Luca de Raad

Professor Du

CSS 143

3/14/2020

Design Document

For homework 7 problem 3 I created a trie to quickly autocomplete queries into a dictionary consisting of random words. For example, if my dictionary had the words "hello", "high", "seattle", "seatac", "see", "hollow", and "how" inputting “ho” would give me “hollow” and “how.” The code also does not check for capitalization as inputting “SEA” would return “seatac” and “seattle” with no capitalizations. My program would also pseudo work with multiple words but they would need to be in one string like “computer science” and if that was in my dictionary, inputting “science” would not work as my autocomplete starts checking at the beginning of an input and not where spaces occur.

The result of this trie with these specifications is that my code is extremely quick. In big O notation, my theoretical complexity becomes O of how many words my autocomplete finds times the average of the size of the words. This is due to the trie immediately discarding any words that do not fit the key cutting away at wasted comparisons. This is a massive improvement over a string comparison because a simple string comparison because even if you finished comparing two strings when they stopped aligning, you would have to repeatedly compare the initial characters of the string. An example of this would be autocompleting “sea” in the first paragraph’s example the program in a string comparison would have to compare the first character of every string and then it would compare the “sea” in “seattle” and “seatac” twice thereby increasing the complexity to O of n times the size of every string that would be autocompleted.

While doing tests I decided to see how fast my program completed tasks. Junit said it took around 3 milliseconds for my slowest comparison (4 letters) in a dictionary of size 7, just like in my example. I decided to create a larger dictionary of size 37 that would return 20 words in an autocomplete to compare the trie with a string comparison method to see if the trie was faster. The answer was yes, it only took around 50 nanoseconds for the trie compared to 80 nanoseconds for the string comparison. When testing further a trend emerged: the trie function consistently did faster comparatively to the string method especially when I increased the autocomplete string size (four letters taking 70 nanoseconds for the trie vs 150 for the string comparison). I used both nanosecond counters and Junit because nanosecond counts only when autocompleting the string while Junit is counting time even when I am building the dictionaries which could make the data hard to read when using comparatively small data sets where setting up the data takes longer than searching through it.

As of right now, creating a trie for speedy autocomplete is the best I can do. Without Wednesday’s lecture I probably would’ve tried a hashmap it would be no where near as quick as while getting a key would be fast, Comparing a linked list or an array within the hashmap would lead into a larger O(n) as I would still be comparing the same letters over again just like in a string comparison.