Report on Trie Structure for Auto-complete

1) Considering wordlist1000.txt

	Part 1	Part 2
Memory (Bytes)	653616	128768
No. of Nodes	3026	4024
Time to store in dictionary (s)	0.001496	0.000697
	0.000988	0.000822
	0.001011	0.000873
	0.000978	0.000758
	0.000997	0.000744
	0.000959	0.000797
	0.000981	0.000755
	0.001472	0.000782
	0.000957	0.000782
	0.000978	0.000753
Time to list suggestions (prefix : y)	0.000021	0.000027
(prefix : yo)	0.000018	0.000016
(prefix : a)	0.000172	0.000164
(prefix : ab)	0.000011	0.000013
(prefix : air) – only one suggestion	0.000004	0.000003
(prefix : agr)	0.000008	0.000007
(prefix : coll)	0.000007	0.000012
(prefix : econo)	0.000007	0.000010
(prefix : yt) – no suggestions	0.000002	0.000004
(prefix : thou)	0.000009	0.000015

When considering about the nodes in part 1, it has low amount of nodes compared to the part 2. But part 1 uses more memory than part 2.

Part 2 is more memory efficient.

Part 1 is implemented as a normal trie structure which takes more space to store. But in part 2, I have used a ternary tree structure which reduces the memory space usage. You can see that it reduces the memory space in a large amount. A ternary tree structure is very much space efficient.

In a trie structure, since our alphabet have 26 characters each node has 26 pointers plus one extra bit for whether or not the node encodes a word. But a ternary search tree stores 3 pointers per node, plus one character and one bit for whether or not the node encodes a word. So the ternary reduce the memory space to store the dictionary.

When comparing the insertion time of part 1 and part 2, most of the time part 2 has lower insertion time than the part 2.

But when comparing the suggestion time to a particular prefix part 1 has a lower time. So in the case of suggestion time, part 1 is more efficient.

Ternary tree take more time to traverse through the tree according to the prefix we enter. Because it has to take more decisions, it has to check whether the character is large, small or equal to the root character before it start the traversing.

But for a trie it do not have to check anything before searching. You just need to find the correct prefix characters and traverse through it to autocomplete.

Another failure of the part 1 trie structure is that it only can store 26 characters (In this case I have only added the characters in the alphabet). But if we want to store characters more than that, we have to expand the array that used store 26 characters. Then it will take more memory than the amount that I have shown here. So anyway this structure is going to be very much memory inefficient.

And again since the part 1, only store 26 characters it cannot distinguish between capital and simple. So for the simplicity, all the words read from the file are converted to simple letters before it is inserted into the data structure.

Although the part 2, structure can distinguish capital and simple for the simplicity all the words are inserted as mentioned above (after converting to simple).

2) Considering wordlist10000.txt

	Part 1	Part 2
Memory (Bytes)	24179	34178
No. of Nodes	5222664	1093696
Time to store in dictionary (s)	0.008894	0.009169
	0.009238	0.008772
	0.009143	0.009398
	0.009009	0.009587
	0.009101	0.008900
	0.008746	0.008910
	0.008835	0.008802
	0.009282	0.009325
	0.009256	0.008881
	0.009278	0.009352
Time to list suggestions (prefix : y)	0.000179	0.000103
(prefix : yo)	0.000024	0.000031
(prefix : a)	0.001526	0.001515
(prefix : ab)	0.000060	0.00059
(prefix : amy) – only one suggestion	0.000004	0.000008
(prefix : agr)	0.000099	0.000029
(prefix : coll)	0.000044	0.000049
(prefix : econo)	0.000016	0.000014
(prefix : yt) – no suggestions	0.000002	0.000006
(prefix : thou)	0.000016	0.000019

When using the 10000 wordlist, again the part 2 is very much space efficient than part 1. And other behaviors are similar to 1000 wordlist.

3) Considering wordlist70000.txt

	Part 1	Part 2
Memory (Bytes)	-	9572512
No. of Nodes	-	299141
Time to store in dictionary (s)	-	0.079132
	-	0.079978
	-	0.079320
	-	0.080837
	-	0.079730
	-	0.079613
	-	0.081260
	-	0.081117
	-	0.079849
	-	0.080187
Time to list suggestions (prefix : y)	-	0.000411
(prefix : yo)	-	0.000108
(prefix : a)	-	0.018952
(prefix : ab)	-	0.000823
(prefix : atrium) – only one suggestion	-	0.000008
(prefix : agr)	-	0.000101
(prefix : coll)	-	0.000236
(prefix : econo)	-	0.000027
(prefix : xx) – no suggestions	-	0.000007
(prefix : thou)	-	0.000045

In this case I could not check for part 1 because, as I mentioned above trie structure that I have implemented in part 1 can only store 26 characters in alphabet. But this word list has more commas, dashes etc. So this list cannot be inserted to the data structure.

But as the ternary tree do not limit to the 26 characters, it insert all the values into its data structure. So considering about that part 2, implementation is good compared to part 1.