### HT - Criocerinae

Arun Chintalapati, Arav Chheda, Marissa Gardner, Logan Mentzer

### Hashing With Open Addressing: Report

### Background

### Open addressing is defined as a method used to handle collisions and store symbol tables. All elements are stored in hash tables, therefore the tables must be equal or greater than the number of keys. Open addressing was introduced through practice in 1954, when Amdahl, Boehme, and Samuel used it for IMB 701 within an assembly program. Despite other references to the open addressing technique, before 1965, the timing characteristics had never been solved. On 7/22/1963, Don Knuth proposed a solution, which was then proved on 5/20/1965.

### Hashing with open addressing is a method of building a hash table. With this method, when elements are placed in the hash table, if the index the element hashes to is already taken, you place the element in the first null index. When searching for an element, you first check the index it would originally hash to. If it is not there, you continue checking the elements to the right until you find it. Despite the simplicity of this approach, it is surprisingly efficient and effectively builds hash tables. These hash tables normally double in size when they reach a certain capacity in order to have enough null indexes to store additional elements.

### The task with this project was to empirically determine just how efficient this approach is at filling, retrieving values from, and deleting elements from a hash table. We also wanted to determine the optimal resizing factor for the hash table to obtain the most efficient performance.

### Implementation

We modified the TaleOfTwoCitiesExtractor class to read the English dictionary file given to us. The LinearProbingHashST class was modified to include a new constructor that takes in a value for the multiplicative factor. Additionally, this constructor automatically sets the symbol table’s size to be 321165 \* the multiplicative factor. If not specified in the constructor, the default multiplicative factor is 2 and the default size is 16. This multiplicative value is also used for resizing when values are deleted from the symbol table. We made a separate get and fill method for both question 4 and question 5.

#### Problem 4

* fill2: This method takes in a parameter for the number of words to add to the symbol table. It utilizes the EnglishDictionaryExtractor next method and the LinearProbingHashST put method within a for-loop to input values from the English dictionary. The method uses the Stopwatch class to time how long this process takes.
* testGet2: This method takes in a parameter for the number of words to put in the symbol table and goes through the same process as the fill2 method to create a symbol table with the given amount of words. The method then uses the get method in the LinearProbingHashST class to get all the words. This process is timed using the Stopwatch class.

#### 

#### Problem 5

* fill: This method takes in a parameter that determines the multiplicative factor. This method utilizes the same process as the fill2 method to time the creation of the symbol table. This method, however, creates a LinearProbingHashST object with the given multiplicative factor rather than using the default value of 2. Additionally, this method always adds the entire English dictionary to the symbol table rather than taking in a parameter.
* testGet: This method takes in a parameter that determines the multiplicative factor and goes through the same process as the fill method to create a symbol table with the given multiplicative factor. The method then uses the get method in the LinearProbingHashST class to get all the words. This process is timed using the Stopwatch class.
* testDelete: This method takes in a parameter that determines the multiplicative factor and goes through the same process as the fill method to create a symbol table with the given multiplicative factor. The method then uses the delete method in the LinearProbingHashST class to delete all the words. This process is timed using the Stopwatch class.

### Results/Data

In order to get an accurate representation of the times for Problem 4, the code was run multiple times (5 times), as the get and fill times depend on the hardware of the computer, and where the keys are hashed during each run. The different runs are averaged out in the tables below. These times may also not be the same on every computer as they are dependent on the hardware of the computer.

|  |  |
| --- | --- |
|  |  |

As similar dependencies on hardware and the hashing function exist in the results for Problem 5, the same process of averaging the numbers from multiple runs was used. The different runs are averaged out in the tables below.

