Algoritmos y Estructuras de Datos

Práctica 8

Análisis de distintas implementaciones de tablas hash

Xiana Carrera Alonso

xiana.carrera@rai.usc.es

Diciembre, 2020

Índice

1. Introducción

En este trabajo se discutirán las diferencias en eficiencia entre distintas implementaciones para tablas hash, de acuerdo a diferencias en el tamaño de la tabla, la función hash elegida, la estrategia de resolución de colisiones (encadenamiento y varios tipos de recolocación) y el tipo de dato utilizado como clave.

La tabla hash se plantea con el objetivo de guardar 10000 datos leídos de un fichero, estructuras con información sobre los jugadores de AmongETSE: su nombre, sus apellidos, su nick y su correo electrónico. Las pruebas que se realizarán comprobarán el rendimiento tanto para la inserción como para la búsqueda de elementos.

1.1. Terminología

Para las comparativas de inserción se utilizarán dos propiedades: las colisiones y los pasos adicionales.

Una colisión se produce cuando el resultado de la función hash para un determinado elemento devuelve una posición que ya había sido asignada anteriormente para otro dato. Con recolocación, se puede probar en otra ubicación diferente, con la posibilidad de escoger entre varias estrategias, como simple, lineal o cuadrática.

Con encadenamiento, una colisión se produce si la lista correspondiente a la posición a la que la función hash envía un elemento es no vacía, es decir, que ya le fueron añadidos datos.

Los pasos adicionales tienen lugar en el caso de que se hayan producido colisiones. Cada nuevo intento de inserción se cuenta como un paso extra. Utilizando recolocación, esto significa que cada espacio comprobado ya ocupado suma una operación a mayores. Utilizando encadenamiento, nunca hay operaciones extra en la inserción, ya que un nuevo elemento siempre se colocará al inicio de su correspondiente lista. Por consiguiente, independientemente de la longitud de dicha lista, se realizarán las mismas secuencias para cada inserción.

La implementación de los aspectos mencionados en inserción se ha planteado de la siguiente forma:

En primer lugar, con recolocación, dado un jugador se calcula la posición en la que debe ser colocado usando una función hash. Si dicha posición está vacía o el elemento que había anteriormente ha sido borrado, el dato se guarda y se devuelve un total de 0 colisiones y 0 pasos adicionales. Si está ocupada por el mismo dato, se devuelven igualmente 0 colisiones y 0 pasos. En caso contrario, se toma nota de la colisión y se continúa buscando.

Ahora, se recalcula la posición y se vuelve a intentar insertar. Si una posición está vacía, tenía un elemento que fue borrado o contiene el mismo dato, se ha encontrado un lugar válido y se devolverá como resultado que hubo 1 colisión y un número de pasos adicionales igual al número de intentos (sin contar el inicial que se comentó).

Si no hay posiciones libres en la tabla, se determina que hubo 1 colisión y que el número de pasos extra es igual al tamaño total - 1, donde ese -1 se debe a que la primera inserción no es "extra".

Con encadenamiento, se tienen operaciones suplementarias si el elemento no es el primero de la lista de la posición que le corresponde según la función hash empleada. El total de pasos lo dará el total de miembros de la lista que se tengan que recorrer antes de encontrar el deseado.

En cuanto a la implementación, para recolocación, como el número de pasos adicionales debe ser el número de comprobaciones con elementos, bastará con contar el número de veces que haga falta llamar a la función hash y restarle 1 (para no tener en cuenta el primer intento). En caso de el elemento no esté en la tabla, el número de pasos adicionales se devolverá como el tamaño - 1,

