A3: Data Structures and Text Editing

Performance Evaluation

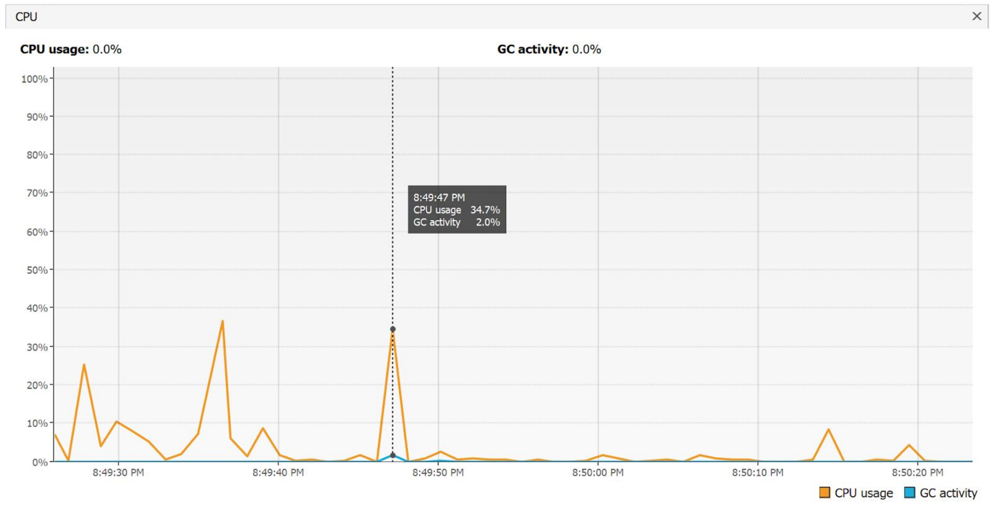
Choice of Data Structures for Text Editor

For AutoComplete and Search, the choice of data structure seemed fairly straightforward: it seemed unlikely that we would be able to implement anything more efficient than the provided String method for searching, and the Trie's `closestWordToPrefix` was a natural fit for AutoComplete. However, SpellCheck seemed viable using all three of the data structures we implemented, so we ran some tests comparing them.

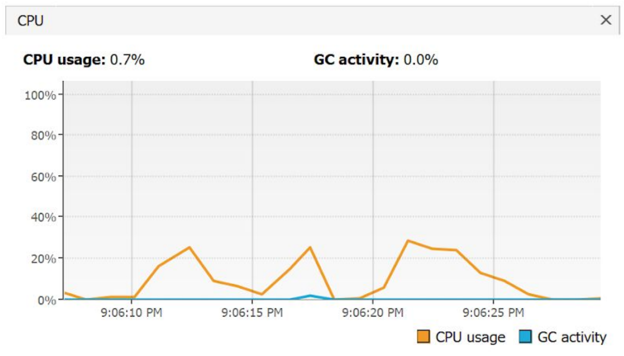
During these tests, the `AutoComplete` module (based on Trie) and the `StringSearchModule` (based on `String.indexOf()`) were both running.

While there didn't appear to be dramatic differences between the different classes, BloomFilter appeared to be perhaps the most conservative in terms of memory usage. Trie seemed to be the most intensive. This could be because it uses HashTable to store the children of each node, meaning there are many more objects created than might be strictly necessary for spell checking. Obviously the BloomFilter does have a risk of false positives, but with a random selection of words and non-words typed into the editor, all three marked the same ones as being incorrect. We used an online calculator to find array size/hash function/element ratios to keep the rate of false positives for the BloomFilter to around 0.01%, which seems viable for every-day usage.

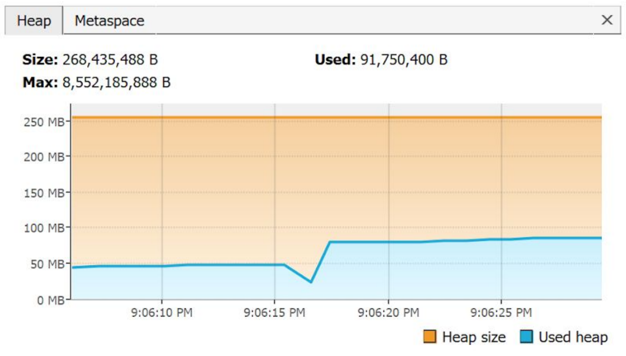
BloomFilterSpellCheck:
Loading the dictionary file (CPU usage 34.7%):



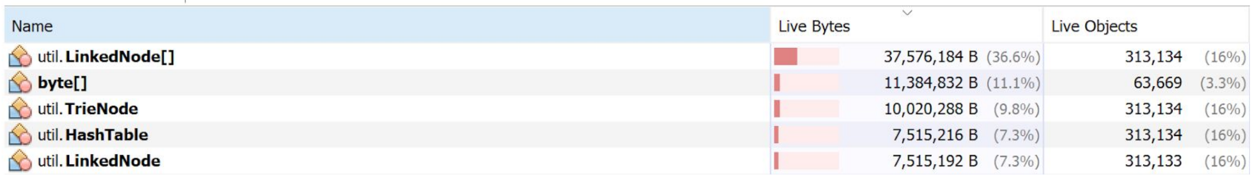
During use:


CPU

CPU usage: 0.7% **GC activity:** 0.0%


Heap Metaspace

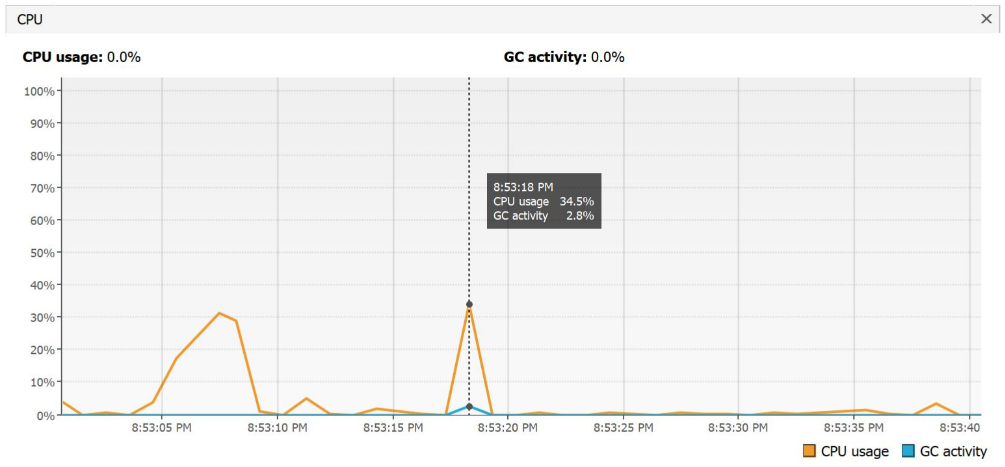
Size: 268,435,488 B **Used:** 91,750,400 B
Max: 8,552,185,888 B



Name	Live Bytes	Live Objects
util.**LinkedNode[]**	37,576,184 B (36.6%)	313,134 (16%)
byte[]	11,384,832 B (11.1%)	63,669 (3.3%)
util.**TrieNode**	10,020,288 B (9.8%)	313,134 (16%)
util.**HashTable**	7,515,216 B (7.3%)	313,134 (16%)
util.**LinkedNode**	7,515,192 B (7.3%)	313,133 (16%)

HashTableSpellCheck:

