Deciding on what Data Structure to use

Since the order of input is of no use, I decided to implement a hash table to store the randomly generated words. A basic implementation maps values to keys that are then stored in memory using a hash function. There are of course some problems that arise with this, mainly multiple keys that correspond to the same hash address, called a collision. Some simple solutions to collisions are to link the values together, using a linked list, or to find another hash address to store it in. Finding another hash address is more memory efficient however cuts into the lookup time since the hash needs to be recomputed. I decided to go with a closed addressing - also called chaining - hash table since I was looking for the most efficient look up times for my base case. This was then improved by changing to an open addressing hash table for my efficient version of a hash table. To accomplish this I used the robin hood method of rehashing.

Mathematical arguments

Traversing a data structure one entry at a time will yield a result at best on the first query but at worst on the last. This means that the average time for lookup in a common array will be O(n/2) and at worst O(n).

A binary search tree will on average yield a result when searching with O(log(n)) efficiency.
This of course varies depending on what search algorithm is used. However will almost always be better than the Array method since with large values it creates a logarithmic graph. Binary search trees are sorted arrays that can be split passed through relatively quickly, and are memory efficient. Yet creating them can take a while since every new entry to the tree has to be ordered, or sorted at the very end.

Better yet however is the hash table, which has an impressive O(1) efficiency (will take a constant amount of time to perform a function). This is because every value has a unique key that can be looked up at any moment. This is very memory intensive since every key has a unique address in memory. At least in a perfect hashTable, where there are no empty entries in the table. This is however near impossible to achieve since many hash functions will map more than one key to an address causing collisions. Depending on what method you use to deal with these collisions, it can cause the table to become O(log(k)) - even O(N) in some cases - where k is NOT the number of elements in the list but rather the number of collisions. This is still considered O(1) however, since the value of k should be negligible when compared to the number of elements. This is due to the probability of getting a repeat collision becoming near zero.

The probability of a collision = number of elements / size of table
So to find P(Collision) find the complement probability (P(of not having a repeat collision))
We can get the expected number of repeated collisions

- $\frac{19}{20} \times \frac{18}{20} \times \ldots \times \frac{1}{20}$

- At 10 repeat collisions the probability is 6.5%
- With Larger data sets this number tend to zero much sooner
- P(k Collisions) $=(\frac{n-1}{Capacity})^k$
-
- Can be generalised for large data sets

- P(k Collisions) = (n / Capacity)^k

- This is a variant of the Birthday Problem. A claim that after you have 70 people in a room, the odds of one of them having the same birthday as one other is 99%, but only reaches 100 on the 365 person.
- This shows why hash tables are considered O(1), since after a relatively small range, the probability tends to zero.

Robin Hood Hashing

Robin Hood hashing is a hashing algorithm implementation that reassigns hash collisions to the nearest unoccupied bucket, or to whichever bucket is closer to its original bucket than the new entry is to its original bucket. This is achieved by having each bucket keep track of its distance to initial bucket (DIB) and by linearly probing to find a suitable bucket. Once the DIB of new entry exceeds that of a stored bucket, swap the two and now repeat process until a bucket is reached that has no entry (is null). This ensures that multiple collisions are near each other in memory (locality) but also that there is no clustering effect due to an inefficient hashing algorithm. Lookup times are drastically reduced when dealing with high load factors, and can theoretically still be a decent choice past a load factor of 0.9. in comparison after a load factor of ~0.7 most hashing algorithms have exponentially increasing lookup times as well as insertion. Robin Hood hashing stays constant though out.

Testing - Conclusions

Both variants of my code were tested under the same environments. This meant running the same program to generate words, and using the same seed for said program. While Also keeping the test cases the same. Results collected are in the table below.

Comparison Table

	Time Taken to Complete Method (Best of 3) / ms		
	test_v1	**test_v2**	**Change**
Creation	1300	1300	0
Add (first million)	13,000	130	100x
Add (second million)	44000	220	200x
Remove()	170	67	2.5x
Count()	150	180	0.8x

As we can see the second variant was exponentially better in almost all ways except for a minor decrease in lookup times. The first variant demonstrates the importance of a good hashing algorithm. With 1 million words added about $3 * 10^5$ unique hashes were generated and with 2 million added words, only $5.5 * 10^5$g unique hashes were generated. This is most likely why the time taken to add words drastically increases with the amount of words added.. Given that collision resolution is handled by linked list chaining. This would mean that the majority of stored words are being kept on various unordered lists.

- As an example using the basic java hashCode() to generate hashes and then using the modulus operation to fit them into the array
  - I found that with 1 million randomly generated words there was on average a 75% collision rate.
  - This high number is due to the fact that both the random words being generated had some bias. As well as the hashing function.
    - This is proven true because the biggest linked list had a size of 4000.
  - When using the collision formula with n = 1 million, capacity = 1 million and k = 3982
    - You get 1.3 *10^-5 % of a size 4000 linked list.
    - Implying that the hash function was weighted in some way.

In the second variant, the hashing was done via Robin Hood hashing and thus is a lot more evenly spread out, it removes clustering, as well as having some locality (distance between two values that are similar) .It is the optimal solution since it is more memory efficient than a basic implementation of a hash table, yet has faster lookup times than ordered trees.

References

- https://stackoverflow.com/questions/10901752/what-is-the-significance-of-load-factor-in-hashmap
- https://tekmarathon.com/2013/03/11/creating-our-own-hashmap-in-java/
- http://bigocheatsheet.com/
- https://en.wikipedia.org/wiki/Hash_table
- https://en.wikipedia.org/wiki/Birthday_problem
- http://interactivepython.org/runestone/static/pythonds/SortSearch/Hashing.html
- https://abhishekchattopadhyay.wordpress.com/2014/06/14/concepts-of-hashing/
- https://en.wikipedia.org/wiki/Big_O_notation
- http://codecapsule.com/2013/11/11/robin-hood-hashing/
- https://www.programiz.com/java-programming/bitwise-operators
- https://www.javamex.com/tutorials/conversion/decimal_hexadecimal.shtml
- http://www.java2s.com/Tutorial/Java/0100__Class-Definition/AveryefficientjavahashalgorithmbasedontheBuzHashalgoritm.htm
- http://robl.co/implement-your-own-hashmap/