donde, de nuevo, ese -1 se debe a que la primera búsqueda no es "extra".

```
1
2  /* Función que localiza la posicion para insertar un elemento */
3  int _PosicionInsertar(TablaHash t, char *cad, int *colisiones, int *adicionales) {
4    // Devuelve el sitio donde podríamos poner el elemento de clave cad
5
6    int ini, aux, i, a = 1;
7
8    ini = Hash(cad);
9
10   // Se comprueba si ya se insertó un elemento anteriormente en la posición ini
       para determinar si hay colisión
11   aux = ini %Tam;
12   if (t[aux].clave[0] == VACIO || t[aux].clave[0] == BORRADO || !strcmp(t[aux].
       clave, cad)){
13     *colisiones = 0;
14     *adicionales = 0;
15     return aux;          // Se devuelve la posición en la que irá el elemento
16   }
17
18   // Si la posición no está desocupada, hubo una colisión. Falta por calcular los
       pasos adicionales
19   *colisiones = 1;
20
21   for (i = 1; i < Tam; i++) {
22     aux = (ini + a*i) %Tam;              // Posición a comprobar
23     if (t[aux].clave[0] == VACIO || t[aux].clave[0] == BORRADO){
24       // Se ha encontrado un hueco
25       *adicionales = i;
26       // Se han necesitado i intentos para ubicar el dato
27       return aux;
28     }
29     if (!strcmp(t[aux].clave, cad)){
30       // El valor ya está en la tabla
31       *adicionales = i;
32       // De igual forma, se ha encontrado la ubicación del dato tras i intentos
33       return aux;
34     }
35   }
36
37   // Se ha recorrido toda la tabla. El contador de pasos adicionales valdrá Tam - 1
38   *adicionales = i - 1;
39   // El -1 se debe a que el primer intento de inserción no se cuenta como paso
       extra
40   return ini;
41 }
```

Para encadenamiento, el proceso se simplifica. Después de llamar a la función hash, tan solo hace falta realizar una comprobación sobre la lista que corresponde. Si es vacía, no hubo colisión. Si es no vacía, hubo colisión.

```
1  /* Devuelve 1 si hubo colisión y 0 en caso contrario */
2  int InsertarHash(TablaHash *t, tipoelem elemento) {
3    int pos, colisiones = 0;
4
5    pos = Hash(elemento.clave);
6
7    if (!esvacia((*t)[pos])) colisiones = 1;
8    inserta(&(*t)[pos], primero((*t)[pos]), elemento);
9
10   return colisiones;
11 }
```

En la búsqueda, se hablará de pasos adicionales en el caso de que el elemento no se encuentre al primer intento.

Con recolocación, esto significa que la posición que le correspondería por la función hash utilizada está ocupada por un dato distinto al mencionado (es decir, fue recolocado en otra ubicación). Cada comprobación infructuosa de un espacio cuenta como un paso.

```c
/* Función que localiza la posición del elemento cuando buscamos*/
int _PosicionBuscar(TablaHash t, char *cad, int *adicionales) {
  /* Devuelve el sitio donde está la clave cad, o donde debería estar. */
  /* No tiene en cuenta los borrados para avanzar.*/

  int ini, i, aux, a = 1;

  ini = Hash(cad);

  for (i = 0; i < Tam; i++) {
    aux = (ini + a * i) %Tam;
    if (t[aux].clave[0] == VACIO) {
      *adicionales = i;
      // Se devuelve el número de pasos adicionales que se han tenido que dar
      return aux;
    }
    if (!strcmp(t[aux].clave, cad)) {
      // Se ha encontrado el dato
      *adicionales = i;
      return aux;
    }
  }

  // Se ha recorrido toda la tabla, pero no se ha encontrado el elemento.
  *adicionales = i - 1;
  // El número de pasos adicionales será Tam - 1, para no contar la primera bú
    squeda
  return ini;
}
```

Para encadenamiento, basta contar el número de miembros de la lista que no coinciden con el elemento que se está buscando, ya que la lista se irá recorriendo en orden.

```c
int Busqueda(TablaHash t, char *clavebuscar, tipoelem *e, int *adicionales) {
  posicion p;
  int enc;
  tipoelem ele;
  int pos = Hash(clavebuscar);

  p = primero(t[pos]);
  enc = 0;
  *adicionales = 0;

  while (p != fin(t[pos]) && !enc) {
    // Se busca mientras no se haya recorrido toda la tabla y no se haya encontrado
     el elemento
    recupera(t[pos], p, &ele);
    if (strcmp(ele.clave, clavebuscar) == 0) {
      // El elemento recuperado es el que se está buscando
      enc = 1;
      *e = ele;
    } else {
      // No se ha encontrado (aún) el dato
      p = siguiente(t[pos], p);
      // Implica un paso adicional más
      (*adicionales)++;
    }
  }

  if (p == fin(t[pos]) && !enc){
    // Se ha llegado al final de la lista y el elemento no estaba
    (*adicionales)--;
    // Como máximo se realizan tantos pasos adicionales como la longitud de la
    lista - 1
    // Por tanto, el último (*adicionales)++ del bucle while no debe contarse
  }

  return enc;
}
```

2. Influencia del tamaño elegido para la tabla

En primer lugar, se determinará cómo afecta el tamaño total de la tabla con respecto a las estrategias de recolocación simple y encadenamiento, utilizando en ambos casos la función hash 2.

