**CMP2003 DATA STRUCTURES & ALGORITHMS TERM PROJECT**

Before we start explaining our code, we would like to compare the processing time of our own code **(hash table)** below and the processing time of the code we wrote using **unordered map structure**, which is one of the optimized hashing methods, in order to show efficiency.

metin, ekran görüntüsü, yazılım, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldumetin, ekran görüntüsü, yazı tipi, tasarım içeren bir resim

Açıklama otomatik olarak oluşturuldu

As you can see, when we run the program with the **unordered map structure**, the process time is around **0.4- 0.5 seconds**, while in our code this time it appears to be **0.8 seconds**. Considering the situations such as not being 100% optimized and some features we added to our code, we agreed as a team that the code performance was sufficient. **(Efficiency 62.5%)**

**STRUCTURES OF OUR CODE**

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu**HASH FUNCTION IMPLEMENTATION**

After a long hash function research and trying various functions, we decided on the following function, which is a bit of a classical structure but allowed us to get the **fastest result among the functions we tried**. If you ask why the starting value is 99999, we tried various numbers in order **to minimize the possibility of collision at the beggining and to get the fastest output**, and we saw that the code runs faster when we increase the number up **to a certain rate (10000).** Since the performance decreased after 99999, we chose to stay at 99999.

**HASH TABLE IMPLEMANTATION**

We started the hash table implementation, which is the most critical part of our Project, by first implementing the **linear probing method**, one of the **open addressing methods**, to prevent a possible collision situation. If you ask why this method, we chose this simple but effective method because we thought that the **number of unique elements in our file and the free space in RAM** would be low. In this way, in case of any collision, it would be much easier to immediately settle into a new index, as there would be large spaces in RAM, rather than dealing with creating a new space in RAM, such as the **chaining method**. Our goal was not to try to choose the most advanced method when writing the code, but to work according to the needs of our code.

metin, ekran görüntüsü, yazılım, ekran, görüntüleme içeren bir resim

Açıklama otomatik olarak oluşturuldu

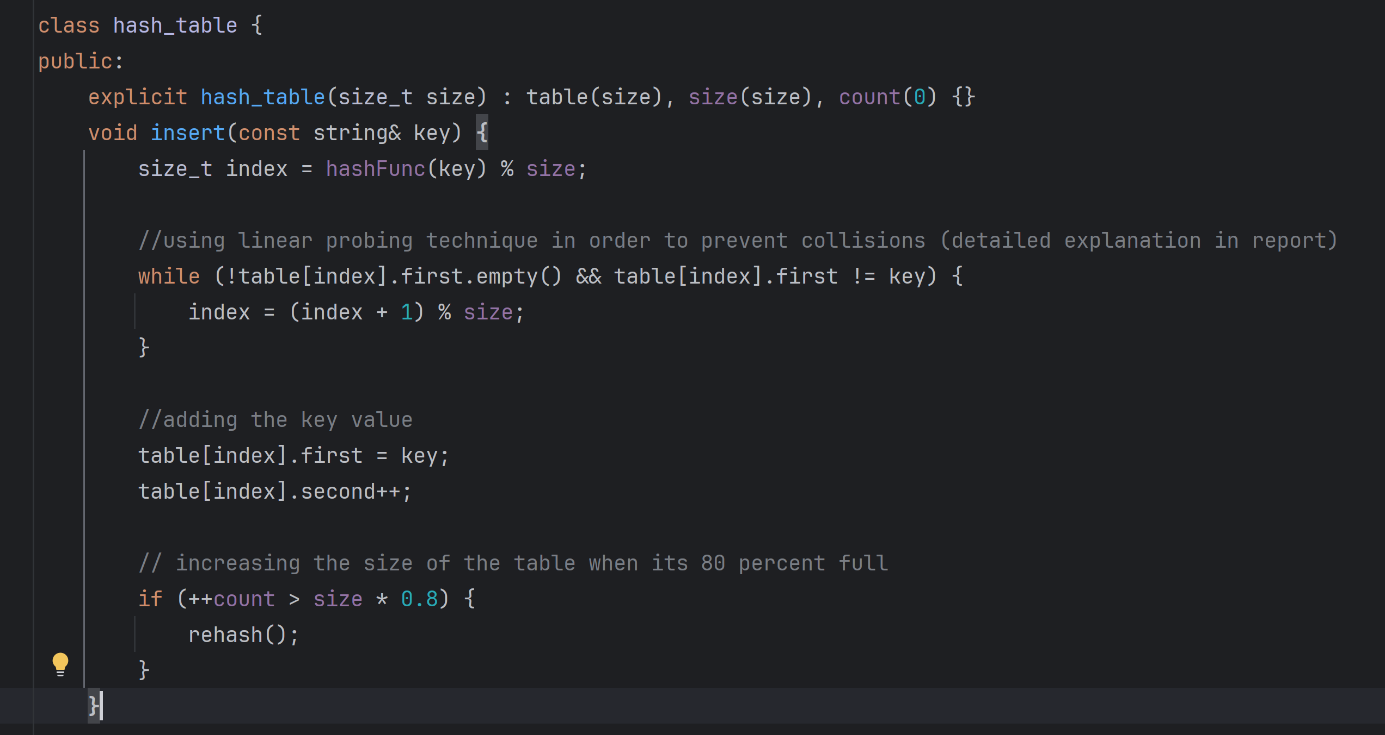
In the continuation of our hash table, we implemented the function that will start a loop on the hash table and read the items. This function allowed us to sort by number of visits.