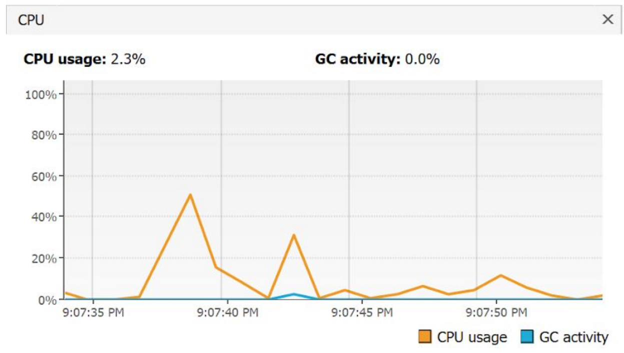
Loading the dictionary file (CPU usage 34.5%):


CPU

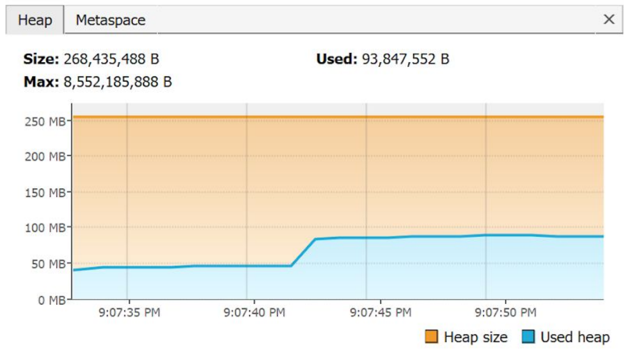
CPU usage: 0.0% **GC activity:** 0.0%

8:53:18 PM
CPU usage 34.5%
GC activity 2.8%

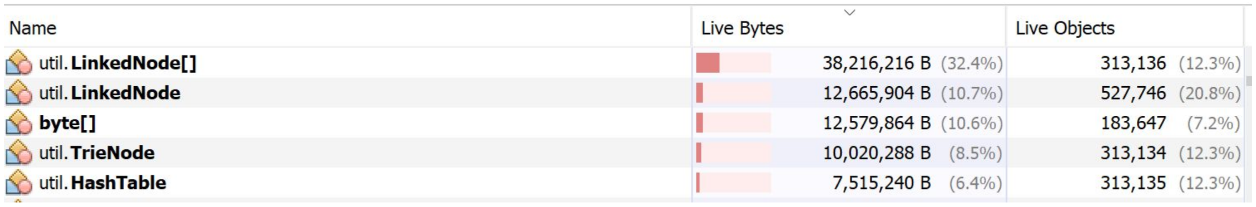
During use:


CPU

CPU usage: 2.3% **GC activity:** 0.0%

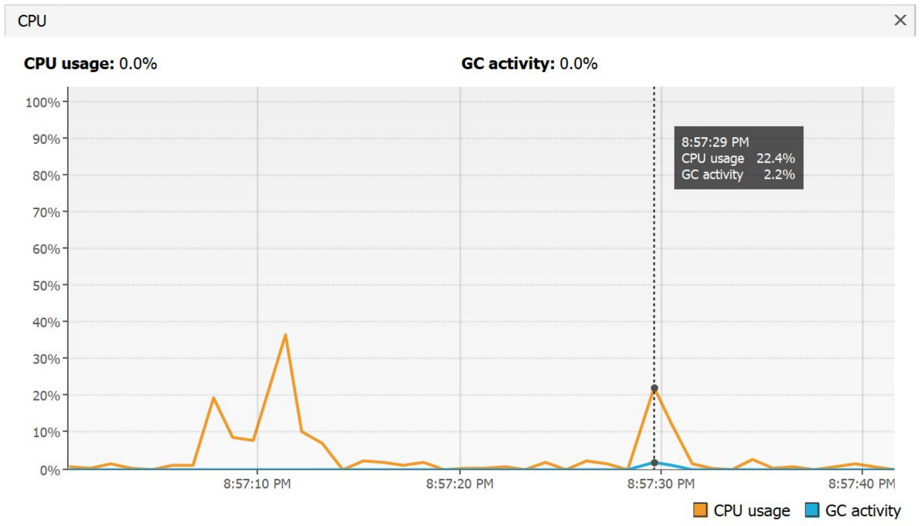

Heap Metaspace

Size: 268,435,488 B **Used:** 93,847,552 B
Max: 8,552,185,888 B

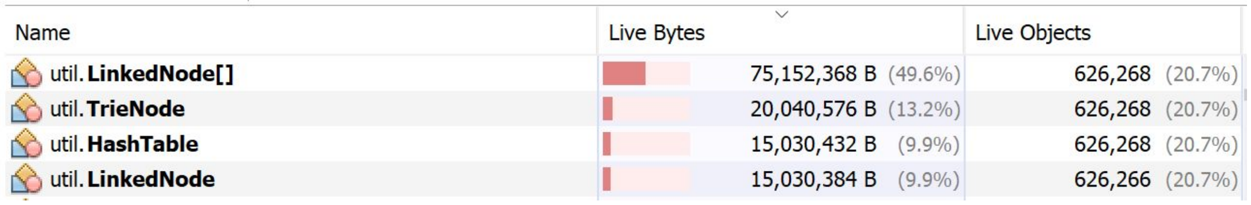


Name	Live Bytes	Live Objects
util.**LinkedNode[]**	38,216,216 B (32.4%)	313,136 (12.3%)
util.**LinkedNode**	12,665,904 B (10.7%)	527,746 (20.8%)
byte[]	12,579,864 B (10.6%)	183,647 (7.2%)
util.**TrieNode**	10,020,288 B (8.5%)	313,134 (12.3%)
util.**HashTable**	7,515,240 B (6.4%)	313,135 (12.3%)

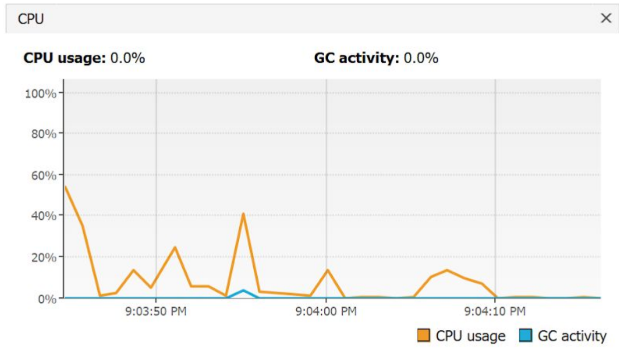
TrieSpellCheck:
Loading dictionary file (CPU usage 22.4%):

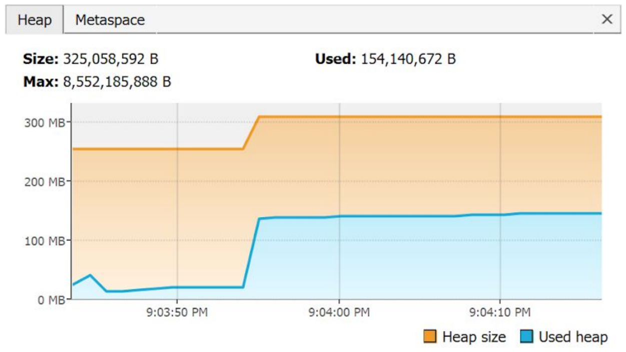


During use:



Name	Live Bytes	Live Objects
util.**LinkedNode[]**	75,152,368 B (49.6%)	626,268 (20.7%)
util.**TrieNode**	20,040,576 B (13.2%)	626,268 (20.7%)
util.**HashTable**	15,030,432 B (9.9%)	626,268 (20.7%)
util.**LinkedNode**	15,030,384 B (9.9%)	626,266 (20.7%)

