Algorithms and Data Structures: project

Victor Tuytte, Wout De Rijck

Group 43

1 Binary tree

1.1 What is the time and the memory complexity of the algorithm?

Assuming a tree is well balanced, then the height of the tree would be $\log(n)$ (average and best case). So space and time complexity would be $O(\log(n))$ for an insert. In a worst case scenario, if the tree is just one linked list (like a sorted array), then it would be $O(n)$ as worst case.

2 Red black tree

2.1 What is the time and memory complexity of the algorithm?

The time complexity is the sum of the insert function of the binary tree and the time complexity of the fixup. The tree is well balanced so the insert takes $O(\log(n))$ time and has a memory complexity of $O(n)$. The fixup takes at most $O(\log(n))$ time if a situation should arise where we need to fixup all the way to the root. So the total time complexity of the algorithm is $O(\log(n))$ and the memory complexity is $O(n)$.

2.2 What are some alternatives that you could use?

We could use a hash table (dictionary) because this data structure can also hold the same information. We only need to search, insert and update values of a given key and a hash table can do all these things in $O(1)$ which is better than a red black tree that needs $O(\log(n))$ time. If we really want a binary tree we could also use a randomly built binary tree as another balanced binary tree that tries to avoid the worst case of $O(n)$ time complexity. A randomly built binary still has a chance of being unbalanced so it might not be the best option.

3 BK-tree

3.1 How much storage space is required for the dynamic programming version of the Levenshtein distance?

This is solved by making a [n x m] matrix ($n = stringlength_1 + 1$; $m = stringlength_2 + 1$). So we get a memory complexity of $O(mn)$. This could be reduced to $O(m)$ or $O(n)$ by making a [n x 2] or [m x 2] matrix and shifting the values each iteration. This approach would add some time and because strings are usually not that big there should be no need to lower the memory complexity. So we kept the $O(mn)$ memory complexity.

3.2 What is the time and memory complexity of the BK-tree algorithm?

The height of a BK Tree depends on the average amount of children a single node has, if we take this to be k on average we get a height of $O(\log_k(n))$ where n is the total amount of items. So in the average case, the depth of the BK Tree will be $\log(n)$. If s_1 is the average length of a word in the dictionary, s_2 the length of the word you are searching and the TOL the tolerance; the time complexity would be $O(s_1 s_2 \log(n))$. Where we calculate the time complexity as TOL times the depth $O(\log(n))$ times the distance $O(s_1 s_2)$. Assuming we take TOL to be small (TOL = 2) we can ignore it for the time complexity. The memory complexity is $O(n)$ as we need to store almost every word that is inserted.

3.3 What is the time and memory complexity of comparing the query word with each word in the dictionary?

The time complexity is $O(s_1 s_2 n)$, with n the dictionary size, s_1 average length of a word in the dictionary and s_2 the length of the query word. This can easily be understood because we need to check n words and comparing 2 words takes $O(s_1 s_2)$ time. The memory complexity is equal to $O(n)$ because you need a place to store the results and n comparisons give n results.

3.4 When many items are added to the tree, it becomes slow. How would you solve this?

BLANK

4 Ranking

4.1 Which data structure did you use to store intermediate results in your ranker? What is the reason?

We stored intermediate results in a dictionary (hash table) because it's easy to check whether or not an element is already in it ($O(1)$ time complexity). In a list this would be $O(n)$. We can also easily fetch all the given keys in $O(1)$ time average case. This makes this python structure ideal for the intended purposes.