2.1. Recolocación simple

En primer lugar, analicemos cuál debe ser un tamaño razonable para la tabla. Definimos el factor de carga, L, como el cociente entre el número de elementos almacenados en la tabla y el número de posiciones de la misma, esto es:

$$L = \frac{n}{N} \tag{1}$$

Al insertar un elemento en la tabla, puede ocurrir que esta posición ya esté ocupada, y que se tengan que comprobar más ubicaciones. No obstante, si el tamaño elegido es demasiado pequeño, la tendencia del comportamiento de la inserción será a formar grandes bloques de elementos costosos de recorrer.

Por tanto, debemos intentar que la media de ensayos que se deben realizar para la inserción no aumente, sino que se mantenga constante. Esto se consigue cuando $L \leq \frac{1}{2}$ o, equivalentemente, cuando $N \geq 2n$.

En el caso que nos ocupa, tenemos n = 10 000 elementos, de modo que un tamaño razonable será aquel mayor o igual a 20 000. Ahora que ya tenemos una estimación, podemos empezar a realizar la medición. Comprobemos los siguientes valores para Tam: 15000, 15013, 19997, 20000, 20003, 24989, 25000.

2.1.1. Mediciones

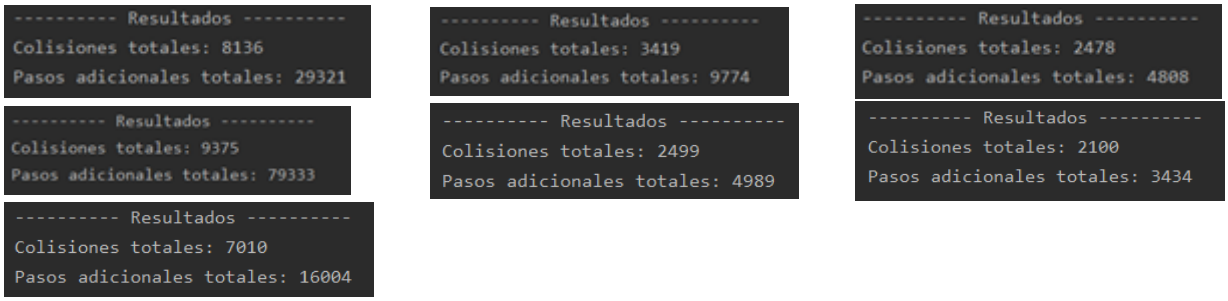


Figura 1: De izquierda a derecha, resultados para tamaños 15000, 15013, 19997, 20000, 20005, 24989 y 25000

Tam	Colisiones	Pasos adicionales
15000	8136	29321
15013	3419	9774
19997	2478	4808
20000	9375	79333
20005	2499	4989
24989	2100	3434
25000	7010	16004

Cuadro 1

Se han elegido 7 valores: 2 por debajo del mínimo recomendado (15000 y 15013), 3 alrededor del mismo (19997, 20000 y 20005) y 2 superiores (24989 y 25000). Entre todos ellos, hay 3 primos:

15013, 19997 y 24989.

A primera vista, es sencillo encontrar 3 valores que producen resultados especialmente malos: 15000, 20000 y 25000. Esto se debe al gran número de factores que comparten con el número de datos, 10000:

$$10000 = 2^4 * 5^4 \tag{2}$$

$$15000 = 2^3 * 3 * 5^4 \tag{3}$$

$$20000 = 2^5 * 5^4 \tag{4}$$

$$25000 = 2^3 * 5^5 \tag{5}$$

Ello provoca que en la función Hash 2, al realizar el módulo del resultado por el tamaño de la tabla, una gran cantidad de datos se vea asignada a las mismas posiciones.

Entre los 3, Tam = 20000 es la peor elección, al ser exactamente el doble del número de datos. Este valor será descartado en siguientes pruebas, pues tal característica siempre provocará resultados por encima de los esperados. En cambio, Tam = 20005 sí devuelve resultados adecuados, aun a pesar de no ser primo, por compartir muchos menos factores con 10000: $20005 = 5 * 4001$. Nótese que entre ambos tamaños solamente hay una diferencia de 5 números, pero esta se ve reflejada en las estadísticas con casi 7000 colisiones menos.