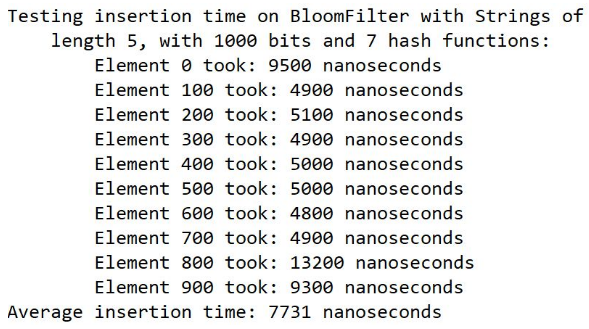




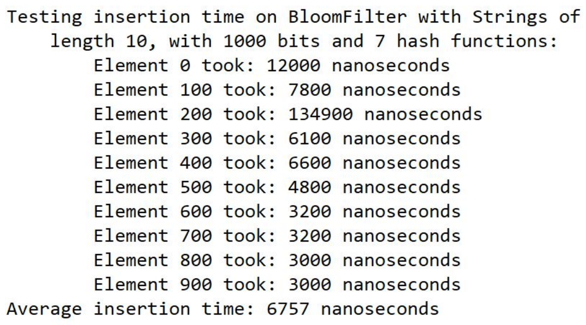
BloomFilter Performance

The BloomFilter should have a constant time for both `insert` and `mightContain`, independent of the element being passed. We tested this by using `System.nanoTime()` and inserting Strings of various lengths into BloomFilters of various sizes.

Comparing different elements (Strings of length 5 and 10):

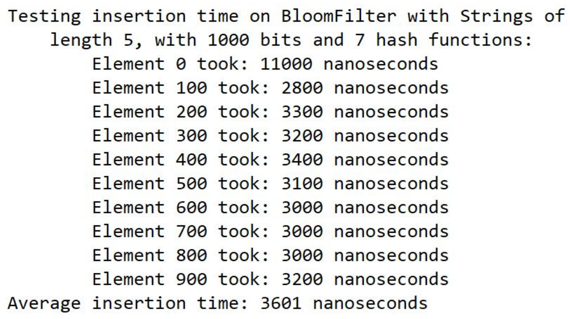


```
Testing insertion time on BloomFilter with Strings of
    length 5, with 1000 bits and 7 hash functions:
        Element 0 took: 9500 nanoseconds
        Element 100 took: 4900 nanoseconds
        Element 200 took: 5100 nanoseconds
        Element 300 took: 4900 nanoseconds
        Element 400 took: 5000 nanoseconds
        Element 500 took: 5000 nanoseconds
        Element 600 took: 4800 nanoseconds
        Element 700 took: 4900 nanoseconds
        Element 800 took: 13200 nanoseconds
        Element 900 took: 9300 nanoseconds
Average insertion time: 7731 nanoseconds
```

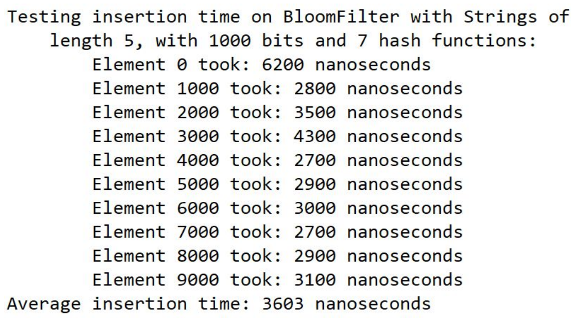


```
Testing insertion time on BloomFilter with Strings of
    length 10, with 1000 bits and 7 hash functions:
        Element 0 took: 12000 nanoseconds
        Element 100 took: 7800 nanoseconds
        Element 200 took: 134900 nanoseconds
        Element 300 took: 6100 nanoseconds
        Element 400 took: 6600 nanoseconds
        Element 500 took: 4800 nanoseconds
        Element 600 took: 3200 nanoseconds
        Element 700 took: 3200 nanoseconds
        Element 800 took: 3000 nanoseconds
        Element 900 took: 3000 nanoseconds
Average insertion time: 6757 nanoseconds
```

Comparing different numbers of elements:

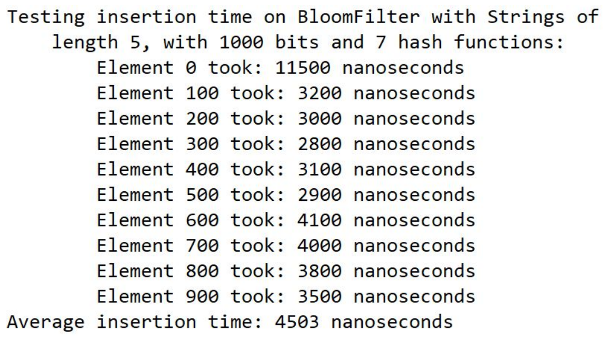


```
Testing insertion time on BloomFilter with Strings of
    length 5, with 1000 bits and 7 hash functions:
        Element 0 took: 11000 nanoseconds
        Element 100 took: 2800 nanoseconds
        Element 200 took: 3300 nanoseconds
        Element 300 took: 3200 nanoseconds
        Element 400 took: 3400 nanoseconds
        Element 500 took: 3100 nanoseconds
        Element 600 took: 3000 nanoseconds
        Element 700 took: 3000 nanoseconds
        Element 800 took: 3000 nanoseconds
        Element 900 took: 3200 nanoseconds
Average insertion time: 3601 nanoseconds
```

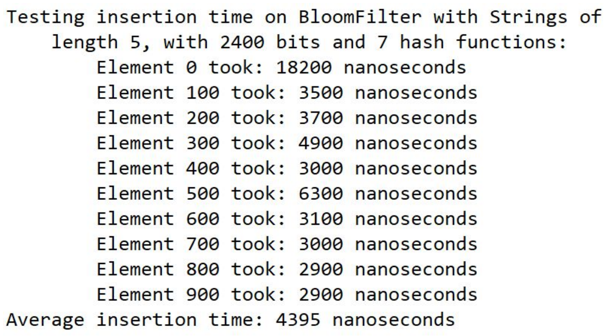


```
Testing insertion time on BloomFilter with Strings of
    length 5, with 1000 bits and 7 hash functions:
        Element 0 took: 6200 nanoseconds
        Element 1000 took: 2800 nanoseconds
        Element 2000 took: 3500 nanoseconds
        Element 3000 took: 4300 nanoseconds
        Element 4000 took: 2700 nanoseconds
        Element 5000 took: 2900 nanoseconds
        Element 6000 took: 3000 nanoseconds
        Element 7000 took: 2700 nanoseconds
        Element 8000 took: 2900 nanoseconds
        Element 9000 took: 3100 nanoseconds
Average insertion time: 3603 nanoseconds
```

Comparing different sized `byte[]`:

