# Part A- Double Hash Map

(assuming net is put collisions, and total/max are total/max collisions)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Mult** | **Mod** | **secMod** | **Size** | **Net** | **Total** | **Max** |
| 1 | 4271 | 1 | 2000 | 843 | 173,036 | 1348 |
| 1 | 4271 | 223 | 2000 | 737 | 3,232 | 40 |
| 1 | 4271 | 647 | 2000 | 721 | 2846 | 51 |
| 1 | 4271 | 1 | 4000 | 524 | 30,743 | 552 |
| 1 | 4271 | 223 | 4000 | 437 | 1,172 | 19 |
| 1 | 4271 | 647 | 4000 | 408 | 994 | 17 |

1. If the double hash map has a secondaryModulus of 1, the secondary hash equals   
   **(1 – abs(hashcode)%1)**, which is always equal to 1. As a result, the function corresponds to a linear probing hash map.
2. From the results above, it appears that the use of double hashing reduces the ‘clustering’ that occurred in linear probing, and significantly reduces the number of iterations required to find a place for a given hash code. The total displacement of a given hash code is higher for double hashing.
3. Using a secondary hash function allows the hash codes to be better distributed in the function; instead of placing a colliding value in the next closest empty space as in linear probing, future put attempts are in more distant parts of the map. For this same reason, the total displacement of a hash code is higher in double hashing.
4. Not all of the keys were able to be entered in the map in the secondary hash function; a secondary modulus of 223 encountered a key-entry failure after storing 1642 hashes, while a secondary modulus of 647 failed after only 1139 hashes were stored. The reason for this error is that the secondary modulus produced was a factor of the map’s size; adjusting the size of the double hash map to a prime number would negate this issue.
5. The results above suggest that for real-world applications, care must be taken to ensure that when using a double hash map, the map should have a prime-number size to avoid key-entry failures.

# Part B- Password Manager

## Required properties:

* Minimize Hash Collisions: In the event of a hash collision, multiple passwords would generate the same hash code, rendering them interchangeable[[1]](#footnote-1).
* Pre-Image Resistant (One-Way): If a list of hashed passwords is obtained, it should not be possible to discern the password from the hash code[[2]](#footnote-2), *even if the function is known*.
* Slow: Only one valid password is required by attackers in order to compromise the hashing function’s security. As it is possible to determine an accepted password through brute-force[[3]](#footnote-3), slower hash functions are important for impeding such attacks[[4]](#footnote-4).

## sdbm Hash Function (derived from C code)[[5]](#footnote-5) :

BEGIN sdbm

hash = 0

FOR char IN str

hash = char + (LEFTSHIFT hash by 6) + (LEFTSHIFT hash by 16) - hash

ENDFOR

RETURN hash

END sdbm

hash is an unsigned long, as otherwise the bit shifts may produce negative values, complicating the function’s logic significantly.

Bit shifting and subtraction is performed on the hash code in order to vary the bits with each character. This ensures the order of the characters affects the output code.

## Test Report:

**Performance of sdbm hashing function**

We found the sdbm function to perform optimally for password hashing, producing a unique hash code for all 10150 passwords in the dataset provided.

This is an important result, as two passwords which generate the same hash code are effectively interchangeable. As this causes an affected user’s account to have multiple valid passwords, such a conflict can heavily reduce security.

Although highly effective at producing unique hash codes, the biggest flaw in the sdbm function is its speed. Our implementation was able to generate 10150 hashes in under 37 milliseconds, which is too fast to impede any significant brute-force attempts. Increasing the complexity of the calculations would significantly increase the function’s security in such a situation.

# Part 3- Skip-List

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| name | searchSteps | | |  |
|  | expt 1 | expt 2 | expt 3 | Average |
| SMITH | 13 | 8 | 7 | 9.33 |
| JOHNSON | 4 | 4 | 9 | 5.67 |
| WILLIAMS | 9 | 9 | 9 | 9 |
| BROWN | 4 | 1 | 9 | 4.67 |
| JONES | 2 | 3 | 11 | 5.33 |
| MILLER | 5 | 3 | 13 | 7 |
| DAVIS | 14 | 6 | 4 | 8 |

1. From the results above, the average lookup time for a given surname fell between 4.6 and 9.4 steps, which is on the same scale as the predicted lookup time of log(1000).
2. In the fastest implementation of the hash map (1, 4271, 647, 4000), the average number of probes was found to be 1.55 compared to the skip list’s average of 7 steps, while the worst-case performance was similar for both functions. In the slowest hash map implementation however, the average number of probes was 96, indicating that hash maps are only the faster data structure when implemented correctly.
3. For highly optimized applications a hash map would be the better representation, as its theoretical lookup time of O(1) is much faster than a skip-list’s O(logn) expected lookup time. However, as the performance of hash maps can decrease significantly if incorrectly implemented or heavily loaded, for more general-purpose applications a skip-list is the safer and more reliable option.

1. "Network Security" Princeton University, accessed October 2, 2015, http://www.cs.princeton.edu/courses/archive/spr11/cos461/docs/rec08-net-security.pdf [↑](#footnote-ref-1)
2. Menezes, A. J., and Paul C. Oorschot. *Handbook of Applied Cryptography*. (Boca Raton: CRC Press, 1997) [↑](#footnote-ref-2)
3. Halderman, J., Waters, B., and Felten, E. *A Convenient Method for Securely Managing Passwords* (Chiba: International World Wide Web Conference Committee, 2005) [↑](#footnote-ref-3)
4. Halderman, J., Waters, B., and Felten, E. *A Convenient Method for Securely Managing Passwords* (Chiba: International World Wide Web Conference Committee, 2005) [↑](#footnote-ref-4)
5. “Hash Functions” York University, accessed October 2, 2015, http://www.cse.yorku.ca/~oz/hash.html [↑](#footnote-ref-5)