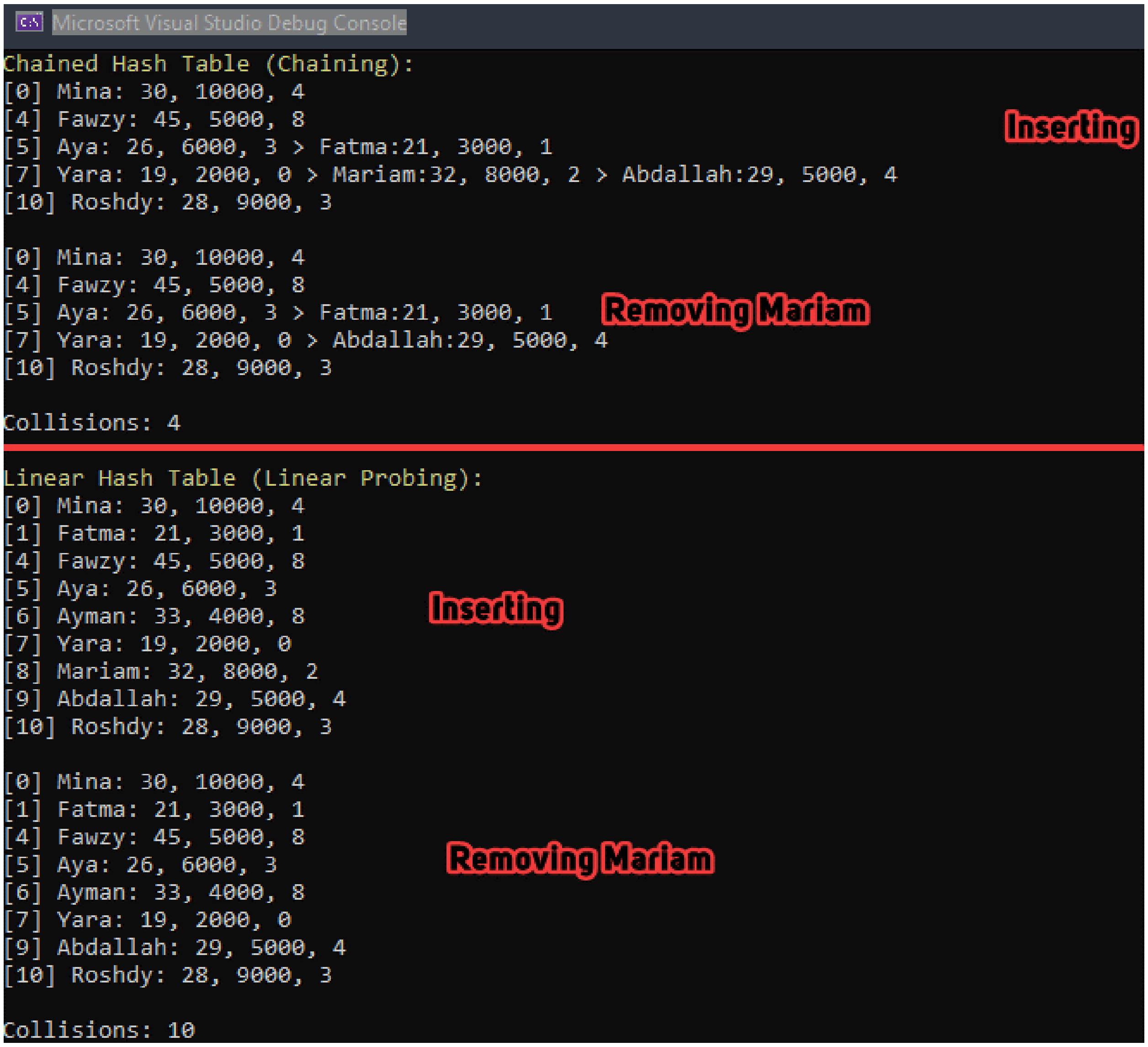
Mazen Kamel
900212271

Output (Size was 11)



```
Chained Hash Table (Chaining):
[0] Mina: 30, 10000, 4
[4] Fawzy: 45, 5000, 8
[5] Aya: 26, 6000, 3 > Fatma:21, 3000, 1
[7] Yara: 19, 2000, 0 > Mariam:32, 8000, 2 > Abdallah:29, 5000, 4
[10] Roshdy: 28, 9000, 3

[0] Mina: 30, 10000, 4
[4] Fawzy: 45, 5000, 8
[5] Aya: 26, 6000, 3 > Fatma:21, 3000, 1
[7] Yara: 19, 2000, 0 > Abdallah:29, 5000, 4
[10] Roshdy: 28, 9000, 3

Collisions: 4
```

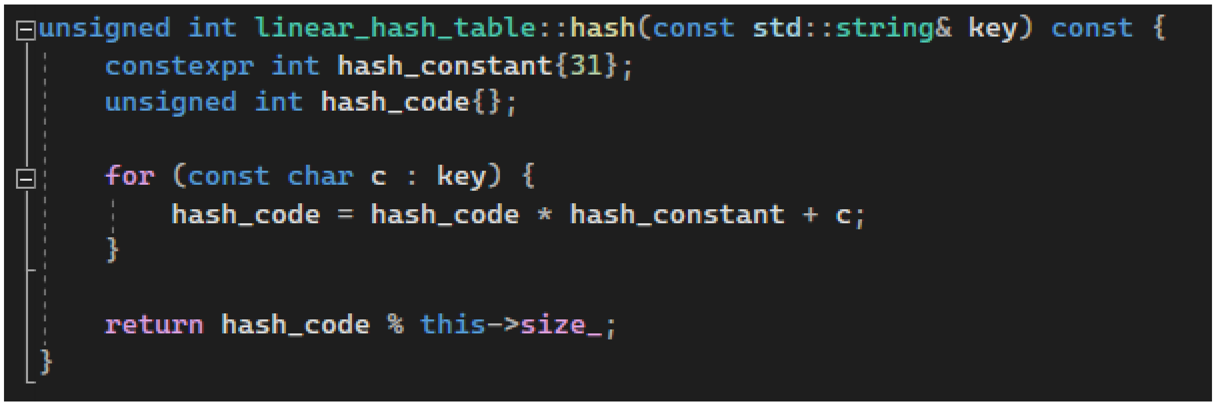
```
Linear Hash Table (Linear Probing):
[0] Mina: 30, 10000, 4
[1] Fatma: 21, 3000, 1
[4] Fawzy: 45, 5000, 8
[5] Aya: 26, 6000, 3
[6] Ayman: 33, 4000, 8
[7] Yara: 19, 2000, 0
[8] Mariam: 32, 8000, 2
[9] Abdallah: 29, 5000, 4
[10] Roshdy: 28, 9000, 3

[0] Mina: 30, 10000, 4
[1] Fatma: 21, 3000, 1
[4] Fawzy: 45, 5000, 8
[5] Aya: 26, 6000, 3
[6] Ayman: 33, 4000, 8
[7] Yara: 19, 2000, 0
[9] Abdallah: 29, 5000, 4
[10] Roshdy: 28, 9000, 3

Collisions: 10
```

For collisions in chaining, it was assumed that each node collision counted, hence 4 collisions. Since Adballah would collide once with Yara and once with Mariam, causing 2 collisions in addition to the previous 2 collisions by Mariam and Fatma.

Lab 7 (Hash tables) was used as reference on how to implement a hash table. However, the code in the lab was only read one to get an idea then the implementation was all on my own, with some influence from what I read.