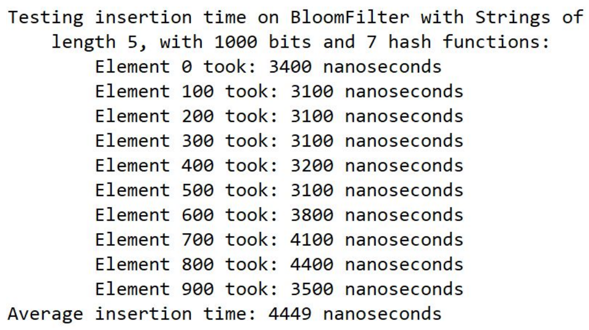


```
Testing insertion time on BloomFilter with Strings of
    length 5, with 1000 bits and 7 hash functions:
        Element 0 took: 11500 nanoseconds
        Element 100 took: 3200 nanoseconds
        Element 200 took: 3000 nanoseconds
        Element 300 took: 2800 nanoseconds
        Element 400 took: 3100 nanoseconds
        Element 500 took: 2900 nanoseconds
        Element 600 took: 4100 nanoseconds
        Element 700 took: 4000 nanoseconds
        Element 800 took: 3800 nanoseconds
        Element 900 took: 3500 nanoseconds
Average insertion time: 4503 nanoseconds
```

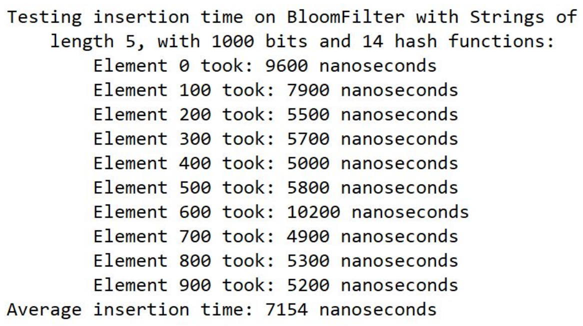


```
Testing insertion time on BloomFilter with Strings of
    length 5, with 2400 bits and 7 hash functions:
        Element 0 took: 18200 nanoseconds
        Element 100 took: 3500 nanoseconds
        Element 200 took: 3700 nanoseconds
        Element 300 took: 4900 nanoseconds
        Element 400 took: 3000 nanoseconds
        Element 500 took: 6300 nanoseconds
        Element 600 took: 3100 nanoseconds
        Element 700 took: 3000 nanoseconds
        Element 800 took: 2900 nanoseconds
        Element 900 took: 2900 nanoseconds
Average insertion time: 4395 nanoseconds
```

Comparing different numbers of hash functions:

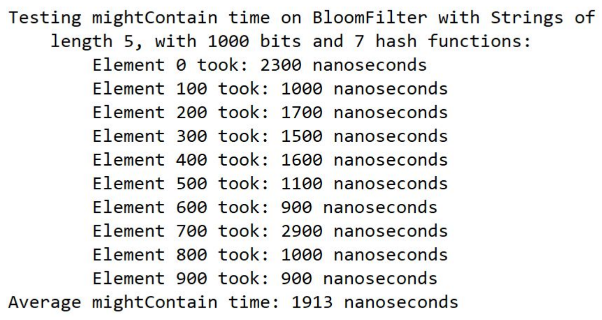


```
Testing insertion time on BloomFilter with Strings of
    length 5, with 1000 bits and 7 hash functions:
        Element 0 took: 3400 nanoseconds
        Element 100 took: 3100 nanoseconds
        Element 200 took: 3100 nanoseconds
        Element 300 took: 3100 nanoseconds
        Element 400 took: 3200 nanoseconds
        Element 500 took: 3100 nanoseconds
        Element 600 took: 3800 nanoseconds
        Element 700 took: 4100 nanoseconds
        Element 800 took: 4400 nanoseconds
        Element 900 took: 3500 nanoseconds
Average insertion time: 4449 nanoseconds
```

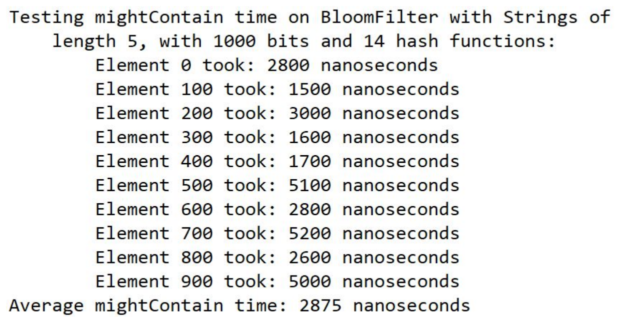


```
Testing insertion time on BloomFilter with Strings of
    length 5, with 1000 bits and 14 hash functions:
        Element 0 took: 9600 nanoseconds
        Element 100 took: 7900 nanoseconds
        Element 200 took: 5500 nanoseconds
        Element 300 took: 5700 nanoseconds
        Element 400 took: 5000 nanoseconds
        Element 500 took: 5800 nanoseconds
        Element 600 took: 10200 nanoseconds
        Element 700 took: 4900 nanoseconds
        Element 800 took: 5300 nanoseconds
        Element 900 took: 5200 nanoseconds
Average insertion time: 7154 nanoseconds
```

The main noticeable difference in time was from the number of hash functions, but as expected neither the objects nor the number of objects previously inserted had any noticeable effect. Results were similar with mightContain:



```
Testing mightContain time on BloomFilter with Strings of
    length 5, with 1000 bits and 7 hash functions:
        Element 0 took: 2300 nanoseconds
        Element 100 took: 1000 nanoseconds
        Element 200 took: 1700 nanoseconds
        Element 300 took: 1500 nanoseconds
        Element 400 took: 1600 nanoseconds
        Element 500 took: 1100 nanoseconds
        Element 600 took: 900 nanoseconds
        Element 700 took: 2900 nanoseconds
        Element 800 took: 1000 nanoseconds
        Element 900 took: 900 nanoseconds
Average mightContain time: 1913 nanoseconds
```



```
Testing mightContain time on BloomFilter with Strings of
    length 5, with 1000 bits and 14 hash functions:
        Element 0 took: 2800 nanoseconds
        Element 100 took: 1500 nanoseconds
        Element 200 took: 3000 nanoseconds
        Element 300 took: 1600 nanoseconds
        Element 400 took: 1700 nanoseconds
        Element 500 took: 5100 nanoseconds
        Element 600 took: 2800 nanoseconds
        Element 700 took: 5200 nanoseconds
        Element 800 took: 2600 nanoseconds
        Element 900 took: 5000 nanoseconds
Average mightContain time: 2875 nanoseconds
```

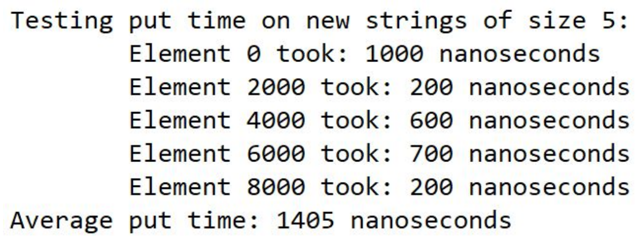
Hash Table Performance

For the HashTable, we wanted to be sure that the methods `put`, `get`, `remove`, `containsKey`, and `keySet` all operate under an amortized time complexity of O(1). To do this, we tested using `System.nanoTime()` to make sure we were getting consistent times as the HashTable grew.

The results were tested on a `HashTable<String, String>` with 2 initial buckets, where a randomly generated String of length 5 was inserted into the table as both the key and value.

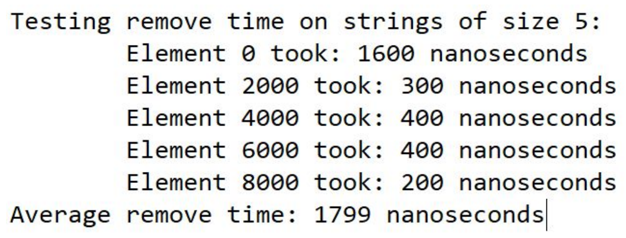
For timing the individual methods, we measured the time taken for each operation, looking at both a few times individually and the average time. Each result here is testing on 10,000 operations, printing out 5 results during testing. Some methods were tested using both results in and not in the HashTable to make sure the speed was constant regardless of key validity. The operations did not generally drastically change in speed between runs or as more (or fewer) elements were contained in the HashTable. Additionally, speeds were not obviously affected by the length of string used (though not explicitly shown in these summaries).