A pesar de ser malas elecciones, 15000 y 25000 ya reflejan una predisposición de la tabla: a mayor tamaño, menor probabilidad de colisiones (siempre que no se den circunstancias especiales como la de 20000 o el caso de números primos, por ejemplo). Esto está relacionado con lo indicado sobre el factor de carga: con un $L <= \frac{1}{2}$, presumiblemente se alcanza un estado óptimo en cuanto a la proporción de espacios vacíos en la tabla. Esto disminuye la probabilidad de que haya colisiones, lo que, a su vez, provoca que los datos queden más uniformemente repartidos, sin obstruirse los unos a los otros durante el proceso de inserción. 15000 da resultados peores porque en su caso, $L = \frac{2}{3} > \frac{1}{2}$, mientras que para 25000, $L = \frac{2}{5} < \frac{1}{2}$.

Sin embargo, se puede apreciar que 15013, 19997 y 24989 producen resultados mejores que 15000, 20005 y 25000, respectivamente. Esto se debe a que son números primos y, por consiguiente, primos con 10000, de modo que tienden a repartir mucho más uniformemente los datos (es menos probable que tengan factores en común con el valor hash generado y, por tanto, que se envíen a las mismas posiciones después de realizar el módulo por el tamaño). Nótese de nuevo la gran diferencia en los resultados en contraposición a la pequeña variación en el tamaño, apenas imperceptible en términos relativos.

Cabe destacar también la proporción entre colisiones y pasos adicionales. Por la propia definición de paso adicional, lo normal es esperar que a menos colisiones haya menos pasos adicionales (pues se formarán menos bloques de elementos que dificulten la inserción, los verdaderos culpables de dichas operaciones extra). No obstante, pueden darse casos en los que esto no se cumpla (por ejemplo, si se forman bloques pequeños, en lugar de ocupar un gran espacio). A partir de los resultados obtenidos, podemos determinar que la tendencia general es que los mejores tamaños son los que tienen un mejor comportamiento en este aspecto. Aunque el número de colisiones de 19997 y 20005 es similar, el valor primo da lugar a menos pasos adicionales, ya que los grandes bloques anteriormente mencionados son más infrecuentes.

2.2. Encadenamiento

Razonemos primero qué tamaños son adecuados cuando la resolución de colisiones se realiza por encadenamiento. Aunque puede parecer contraintuitivo, la relación entre n, número de datos, y N, tamaño de la tabla, no tiene restricciones: estamos utilizando listas que pueden tener cualquier tamaño. Se podría usar incluso una tabla con una única lista en la que se colocan todos los datos. Sin embargo, claramente esta sería la peor situación posible, y estaríamos desaprovechando las ventajas de las tablas hash.

A raíz de lo anterior, es preferible asegurar una distribución uniforme de los jugadores. Los factores de carga recomendados son aquellos que cumplen $L \leq 0{,}75$, es decir, $N \geq \frac{4n}{3}$.

Para el caso que nos ocupa, deberíamos tener $Tam \geq \frac{4*10000}{3} = 13333{,}3333$, de lo que se deduce que el encadenamiento es menos exigente que la recolocación.

2.2.1. Mediciones

Probemos a ejecutar el programa con los siguientes tamaños: 12000, 12007 (primo), 15000, 15013 (primo), 19997 (primo), 20000 y 20005.

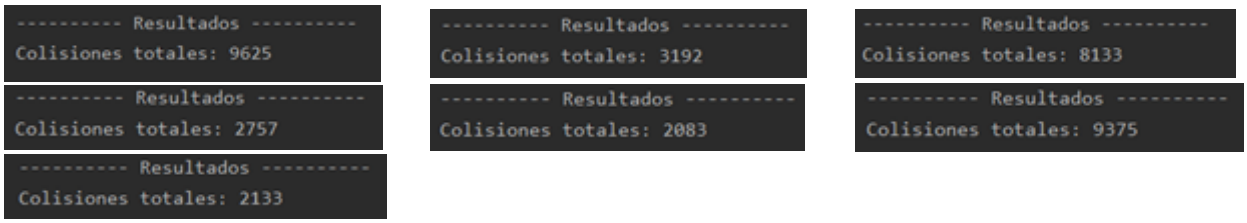


Figura 2: De izquierda a derecha, resultados para tamaños 12000, 12007, 15000, 15013, 19997, 20000 y 20005

Tam	Colisiones
12000	9625
12007	3192
15000	8133
15013	2757
19997	2083
20000	9375
20005	2133

Cuadro 2

El fundamento para las estadísticas mostradas es análogo a la del caso de recolocación.