metin, ekran görüntüsü, yazı tipi, yazılım içeren bir resim

Açıklama otomatik olarak oluşturuldu

**REHASHING**

At the beggining of our code, we added **a rehash function** to increase the size of the table after the table ratio reached **80%** during the hash table implementation. In the first image, you can see the enlargement condition we put in the hash table.





With this function, if our table size exceeds 80%, we aimed to double the current table size, start a cycle again, re-index the items and transfer them to a new hash table after using linear probing if necessary.

**MAIN FUNCTION STRUCTURE**

metin, ekran görüntüsü, yazılım, ekran, görüntüleme içeren bir resim

Açıklama otomatik olarak oluşturulduWe can say that we wrote our main function in 2 parts. In the first part, we accessed the Access\_log file and implemented the function that started measuring the time after starting the reading process. Then, we created a hash table with a capacity of **200 elements**. The reason we were comfortable with this number was that we knew that we could automatically increase its size through our **rehash function**. Afterwards, we prepared for the reading process and ensured that it would give an **error output if the file was not found**. In addition, we added a function that will **output the size of the file** at our own will.

In the second part of our main function, we aimed to read the Access\_log file line by line and analyze by HTTP requests and keep the number of these requests **in the web\_request\_count object**.

metin, ekran görüntüsü, yazılım, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

To find the part we needed, we searched for some characters **(such as / and “”)** in each line in the file and found our request area. After assigning it as a string and performing operations on these strings, we ensured that it was assigned to the hash table.

**FINDING TOP 10 VISITED PAGES**

We extracted and printed out the 10 most visited pages from all the data we kept in the **web\_request\_counts object** with the **.getTopItems()** function. To maket he output more readable, we added **“---“ separators.**

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

**MEASURING ELAPSED TIME**

We started measuring the time from the moment the process started in the main function section. After the files were read and sorted and the necessary outputs were taken, we finished the measurement and added the function that would output the total elapsed time and finished our Project.

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

**USED LIBRARIES DURING THE PROJECT**

**metin, yazı tipi, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldu**

**STUDENTS WHO DID THIS PROJECT**

**-Berika DİKİCİ 2200424**

**-Burak AKÇAY 2102816**

**-Beyza YAYLACIOĞLU 2001308**

**-Zeren KORKMAZ 2101901**