Put



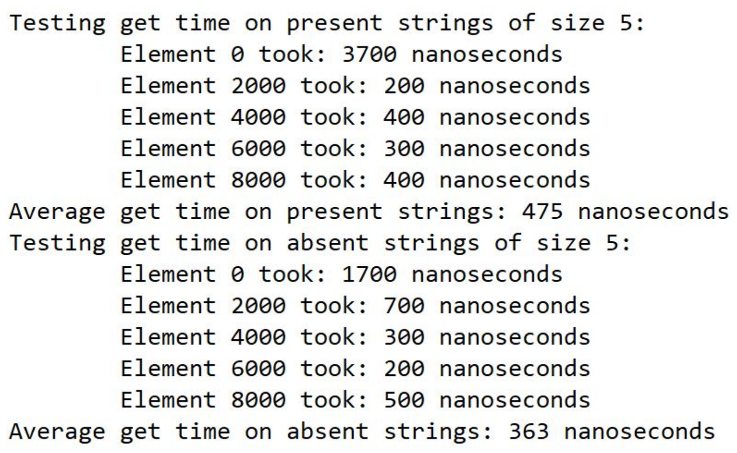
```
Testing put time on new strings of size 5:
        Element 0 took: 1000 nanoseconds
        Element 2000 took: 200 nanoseconds
        Element 4000 took: 600 nanoseconds
        Element 6000 took: 700 nanoseconds
        Element 8000 took: 200 nanoseconds
Average put time: 1405 nanoseconds
```

Remove

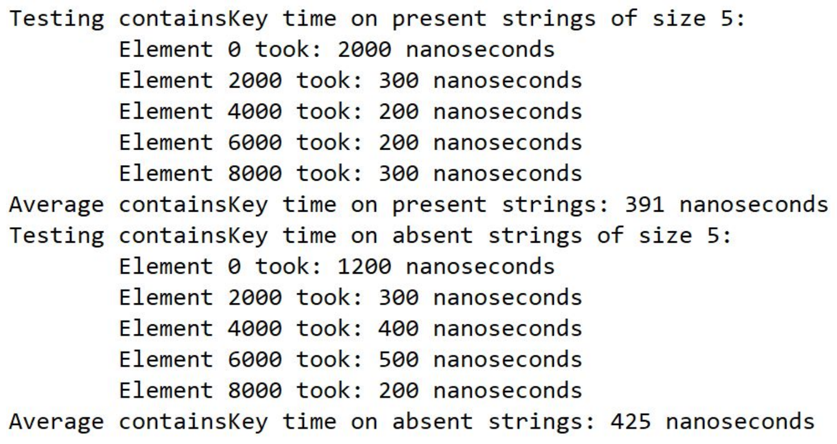


```
Testing remove time on strings of size 5:
        Element 0 took: 1600 nanoseconds
        Element 2000 took: 300 nanoseconds
        Element 4000 took: 400 nanoseconds
        Element 6000 took: 400 nanoseconds
        Element 8000 took: 200 nanoseconds
Average remove time: 1799 nanoseconds
```

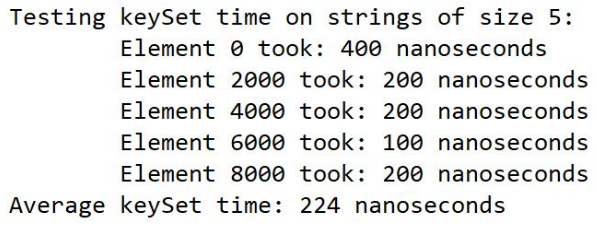
Get


```
Testing get time on present strings of size 5:
        Element 0 took: 3700 nanoseconds
        Element 2000 took: 200 nanoseconds
        Element 4000 took: 400 nanoseconds
        Element 6000 took: 300 nanoseconds
        Element 8000 took: 400 nanoseconds
Average get time on present strings: 475 nanoseconds
Testing get time on absent strings of size 5:
        Element 0 took: 1700 nanoseconds
        Element 2000 took: 700 nanoseconds
        Element 4000 took: 300 nanoseconds
        Element 6000 took: 200 nanoseconds
        Element 8000 took: 500 nanoseconds
Average get time on absent strings: 363 nanoseconds
```

ContainsKey


```
Testing containsKey time on present strings of size 5:
        Element 0 took: 2000 nanoseconds
        Element 2000 took: 300 nanoseconds
        Element 4000 took: 200 nanoseconds
        Element 6000 took: 200 nanoseconds
        Element 8000 took: 300 nanoseconds
Average containsKey time on present strings: 391 nanoseconds
Testing containsKey time on absent strings of size 5:
        Element 0 took: 1200 nanoseconds
        Element 2000 took: 300 nanoseconds
        Element 4000 took: 400 nanoseconds
        Element 6000 took: 500 nanoseconds
        Element 8000 took: 200 nanoseconds
Average containsKey time on absent strings: 425 nanoseconds
```

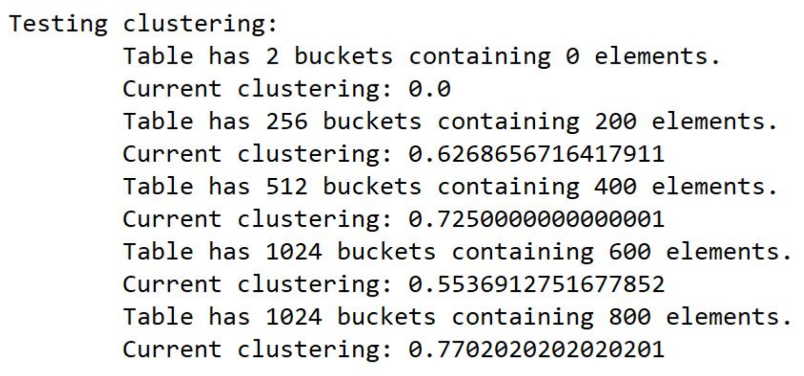
KeySet


```
Testing keySet time on strings of size 5:
        Element 0 took: 400 nanoseconds
        Element 2000 took: 200 nanoseconds
        Element 4000 took: 200 nanoseconds
        Element 6000 took: 100 nanoseconds
        Element 8000 took: 200 nanoseconds
Average keySet time: 224 nanoseconds
```

Hash Table Diffusion

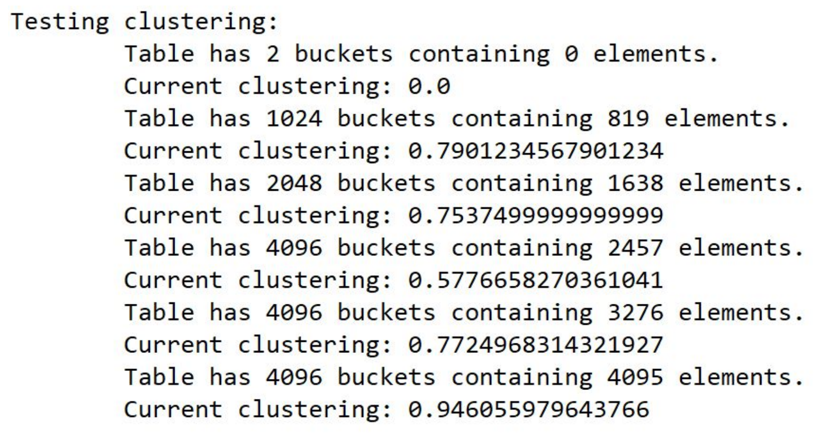
We measured the diffusion of the HashTable using both the clustering function described in the course notes and a direct bucket sampling. Both of these methods seemed to indicate we were getting a decent distribution of elements across the buckets.