|  |  |
| --- | --- |
|  |  |



### Analysis

#### Problem 4

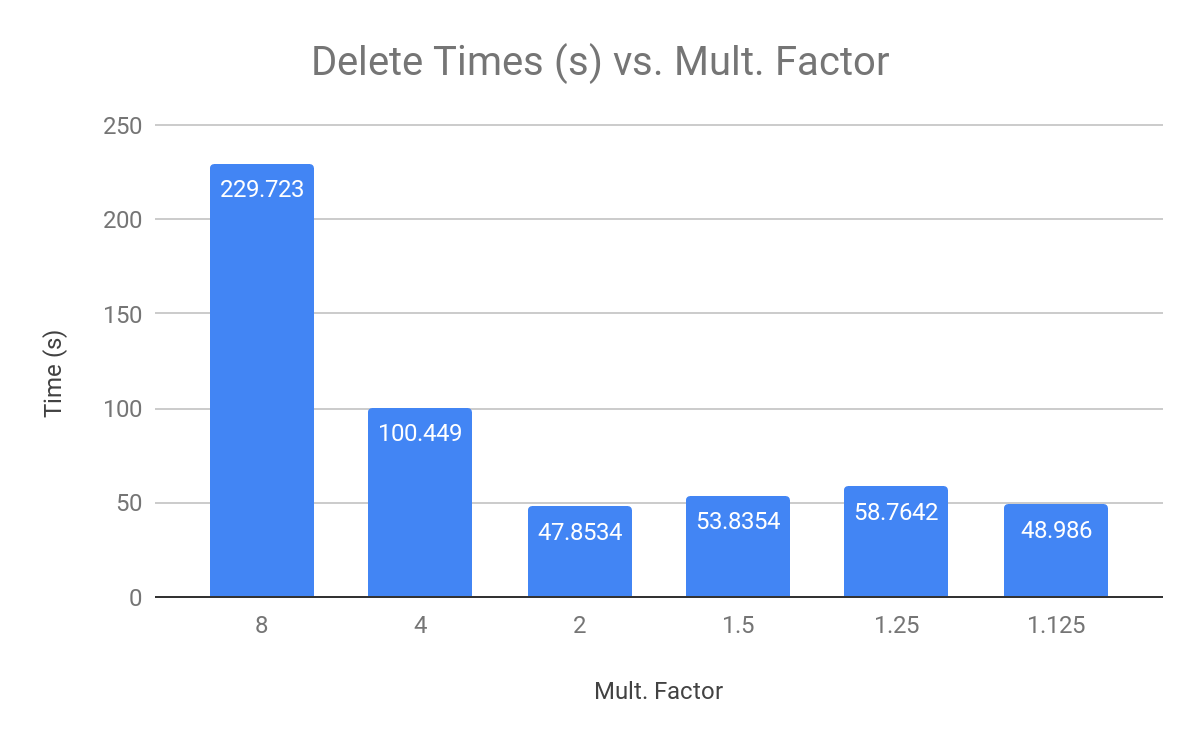
The data from Problem 4 is consistent with what we expected. When fewer words from the words.english.txt file were put into the symbol table, the time needed to put all the values was less (Refer to Table 1). These results remained true for getting each value from the symbol table. When getting fewer words from the symbol table, the time needed was less (Refer to Table 2).

|  |  |
| --- | --- |
| Chart | Chart |

#### Problem 5

The data from Problem 5 was more interesting to analyze. As can be seen in Tables 3 & 4 and the graphs below, the higher multiplicative factors generally had better times when placing the values from words.english.txt into a symbol table. We believe that this is because there would be a smaller chance of resizing because there would be sufficient null spaces for values to be placed in. However, it can be seen that the symbol tables with a multiplicative factor of 8 seemed to perform worse on average than the symbol tables with a multiplicative factor of 4. We believe that this is due to the insertion time in an array: because there are more spaces in the array, it would take more time to traverse it to place a value. For instance, if key-value hashes to a spot near the end of the array, it will take a lot longer to traverse to that spot if the multiplicative factor is 8, as the size of the array is a lot larger. This same issue is what causes the multiplicative factor of 8 to be slower than the multiplicative factor of 4 for getting each value from the hash table. Based on the data from this program, a multiplicative factor of 4 seems to be the optimal multiplicative factor for resizing. However, when deleting value from the symbol table, a multiplicative factor of 2 seems to be optimal. We believe this is due to the fact that every time a value is deleted, the program needs to check the entire array to make sure that every value is in the correct place, and to move any values that may have initially been placed in a different index than they were hashed to due to the way the LinearProbingHashST handles collisions.

|  |  |
| --- | --- |
| ChartChart |  |



|  |  |
| --- | --- |
| Chart | Chart |

