ESKISEHIR TECHNICAL UNIVERSITY

FACULTY OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

EEM 480 HOMEWORK 4

ALGORITHM AND COMPLEXİTY

Selin Akalın
40444602194

Purpose

The goal of this project is to read words in a text file, place them into a hash table, and then analyze the frequencies and other statistics of the words in this hash table.

Algorithm

Hash Class:

`table`: An array of LinkedLists to store WordInfo objects.
`size`: An integer representing the size of the hash table, initialized to 1000.

Constructor:
`Hash(int size)`: Initializes the hash table with a specified size.

WordInfo Class:

`word`: String, storing the word.
`index`: Integer, representing the index of the word.
`hash`: Integer, storing the hash value of the word.
`frequency`: Integer, storing the frequency of the word.

Constructors:
`WordInfo(String word, int index, int hash)`: Initializes a WordInfo object with word, index, and hash.
`WordInfo(String word, int frequency)`: Initializes a WordInfo object with word and frequency.

GetHash(String mystring):

1. Aggregation-Based ASCII Hashing:
   - This function creates an integer by adding the ASCII value of each character of a string.
   - The ASCII value of each character in the string `mystring` is added with the previous total value.
   - The resulting total is modulated according to the hash table size and an appropriate index is obtained.
2. XOR-Based ASCII Hashing:
   - This function creates an integer by concatenating the ASCII value of each character of a string with the XOR (bitwise exclusive OR) operation.
   - A new XOR operation is performed by subjecting the ASCII value of each character in the `mystring` to the previous XOR operation.
   - The resulting XOR value is modulated according to the hash table size and an appropriate index is obtained.
3. Prime Number-Based Hashing:
   - This function creates an integer by multiplying the ASCII value of each character by a prime number, 31. A new multiplication is performed by subjecting the ASCII value of each character in the `mystring` to the previous multiplication.
   - The resulting multiplication value is combined with the value obtained from the previous step.

- The resulting number is modulated according to the hash table size to obtain an appropriate index.
- This approach aims to provide a better hash distribution by using a prime number.

ReadFileandGenerateHash(String filename, int size):

1. Reads the `text.txt` file and processes each line.
2. Cleans up punctuation and extracts words from each line.
3. Gets the hash value of each word using the `GetHash` method.
4. Adds the `WordInfo` object, which contains the word, index and hash information, together with the obtained hash value, to the appropriate hash table.

printHashTable():

1. It browses the hash table and prints each index and the words belonging to this index on the screen.

DisplayResult(String OutputFile):

2. Calculates the frequency of each word and writes the results to the `DisplayResult.txt` file.
3. Adds words whose frequency has already been calculated to a list and prevents the same word from being calculated again.

DisplayResult():

1. Calculates the frequency of each word and prints the results on the screen.
2. Adds words whose frequency has already been calculated to a list and prevents the same word from being calculated again.

DisplayResultOrdered(String OutputFile):

1. It calculates the frequency of each word, sorts it by frequency, and writes the results to the `DisplayResultOrdered.txt` file.
2. Adds words whose frequency has already been calculated to a list and prevents the same word from being calculated again.

showFrequency(String myword):

1. Calculates and returns the frequency of the specified word (`myword`) in the hash table.

showMaxRepeatedWord():

1. It calculates the frequency of each word, sorts it by frequency, and returns the word with the highest frequency.

checkWord(String myword):

1. It calculates the frequency of the specified word (`myword`) in the hash table and prints the indexes where the word occurs on the screen.

TestEfficiency():

1. Calculates and returns the efficiency using the total number of elements in the hash table and the size of the hash table.
2. The Efficiency value could theoretically be negative. However, this usually indicates an undesirable situation. Efficiency is usually a value between 0 and 1, the closer to 1 the more effective.

