CISC 235 Assignment 3

Jacob Roe - 20351389

March 12, 2024

I confirm that this submission is my own work and is consistent with the Queen's regulations on Academic Integrity.

Experiments

Part 1

For part one, to convert the strings to an integer, I took the approach of iterating over each character of the string and using pythons ord() function which converts characters to their Unicode numerical value, then summing these values multiplied by a constant c - which is determined by the length of the string being hashed. I found that this had results that worked for the requirements of the assignment.

Part 2

For part two, my implementation of the hash table, used the hashing method from part 1 as the hash function along with the collision resolution methods discussed in class. To actually test these methods, I generated 2000 random strings, 1000 length 7, and 1000 length 8, and inserted them all into the table using the hashign methods. For the different values of c1, and c2 I chose the there combinations:

1. c1 = 1, c2 = 1
2. c1 = 2, c2 = 0.5
3. c1 = 3, c2 = 0.33

For each of the combinations of c1, and c2 I would start by testing table sized of 2500, and work downwards by intervals of either 25 or 50 in order to find the lowest possible table size that still provides the required performance averages. From this method of testing, I found that:

1. For c1 = 1, c2 = 1: I found that a table size of 2125 works well and yields an average number of comparisons of roughly 2.5
2. For c1 = 2, c2 = 0.5: I found that a table size of 2075 works well and yields an average number of comparisons of roughly 3.2
3. For c1 = 3, c2 = 0.33: I found that a table size of 2025 works well and yields an average number of comparisons of roughly 3.8

In each of the cases above, these were the lowest table sizes that would consistently insert all values, and maintain the average performance.

Part 3

For part three, my approach was similar to that of part 2, with the difference being that it uses double hashing apposed to quadratic probing, I chose three different hash functions, and tried different combinations of them to find minimum table sizes for the respective combinations that fell within the required performance. The three hash functions used were:

1. Hash_1 : The same hash from part 1, simple sum the Unicode values of the characters in the string while multiplying by a constant c - that is the length of the string being hashed.
2. Hash_2 : This hash works by summing the square of all of the Unicode values of the characters in the string.
3. Hash_3 : This hash is the square of the sum of the square roots of all of the Unicode values of the characters in the string.

I then took three combinations of these hash functions, and applied them to the double hashing methods described in class. The combinations are as follows:

1. h' = Hash_1, h" = Hash_2
2. h' = Hash_2, h" = Hash_3
3. h' = Hash_1, h" = Hash_3

With these three combinations, I took the same testing approach as part 2 in order to find the lowest table size that fits within the required performance, the results were as follows:

1. For h' = Hash_1, h" = Hash_2 : I found that a table size of 2500 was needed to consistently insert all values, and maintain the required performance, this game me an average number of comparisons of roughly 3.
2. For h' = Hash_2, h" = Hash_3 : I found that a table size of 2050 worked, and gave an average number of comparisons of roughly 2.85
3. For h' = Hash_1, h" = Hash_3 : I found that a table size of 2050 also worked, and gave an average number of comparisons of roughly 2.8

Part 4

Overall, my findings show that quadratic probing can provide smaller table sizes within a required performance range, however this mainly depends on the hashing functions chosen for both the quadratic probing approach, and the double hashing approach.