Using the clustering measure, we observed the following results with a `HashTable<String, String>` with 2 initial buckets filled with random generated Strings of length 5:
Testing with 1000 elements:



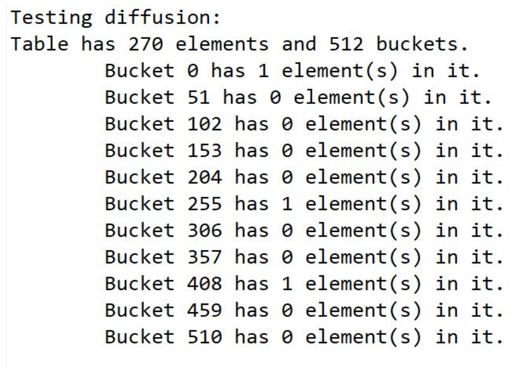
```
Testing clustering:
        Table has 2 buckets containing 0 elements.
        Current clustering: 0.0
        Table has 256 buckets containing 200 elements.
        Current clustering: 0.6268656716417911
        Table has 512 buckets containing 400 elements.
        Current clustering: 0.7250000000000001
        Table has 1024 buckets containing 600 elements.
        Current clustering: 0.5536912751677852
        Table has 1024 buckets containing 800 elements.
        Current clustering: 0.7702020202020201
```

Testing with 5000 elements:

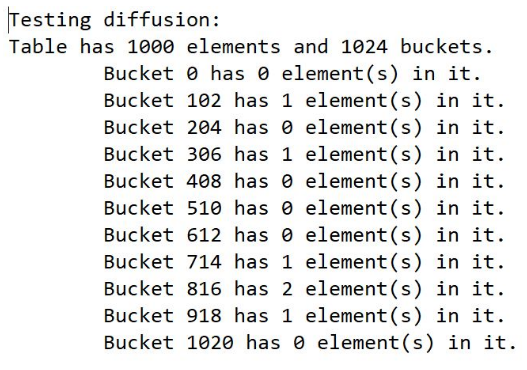


```
Testing clustering:
        Table has 2 buckets containing 0 elements.
        Current clustering: 0.0
        Table has 1024 buckets containing 819 elements.
        Current clustering: 0.7901234567901234
        Table has 2048 buckets containing 1638 elements.
        Current clustering: 0.7537499999999999
        Table has 4096 buckets containing 2457 elements.
        Current clustering: 0.5776658270361041
        Table has 4096 buckets containing 3276 elements.
        Current clustering: 0.7724968314321927
        Table has 4096 buckets containing 4095 elements.
        Current clustering: 0.946055979643766
```

We also checked bucket diffusion directly, creating a `HashTable<String, String>` with 2 buckets initially, filling it with randomly generated Strings of length 5, and then selecting some buckets to measure. As these results show, we don't seem to be getting large chains, but the diffusion could probably be improved with a more robust hash function, though perhaps at the expense of speed.



```
Testing diffusion:
Table has 270 elements and 512 buckets.
        Bucket 0 has 1 element(s) in it.
        Bucket 51 has 0 element(s) in it.
        Bucket 102 has 0 element(s) in it.
        Bucket 153 has 0 element(s) in it.
        Bucket 204 has 0 element(s) in it.
        Bucket 255 has 1 element(s) in it.
        Bucket 306 has 0 element(s) in it.
        Bucket 357 has 0 element(s) in it.
        Bucket 408 has 1 element(s) in it.
        Bucket 459 has 0 element(s) in it.
        Bucket 510 has 0 element(s) in it.
```



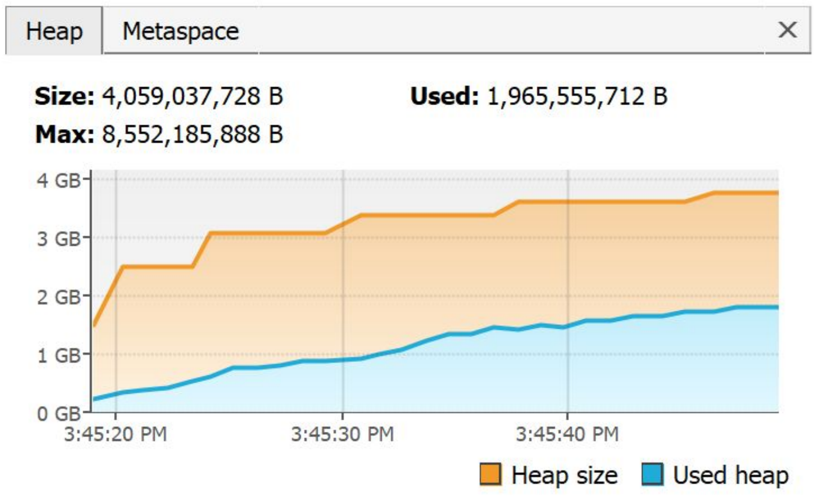
```
Testing diffusion:
Table has 1000 elements and 1024 buckets.
        Bucket 0 has 0 element(s) in it.
        Bucket 102 has 1 element(s) in it.
        Bucket 204 has 0 element(s) in it.
        Bucket 306 has 1 element(s) in it.
        Bucket 408 has 0 element(s) in it.
        Bucket 510 has 0 element(s) in it.
        Bucket 612 has 0 element(s) in it.
        Bucket 714 has 1 element(s) in it.
        Bucket 816 has 2 element(s) in it.
        Bucket 918 has 1 element(s) in it.
        Bucket 1020 has 0 element(s) in it.
```

```
Testing diffusion:
Table has 5000 elements and 8192 buckets.
        Bucket 0 has 0 element(s) in it.
        Bucket 819 has 1 element(s) in it.
        Bucket 1638 has 0 element(s) in it.
        Bucket 2457 has 1 element(s) in it.
        Bucket 3276 has 0 element(s) in it.
        Bucket 4095 has 0 element(s) in it.
        Bucket 4914 has 0 element(s) in it.
        Bucket 5733 has 0 element(s) in it.
        Bucket 6552 has 1 element(s) in it.
        Bucket 7371 has 0 element(s) in it.
        Bucket 8190 has 1 element(s) in it.
```

```
Testing diffusion:
Table has 8000 elements and 8192 buckets.
        Bucket 0 has 1 element(s) in it.
        Bucket 819 has 1 element(s) in it.
        Bucket 1638 has 0 element(s) in it.
        Bucket 2457 has 1 element(s) in it.
        Bucket 3276 has 1 element(s) in it.
        Bucket 4095 has 1 element(s) in it.
        Bucket 4914 has 0 element(s) in it.
        Bucket 5733 has 1 element(s) in it.
        Bucket 6552 has 1 element(s) in it.
        Bucket 7371 has 0 element(s) in it.
        Bucket 8190 has 1 element(s) in it.
```

Hash Table Memory

The HashTable's memory usage grew as the number of elements inserted into it grew, as would be expected (tested on the same HashTable as above, but with a total of somewhere above 50,000,000 elements inserted by the end of this graph):



Size: 4,059,037,728 B **Used:** 1,965,555,712 B
Max: 8,552,185,888 B

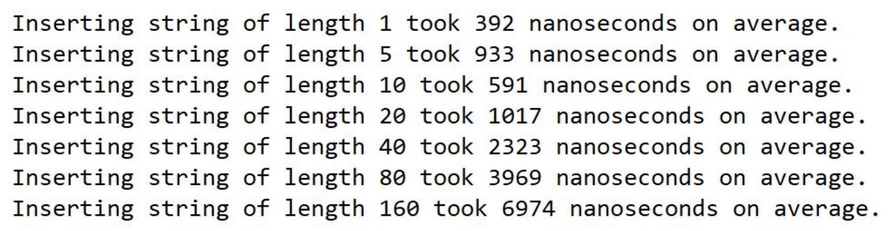
Trie Performance

For the Trie data structure, we wanted to ensure that the methods `delete`, `contain`, and `insert` all perform under a time complexity of $O(k)$ where k is the length of the string.