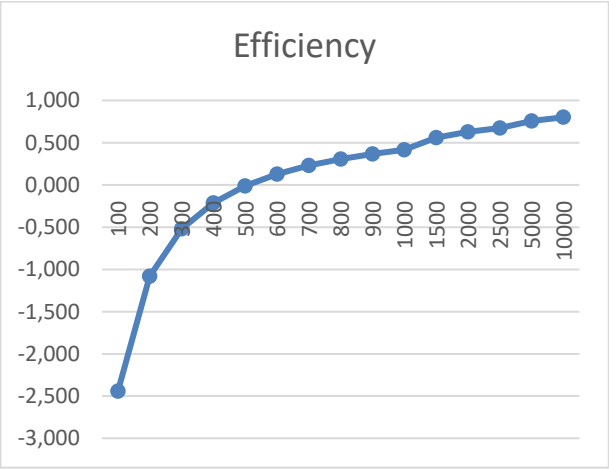
$$Efficiency = (1 - LoadFactor) * (1 - \frac{NumberofCollusion}{TotalElement})$$

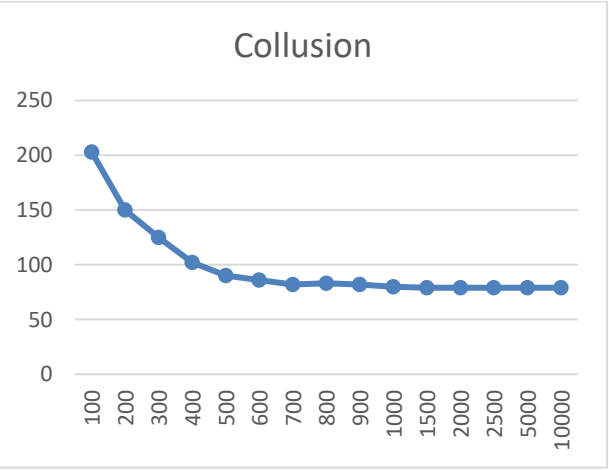
NumberOfCollusion():

1. There is an outer loop that traverses the hash table indices up to `size`.
2. The `currentList` representing the `LinkedList<WordInfo>` list corresponding to each hash table index is retrieved. Another inner loop represents each `WordInfo` item in this list.
3. Each time a new hash table index is passed, an `indexList` is created for this index. This list keeps the information of the words belonging to the current index.
4. The hash value of each word is retrieved from `WordInfo`. This hash value is compared with the hash values of words previously found in the same index. If the same hash value has been found in the `indexList` before and these words themselves are not the same, the collision counter (`collisionCount`) is incremented.
5. If the word has not been found in the `indexList` before, the `WordInfo` of that word is added to the indexList.
6. After checking all `hash` table indices and the words in these indices, the method returns the total number of collisions (`collisionCount`).

Different Hash Functions

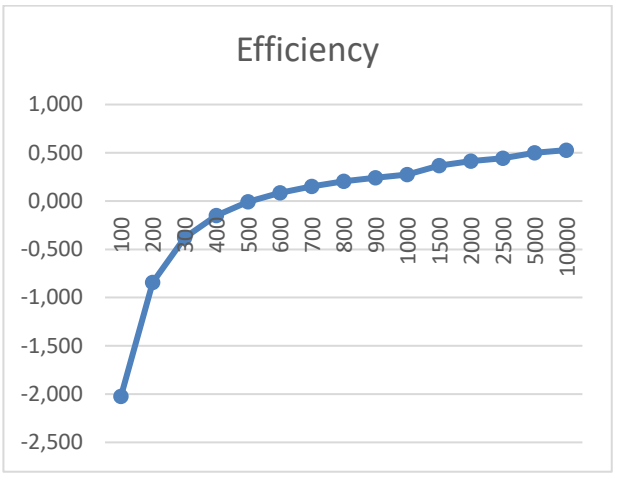
Hash Function 1 - Aggregation-Based ASCII Hashing

