**Introduction**

This report looks at how three different methods of storing data in a hash table—Chaining, Double Hashing, and Linear Probing—perform when the hash table gets fuller. We are focusing on how long it takes to add data (insert time) and find data (search time) as the hash table fills up (load factor).

**HashTableC(Chaining)**

It works like this, creates a table of lists, or chains, for data storage during initialization.

Hash Function: Computes the sum of the characters in the key's ASCIl values and uses modulus to determine the index.

Insertion: Adds the key-value pair to the list at that index after calculating the index.

Search: Uses the calculated index to search the list for the key.

**HashTableDH(Double Hashing)**

Initialization: Creates a table and a keys array, both starting empty. Hash Functions: Primary Hash: Uses a polynomial method on the key. Secondary Hash: Adds ASCIl values of the key's characters and uses modulus to get a step size. Insertion: Checks if the table is too full and resizes if needed. Calculates the primary index and step size; uses these to find the right spot. Search: Uses the primary index and step size to find the key.

**HashTableLP(Linear Probing)**

Initialization: Creates a table and a keys array, both starting empty. Hash Function: Adds up ASCIl values of the key's characters and uses modulus to get the index. Insertion: Checks if the table is too full and resizes if needed. Calculates the index and finds the right spot using linear probing (sequentially checks next slots). Search: Uses the index and linear probing to find the key. Resizing: Doubles the table size and reinserts all key-value pairs. Insertion: Assesses if the table is overflowing and adjusts its size accordingly. determines the step size and primary index and uses these to determine the optimal location. Search: To locate the key, use the step size and primary index.

**Metholodgy**

The function *read\_names\_from\_file(file\_path)* reads names from a file and stores them in a list. insert\_names *(hash\_table, names)*: Measures the time to insert each name into the hash table. *search\_names(hash\_table, names)*: Measures the time to search for each name in the hash table. *calculate\_load\_factor(hash\_table)*: Calculates the load factor (number of elements divided by table size) of the hash table. Main function: Reads names from "english\_words.txt". Defines a range of sizes for testing. Initializes hash tables of each type for each size. Measures insertion times, search times, and load factors for each hash table type. Plots the results.

**Insert Time vs. Load Factor**

Chaining Insert Times (Blue): The time it takes to add data stays about the same no matter how full the table gets. This shows that chaining works well even as the table fills up. Double Hashing Insert Times (Orange): Adding data with double hashing is also quick and stable across different levels of fullness. Linear Probing Insert Times (Green): The time it takes to add data increases a lot as the table gets fuller. This means linear probing slows down when the table is more filled.

**Search Time vs. Load Factor**

Chaining Search Times (Blue):The time it takes to find data remains low and steady, showing that chaining is efficient at finding data regardless of how full the table is. Double Hashing Search Times (Orange): Finding data with double hashing is consistently fast, no matter the load factor. Linear Probing Search Times (Green): The time it takes to find data increases significantly as the table gets fuller, similar to the insert times. This shows that linear probing gets less efficient with higher load factors.

**Conclusion**

To sum up, chaining and double hashing are better at keeping insert and search times short, even as the hash table gets fuller. Linear probing, on the other hand, becomes much slower as the table fills up.

