements to iterate through for lookup. As the Hash table size increased, less linked lists lookups were required, which in turn dramatically decreased the amount of links compared to seeks. Would turning on MTF change anything? Hardly, The average seek length will start off slightly lower but nothing of significance.

Increasing Both Hash size and Bloom Filter Size

The Graph on the right is using badspeak and newspeak from the asgn7 resources, filtering book1.

In this graph, we linearly increased both the Bloom Filter size and Hash table size. We can see that the total amount of Seeks decreases while the Average seek length decreases a lot in the beginning, then very slowly. This graph is very similar to when we were only increasing the Bloom Filter size, and with similar reasoning. With a larger bloom filter size, there's less false positives, which in turn reduces the total amount of seeks required.


Total Seeks
Average Seek Length
Total Seeks / Avg seek length
Hash Table & Bloom Filter Size