Mazen Kamel
900212271

Hashing Function (Same for linear probing and chaining):

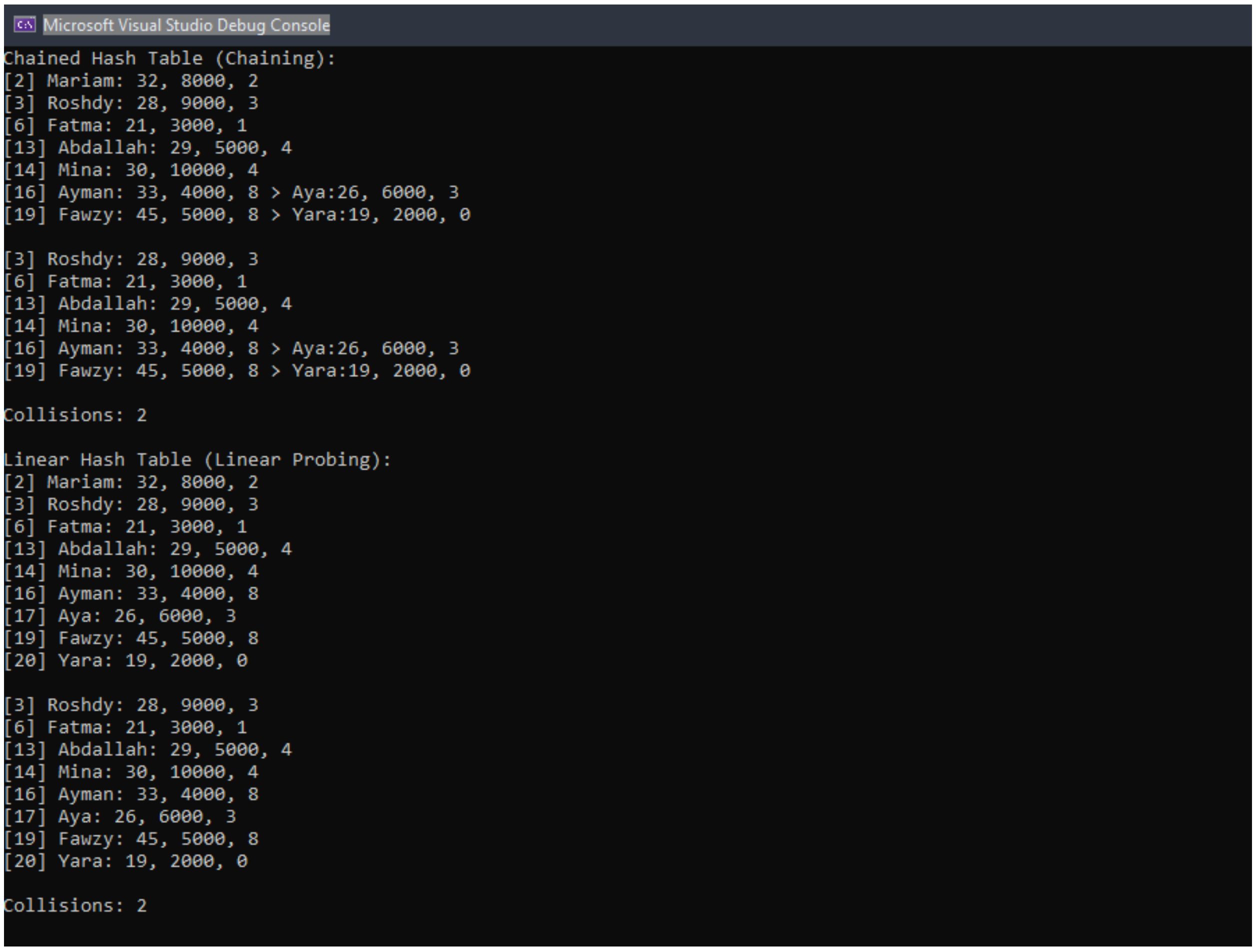


```cpp
unsigned int linear_hash_table::hash(const std::string& key) const {
    constexpr int hash_constant{31};
    unsigned int hash_code{};

    for (const char c : key) {
        hash_code = hash_code * hash_constant + c;
    }

    return hash_code % this->size_;
}
```