Insert

For testing `insert`, we tested that the time was $O(k)$ in a few different ways.

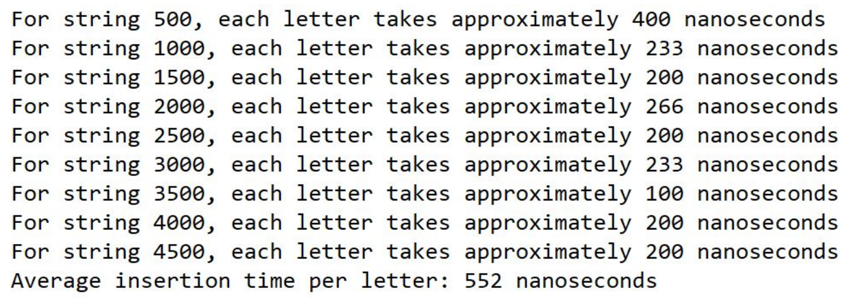
First, we tested that as the length of the word increased, the average time it took to insert the word also grew:



```
Inserting string of length 1 took 392 nanoseconds on average.
Inserting string of length 5 took 933 nanoseconds on average.
Inserting string of length 10 took 591 nanoseconds on average.
Inserting string of length 20 took 1017 nanoseconds on average.
Inserting string of length 40 took 2323 nanoseconds on average.
Inserting string of length 80 took 3969 nanoseconds on average.
Inserting string of length 160 took 6974 nanoseconds on average.
```

While there were generally some slightly off values (such as length 5 taking longer than length 10), the values do generally increase as the string length increases.

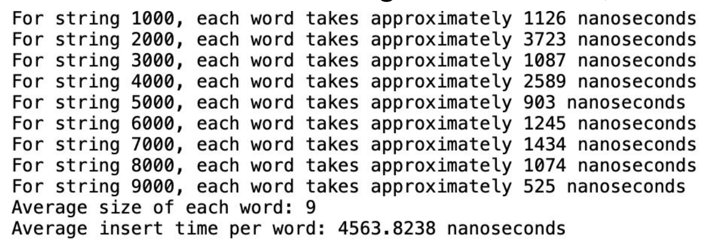
Secondly, we also tested with randomly generated strings to make sure that the average time per letter stayed approximately the same:



```
For string 500, each letter takes approximately 400 nanoseconds
For string 1000, each letter takes approximately 233 nanoseconds
For string 1500, each letter takes approximately 200 nanoseconds
For string 2000, each letter takes approximately 266 nanoseconds
For string 2500, each letter takes approximately 200 nanoseconds
For string 3000, each letter takes approximately 233 nanoseconds
For string 3500, each letter takes approximately 100 nanoseconds
For string 4000, each letter takes approximately 200 nanoseconds
For string 4500, each letter takes approximately 200 nanoseconds
Average insertion time per letter: 552 nanoseconds
```

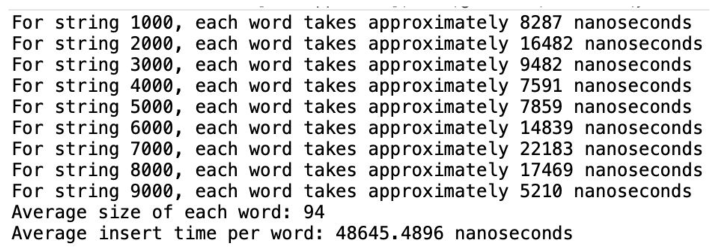
Then we combined approaches, generating random words and recording their length and the insertion time. While there is variation in the individual Strings based on the Trie's contents, the average insertion time grows proportionally to the length of the String, while the insertion time per letter stays fairly constant:

- Size of the word: ~9 → average insert time: ~4,500 nanoseconds (~507 a letter)



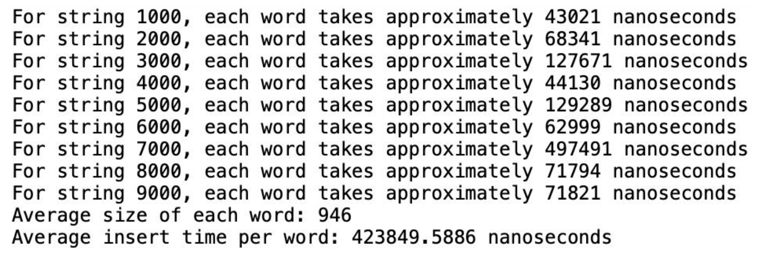
```
For string 1000, each word takes approximately 1126 nanoseconds
For string 2000, each word takes approximately 3723 nanoseconds
For string 3000, each word takes approximately 1087 nanoseconds
For string 4000, each word takes approximately 2589 nanoseconds
For string 5000, each word takes approximately 903 nanoseconds
For string 6000, each word takes approximately 1245 nanoseconds
For string 7000, each word takes approximately 1434 nanoseconds
For string 8000, each word takes approximately 1074 nanoseconds
For string 9000, each word takes approximately 525 nanoseconds
Average size of each word: 9
Average insert time per word: 4563.8238 nanoseconds
```

- Size of the word: ~95 → average insert time: ~48,645 nanoseconds (~517 a letter)



```
For string 1000, each word takes approximately 8287 nanoseconds
For string 2000, each word takes approximately 16482 nanoseconds
For string 3000, each word takes approximately 9482 nanoseconds
For string 4000, each word takes approximately 7591 nanoseconds
For string 5000, each word takes approximately 7859 nanoseconds
For string 6000, each word takes approximately 14839 nanoseconds
For string 7000, each word takes approximately 22183 nanoseconds
For string 8000, each word takes approximately 17469 nanoseconds
For string 9000, each word takes approximately 5210 nanoseconds
Average size of each word: 94
Average insert time per word: 48645.4896 nanoseconds
```

