Lab 4 – Option B
Mark Williams
CS 2302

Introduction: In this lab, I improve the running time of the program created in Lab 3 by using a hash table instead of a binary tree. The hash table solves collisions with chaining. The user has the option to choose between three different hash functions when creating the table, and through some test runs I determine which of the hash functions is the most optimal.

Design Implementation: In the previous lab, we had to create a data structure, english_words, using a binary tree. To retrieve an element from this tree, an average of about 18 comparisons had to be made. The main purpose of this lab is to reduce the number of comparisons needed to find an element. To do so, I create the data structure english_words with a hash table instead of with a binary tree. The hash table's size is 400,000 and I chose this size because I have the memory space to support it, and it is less than the number of words in words.txt. I also create three different hash functions for the table. At this point, it is important to note that to hash a string I first convert it to a base 10 number. I do this by considering a string as a base 26 number and then from there I convert it to a base 10 number using rudimentary conversion methods. The first hash function is a modulo hash. The second function is a multiplicative hash. In this function, the modulo hash of a given string is first found. That hash is then multiplied by an arbitrary integer and a number related to the mapping of a character in the given string is added to the hash. This occurs for every character in the given string. This hash function is like the Zybooks implementation of the multiplicative hash. The third hash function is a random hash. In this function, a random number is chosen for each valid string based on the size of the table. The last two functions I create for this program compute the load factor of the hash table and record the average number of comparisons required to access an element in the hash table respectively.

Results: To test my hash functions, I record the average number of comparisons required to access an element in the hash table for each function. To do this, I run the program ten times for each hash function. The average of all the runs is shown in this table:

Hash Function	Modulo	Multiplicative	Random
Average Comparisons	3.05	1.55	1.54

Below is a table that shows the time complexity of creating a hash table for each hash function using the input file words.txt over ten iterations per hash function.

Hash Function	Modulo	Multiplicative	Random
Running Time (in seconds)	8.76	8.67	8.08

To show that the program can create the table for a given hash function, compute the load factor of a table, and show the user the number of comparisons needed to find an element in words.txt, I include three sample outputs for each hash function below.

```
What type of hash function do you want to use?
Enter 1 for modulo, 2 for multiplicative, 3 for random.
1
The load factor of this hash_table is 1.0408525
The average number of comparisons required to perform a successful retrieve operation are: 2.5511371688111426
```

```
What type of hash function do you want to use?
Enter 1 for modulo, 2 for multiplicative, 3 for random.
2
The load factor of this hash_table is 1.0407325
The average number of comparisons required to perform a successful retrieve operation are: 1.045984438844756
```

```
What type of hash function do you want to use?
Enter 1 for modulo, 2 for multiplicative, 3 for random.
3
The load factor of this hash_table is 1.0408525
The average number of comparisons required to perform a successful retrieve operation are: 1.0407694654141677
```

Overall, I found that the random hash had both the best time complexity and best distribution. It is important to note that all hash functions yielded a table that required significantly less comparisons than the binary tree used in Lab 3B.

Conclusion: In this project, I learned how to practically implement a hash table for a real-world problem. I also learned how to create hash functions that make hash tables faster and more balanced.

Appendix: Code for this lab is located at: https://github.com/mawilliams7/lab4

"I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provide inappropriate assistance to any student in the class."