- Unsigned was used to prevent overflowing to negative, and to give more space for hashing code
- Hash constant 31 was used because it is an odd prime integer.
    - https://www.youtube.com/watch?v=jtMwp0FqEcg was used to implement it
    - https://stackoverflow.com/questions/299304/why-does-javas-hashcode-in-string-use-31-as-a-multiplier was used to understand why specifically 31, referencing the book “Effective Java”
- I used this hash function rather than just summing up the ascii values of the string to prevent collisions for string with same characters such as: ABC, and CBA. This has function includes additional multiplication with each consecutive index (character) thus including the order of the characters in the hash code, making it unique compares to another string with the same characters but different order.
  The has function also has a relatively low collision rate with fewer data. If the size were to be bigger than the data, the collision rate would greatly fall.
  A hash table size of 11 was chosen because it was bigger than the data count and it is prime. I could have chosen bigger sizes but 11 was fine.

Mazen Kamel
900212271

I made a test with bigger sizes (21, also prime) and the collision rates matched this time, both being 2. As mentioned previously, the bigger the size the lower the collision rate.



```
Chained Hash Table (Chaining):
[2] Mariam: 32, 8000, 2
[3] Roshdy: 28, 9000, 3
[6] Fatma: 21, 3000, 1
[13] Abdallah: 29, 5000, 4
[14] Mina: 30, 10000, 4
[16] Ayman: 33, 4000, 8 > Aya:26, 6000, 3
[19] Fawzy: 45, 5000, 8 > Yara:19, 2000, 0

[3] Roshdy: 28, 9000, 3
[6] Fatma: 21, 3000, 1
[13] Abdallah: 29, 5000, 4
[14] Mina: 30, 10000, 4
[16] Ayman: 33, 4000, 8 > Aya:26, 6000, 3
[19] Fawzy: 45, 5000, 8 > Yara:19, 2000, 0

Collisions: 2

Linear Hash Table (Linear Probing):
[2] Mariam: 32, 8000, 2
[3] Roshdy: 28, 9000, 3
[6] Fatma: 21, 3000, 1
[13] Abdallah: 29, 5000, 4
[14] Mina: 30, 10000, 4
[16] Ayman: 33, 4000, 8
[17] Aya: 26, 6000, 3
[19] Fawzy: 45, 5000, 8
[20] Yara: 19, 2000, 0

[3] Roshdy: 28, 9000, 3
[6] Fatma: 21, 3000, 1
[13] Abdallah: 29, 5000, 4
[14] Mina: 30, 10000, 4
[16] Ayman: 33, 4000, 8
[17] Aya: 26, 6000, 3
[19] Fawzy: 45, 5000, 8
[20] Yara: 19, 2000, 0

Collisions: 2
```

Another test with 51 as size was made and the collision rate dropped to 1.

Conclusion

A lot of research was done on effective hashing functions on string, most of which I did not understand. Notably the "djb2" hashing function here: https://stackoverflow.com/questions/7666509/hash-function-for-string, and http://www.cse.yorku.ca/~oz/hash.html, and https://stackoverflow.com/questions/1579721/why-are-5381-and-33-so-important-in-the-djb2-algorithm. I just can't really grasp why we would shift 5 bits specifically and why the initial value of 5381, I was also completely lost when I saw "Linear Congruential Generator". I settled for the YouTube video mentioned previously to implement a hash function.

As for hash tables, the chained hash table, using chaining, seemed to be more effective for smaller sizes with less collisions. However, as I increased the size of both hash tables, using linear probing and chaining, the collision rates started to match and the difference was no longer noticeable.

Mazen Kamel
900212271

I would image for larger sizes linear probing would be better as if there is a low collision rate I think accessing the indices even after collisions will be faster than using linked nodes, not sure though.