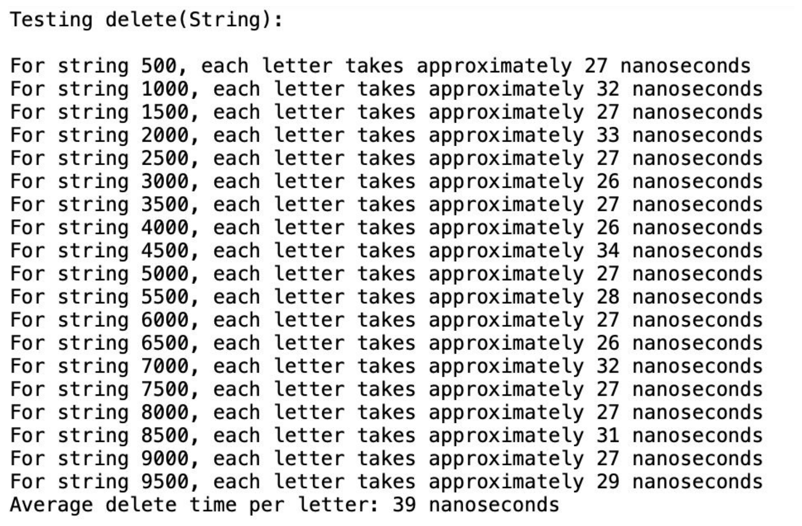
- Size of the word: ~950 → average insert time: ~423,850 nanoseconds (~448 a letter)



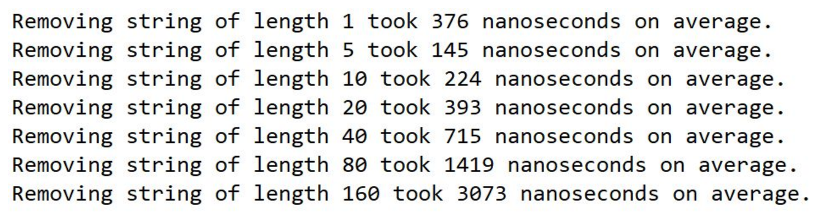
```
For string 1000, each word takes approximately 43021 nanoseconds
For string 2000, each word takes approximately 68341 nanoseconds
For string 3000, each word takes approximately 127671 nanoseconds
For string 4000, each word takes approximately 44130 nanoseconds
For string 5000, each word takes approximately 129289 nanoseconds
For string 6000, each word takes approximately 62999 nanoseconds
For string 7000, each word takes approximately 497491 nanoseconds
For string 8000, each word takes approximately 71794 nanoseconds
For string 9000, each word takes approximately 71821 nanoseconds
Average size of each word: 946
Average insert time per word: 423849.5886 nanoseconds
```

Delete

We took similar approaches with `delete`, testing the letter time to make sure it was fairly constant as well as the time relative to length.



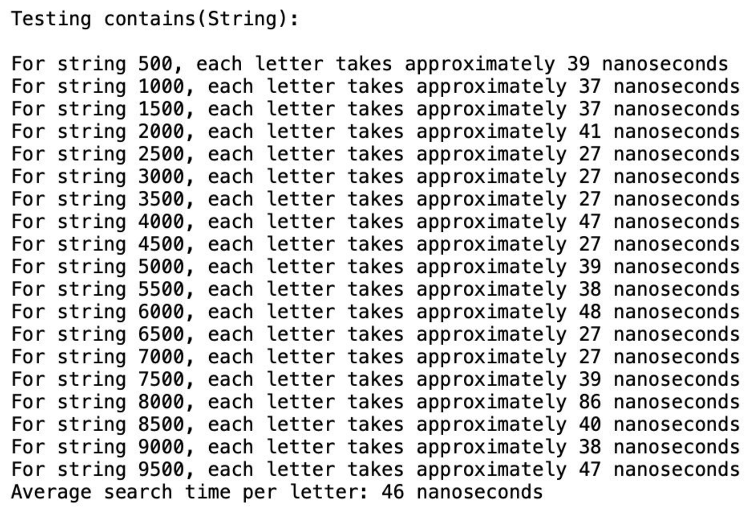
```
Testing delete(String):

For string 500, each letter takes approximately 27 nanoseconds
For string 1000, each letter takes approximately 32 nanoseconds
For string 1500, each letter takes approximately 27 nanoseconds
For string 2000, each letter takes approximately 33 nanoseconds
For string 2500, each letter takes approximately 27 nanoseconds
For string 3000, each letter takes approximately 26 nanoseconds
For string 3500, each letter takes approximately 27 nanoseconds
For string 4000, each letter takes approximately 26 nanoseconds
For string 4500, each letter takes approximately 34 nanoseconds
For string 5000, each letter takes approximately 27 nanoseconds
For string 5500, each letter takes approximately 28 nanoseconds
For string 6000, each letter takes approximately 27 nanoseconds
For string 6500, each letter takes approximately 26 nanoseconds
For string 7000, each letter takes approximately 32 nanoseconds
For string 7500, each letter takes approximately 27 nanoseconds
For string 8000, each letter takes approximately 27 nanoseconds
For string 8500, each letter takes approximately 31 nanoseconds
For string 9000, each letter takes approximately 27 nanoseconds
For string 9500, each letter takes approximately 29 nanoseconds
Average delete time per letter: 39 nanoseconds
```



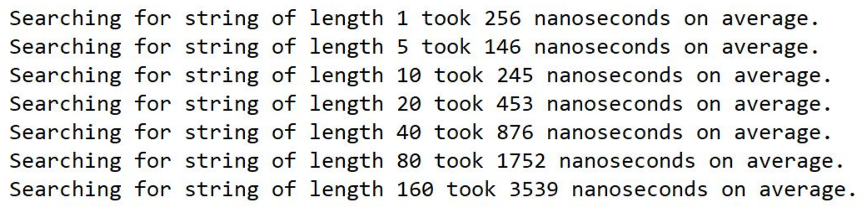
```
Removing string of length 1 took 376 nanoseconds on average.
Removing string of length 5 took 145 nanoseconds on average.
Removing string of length 10 took 224 nanoseconds on average.
Removing string of length 20 took 393 nanoseconds on average.
Removing string of length 40 took 715 nanoseconds on average.
Removing string of length 80 took 1419 nanoseconds on average.
Removing string of length 160 took 3073 nanoseconds on average.
```

Contains

We followed the same approach with `contains` as well:



```
Testing contains(String):

For string 500, each letter takes approximately 39 nanoseconds
For string 1000, each letter takes approximately 37 nanoseconds
For string 1500, each letter takes approximately 37 nanoseconds
For string 2000, each letter takes approximately 41 nanoseconds
For string 2500, each letter takes approximately 27 nanoseconds
For string 3000, each letter takes approximately 27 nanoseconds
For string 3500, each letter takes approximately 27 nanoseconds
For string 4000, each letter takes approximately 47 nanoseconds
For string 4500, each letter takes approximately 27 nanoseconds
For string 5000, each letter takes approximately 39 nanoseconds
For string 5500, each letter takes approximately 38 nanoseconds
For string 6000, each letter takes approximately 48 nanoseconds
For string 6500, each letter takes approximately 27 nanoseconds
For string 7000, each letter takes approximately 27 nanoseconds
For string 7500, each letter takes approximately 39 nanoseconds
For string 8000, each letter takes approximately 86 nanoseconds
For string 8500, each letter takes approximately 40 nanoseconds
For string 9000, each letter takes approximately 38 nanoseconds
For string 9500, each letter takes approximately 47 nanoseconds
Average search time per letter: 46 nanoseconds
```



```
Searching for string of length 1 took 256 nanoseconds on average.
Searching for string of length 5 took 146 nanoseconds on average.
Searching for string of length 10 took 245 nanoseconds on average.
Searching for string of length 20 took 453 nanoseconds on average.
Searching for string of length 40 took 876 nanoseconds on average.
Searching for string of length 80 took 1752 nanoseconds on average.
Searching for string of length 160 took 3539 nanoseconds on average.
```

For testing these strings, we made sure the strings were in the Trie so that it was actually doing a search, not simply returning false immediately upon realizing the word was not there.