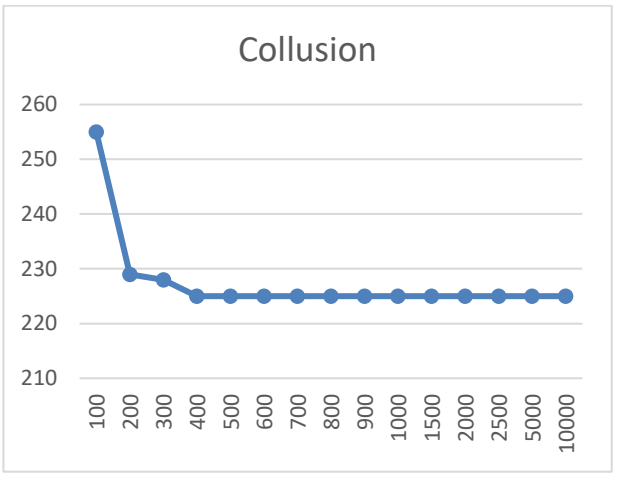




- ✓ The efficiency of 10000 size hash table is 0,80138147. A hash table of size 100 is -2,4403946. As you grow size, the load factor decreases proportionally and efficiency increases. In this case, a hash table of 10000 sizes is more efficient than 100 sizes.
- ✓ When the table size is chosen too large, the tendency to reduce conflicts may be weakened because the hash values are more distributed. This shows that conflicts do not decrease stably despite the small efficiency increases observed after size 700.

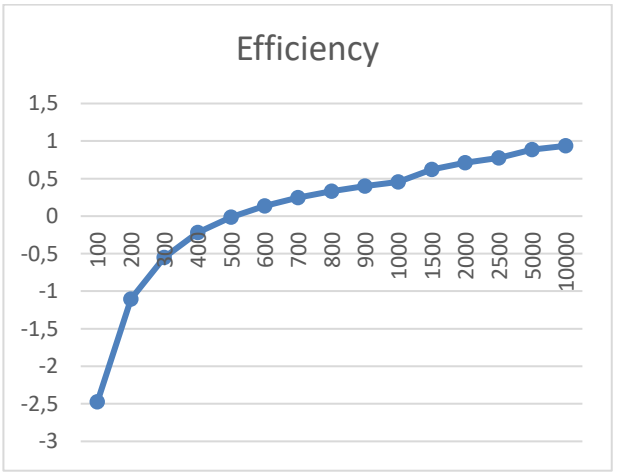
Hash Function 2 - XOR-Based ASCII Hashing

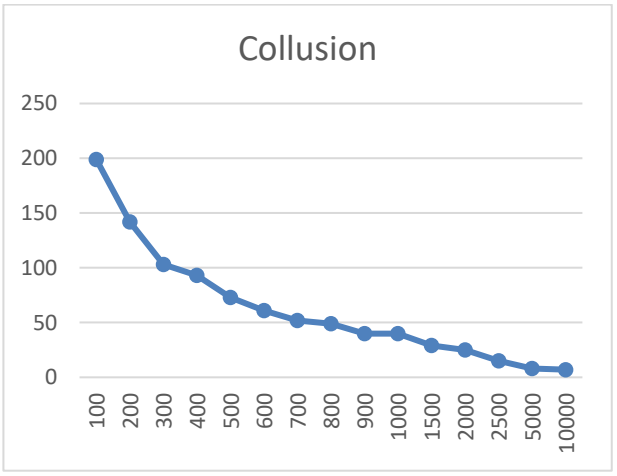




- ✓ The efficiency value is similar to Aggregation-Based ASCII Hashing, but there is a change between the values depending on the size. While Aggregation-Based ASCII Hashing gets closer to our target value of 1, XOR-Based ASCII Hashing still gives an efficiency value of 0.528013 at 10000 size. In this case Aggregation-Based ASCII Hashing works better.
- ✓ When we examine the collision values, we observe a serious decrease between 100-200 and there is no significant change after 400. We have a hash function that performs worse and produces more collisions in each dimension compared to Aggregation-Based ASCII Hashing.

Hash Function 3 -





- ✓ The efficiency value increases regularly and reaches 0.9361933, which is almost the value of 1 we aimed for.
- ✓ The collusion value also decreases regularly, falling to 7 and almost reaching 0.
- ✓ We can say that this function is better than others because it has the least collisions and the highest accuracy rate.

Referances

[1] https://medium.com/@AlexanderObregon/from-collision-to-resolution-a-guide-to-handling-collisions-in-java-hash-tables-d04ec06a497a

[2] https://www.scaler.com/topics/data-structures/separate-chaining/

[3] https://www.geeksforgeeks.org/c-program-hashing-chaining/

[4] https://www.infoq.com/articles/java-collections-streams/

[5] https://www.geeksforgeeks.org/stream-anymatch-java-examples/