mydictionary: a dictionary lookup program based on binary search

Máté Hekfusz and Li Cheng

*Mydictionary* is a program written in C++ that, when given a dictionary file, is capable of looking up full words, prefixed words, and words with a wildcard character, and returning them in a speedy manner. The program is built on the binary search algorithm, which runs in log(n) time and can thus find a word in a 460,000-entry dictionary in at most 19 comparisons, a massive improvement over linear time. This write-up will outline and explain the algorithmic and design choices the creators have made for the program.

File structure

*Mydictionary* is composed of three files: mydictionary.cpp and mydictSearches.cpp contain driver and search functionality, respectively, while mydictSearches.h is the header file for the functions. A makefile is also included; it builds an executable called mydictionary in the program folder. The program is designed to be executed with two parameters: a dictionary file to read and the max number of outputs it should display for any given search.

Main functionality

The task of mydictionary.cpp is to parse the user input, both on program execution and during the actual lookups and decide on the type of search to be performed. After taking the parameters, the dictionary file’s contents are stored in a vector. A vector was chosen for its greater flexibility and ease of use when passing to functions when compared to arrays.

After user input, the function analyses the input string for two special characters (\* and ?) that signify different kinds of lookup. For \* or prefix search, knowing that it is present is enough, since it will always be at the end of the string. The input string is sliced to cut off the \* and sent to the prefix search function. For ? or wildcard, the position of the ? is also determined with the find() function and passed to the wildcard search function as an integer. If neither of these special characters were found, the program sends the input to the full-word search.

Search functionality

Full-word search

Full-word search is the simplest of the three functions: it just performs a binary search through the dictionary to check if the input is present. Binary search works by first comparing the input to the middle element of the dictionary, and since the dictionary is essentially a sorted array, it drops whichever half the input cannot be in. Then it repeats this halving procedure until it either finds the word or finishes looking through the dictionary.

Prefix search

Prefix search is made up of two parts: finding an element that matches the prefix, then finding *every* element that matches it. The first part is accomplished with binary search; the input is compared to equal-length substrings of dictionary entries until a match is found. For example, if the input was ‘expl\*’, the program will look for words whose first four letters are ‘expl’. The second part is, just like binary search, built on the fundamental attribute of dictionaries that they are sorted. Since they are sorted, every entry that starts with the same prefix will be in *one, continuous list* within the dictionary. The algorithm’s task is thus finding how far the list extends within the dictionary.

Since binary search only compares the input to the n/(2^k)th elements, it is very likely that it will find a random match, instead of the first or the last matching entry. This necessitates a two-way search. For this, a while loop is utilised, displaying every element as long as the prefix matches (or until the max output count is reached). Once either edge of the match list is reached, the iterator jumps back to the original match (stored earlier) and repeats the process in the other direction. This ensures that every single prefix match is found while only adding O(k) complexity, where k is the length of the match list.

Wildcard search

Wildcard search builds mainly on the prefix functionality. In the first part, the input is compared to substrings that match the input length until the ?. Once a match is found, it begins the same two-way search through the matching list. There are two differences. First, there is an extra error check before any further comparison, which checks whether the compared entry’s length is less than the position of the ?. Any entry for which that is true is skipped over, since not only can they not be matches (being necessarily shorter than the input string), they also throw out-of-range errors at the next comparison. After this sanity check, the key comparison is performed: whether the *second half* of the entry (after the ?) also matches the second half of the input. The appropriate substring for this is acquired by subtracting the position of the ? from the length of both the input and the dictionary entry, which results in a substring that is the second half of the string after the ?. The comparison count for this search is also calculated differently: instead of stopping the counter upon finding the first match, it keeps incrementing through the two-way search to give the comparison count for every match. Otherwise, this also adds O(k) complexity.