Andy Fleischer

Prof. Karla Fant

CS163

14 May 2021

Program 3 Efficiency Writeup - Hash Table

This program was required to map a term to a description and list of related websites. For this table abstract data type, we used a hash table data structure where data is hashed into an index in an array of linear linked lists (to deal with chaining). This data structure worked well and was very efficient for almost every function. It was much more efficient than a linear linked list, doubly linked list, circular linked list, or linked list of arrays, since in any of those you will need some sort of traversal. The data is also not inherently circular or two-way, so I would not switch the data structure used.

As mentioned above, the hash table was efficient for most of the searches as long as there were not too many collisions. With no collisions, any function involving searching by key (add a term, remove by term, display by term, etc) was effectively O(1) efficiency because of the direct member access of the array. Even with a few collisions, it was still far faster than any other data structure. What was not efficient in the hash table was any function requiring full table traversal, like delete all (destructor), display all (a function that I added), and remove by link. Of course, any data structure will require full traversal for delete or display all, but removing by link was less efficient than it could have been with another data structure. This is because hash tables are very efficient when searching by key, but very inefficient when searching by value.

If I had more time to work on this project, there is a lot I would like to do. As always, I could use more time to further optimize and modularize the code, and create a cleaner menu interface. A cleaner UI would especially benefit this program because my menu had many different functions since I had to add a couple for debugging purposes. Also, I could spend some time to create more functions, like the display all that I added. I might add a function to check the number of collisions in the table, check the number of empty slots in the table, and so on. An interesting function I also thought about was a resize function to either automatically (as a private function, more likely) or on user demand (as a public function) resize the hash table and copy in all the data rehashed to the new size. The main issue is this would be another very slow function, and would require a lot of code to manage. Finally, specific to this program, I could spend some more time creating a large external data file to get some more accurate data on my hash functions (which I talk about below). From there, I could try to optimize my hash a little more too, seeing how spreads data on those larger data files.

As for the hash functions, my first one simply added together the ascii values of every character in the term and modding by table size. There are many issues with this, such as keys that are anagrams will always collide. On terms that can often be pretty short, this is a key (no pun intended) issue. Also, with these short terms, there is a chance of unfilled table slots if the size of the table was pretty large (100-500). That is why my second function also multiplies the letter value (A=1, B=2, etc.) of each character by some number (usually a power of 2, I used 16) to the power of the character's index. This hash creates much larger numbers that are unique to the certain positioning of letters in the word. Since the resulting number could result in integer overflow on fairly large keys (about 8 letters with my hash), I applied Horner's rule to factor out the powers and applied the modulo operator after factoring to maintain a small number (Horner's rule also increases runtime efficiency).

With the second hash function, collisions were minimal. I noticed that with a table size of a power of 2 (8), the second hash had more collisions with my external data file than if the size was prime (7). There were 9 and 8 respectively, and the table size of 8 even had 7 nodes on one of the indices. This is because the sum of my second hash will cluster around multiples of 16, the base that I chose. Thus, once modded by table size, the data will cluster around common multiples of 16 and the table size. This is not a problem if the table is prime to reduce clustering, but is very problematic when the table is a power of 2. Because of the many collisions, all functions of the hash table lose performance.