De nuevo, se puede observar tanto la importancia del tamaño elegido como de si este es primo o no.

Hay una variación importante entre el número de colisiones entre los primos y sus parejas no primas: entre 12000 y 12007, por ejemplo, se disminuye el problema en más de 6000 unidades.

De igual modo, la influencia del factor de carga no es despreciable: del cambio de 12007 a 15013 se logra limitar las confluencias en más de 400 unidades, y de 15013 a 19997, en casi 700. Este fenómeno se nota también entre 12000 y 15000, por ejemplo.

Vuelve a haber un pico en el valor 20000, por el mismo motivo que en recolocación: es exactamente el doble del número de datos. La semejanza de la descomposición de este último con 15000 y 12000 ($12000 = 2^5 * 3 * 5^3$) también explica la baja eficiencia de los mismos. Comparando 20000 con 20005, se reafirma la importancia de evitar estos valores "redondos".

En el caso del encadenamiento, es interesante señalar que el número de colisiones es exacto, en el sentido de que cada elemento va a introducirse siempre en la lista que le corresponde por la función hash 2, y no en otra. Esto es, cuando en recolocación había algún conflicto, este se resolvía ocupando una posición anteriormente vacía, por lo que un dato que tuviera que ser asignado a ella daría otro choque en el futuro. En contraposición, tal situación no ocurre en encadenamiento. Las listas permanecen vacías hasta que un jugador que deba ir a la misma sea leído.

3. Influencia de la elección de la función hash y de la clave utilizada para obtener la posición del jugador en la tabla

Para empezar, estudiemos las características de las 3 funciones hash incluidas en el proyecto.

3.1. Análisis de las funciones

3.1.1. Hash 1

Hash 1 genera una posición en la tabla a partir de una clave sumando los valores ASCII de cada uno de los caracteres de dicha clave. Posteriormente se realiza el módulo por el tamaño de la tabla, para restringir los resultados al espacio total. No obstante, teniendo en cuenta los potenciales tamaños de la tabla (como mínimo tendrá 10 000 posiciones) y las características de la clave elegida, resulta una función hash inadecuada.

El campo del nick tiene, como máximo, 20 caracteres, por lo que la mayor posición posible sería 20*126=2520 (teniendo en cuenta que el último carácter ASCII es ' ', con valor 126). Como 2520 es un número muy inferior ya incluso al tamaño mínimo total, 10 000, tendrá lugar un gran número de colisiones: se intentaría colocar todos los elementos en esas 2520 primeras posiciones, pero llegado el momento en el que se empiece a llenar ese espacio, se impondría la necesidad de reasignar muchos de ellos por recolocación, o de empezar a llenar las listas por encadenamiento.

Si bien esta situación ya es bastante problemática, se debe tener en cuenta que 2520 resulta de una sobreestimación. Una comprobación de la estructura de las claves muestra que la longitud de la mayoría ronda los 10 caracteres, y no se utiliza toda la variedad de caracteres ASCII. Se empieza siempre por '@', por lo que el primer carácter no aporta ningún tipo de variedad al resultado del cálculo y, además, los restantes son letras minúsculas, cuyos valores se restringen al intervalo [97, 122]. En caso de que se repitan nombre e iniciales entre usuarios, hay un único dígito al final, pero ese dígito no ofrece la suficiente diferencia cuando ya hay tantos caracteres iguales entre claves.

En conjunto, estas particularidades provocan que todas las posiciones generadas tengan valores muy próximos, incrementando todavía más la probabilidad de colisiones.

3.1.2. Hash 2

Hash 2 realiza una suma ponderada de los valores ASCII de los elementos de la clave, transformándola a base 256. A cada carácter se le da un peso correspondiente al índice que ocupa dentro de la cadena, esto es, 256^i, siendo i dicho índice. Así, se lleva la cuenta de una suma que en cada iteración se ve multiplicada por 256 (aumentando en 1 el índice de cada uno de los elementos ya añadidos). Se añade entonces el ASCII del nuevo carácter, y se realiza el módulo del total hasta el momento por el tamaño de la tabla.

De este modo, se obtienen valores mucho más altos que con Hash 1, por lo que realizar el módulo se vuelve completamente necesario ya antes de aplicar recolocación o encadenamiento. El hecho de realizar el módulo no tras realizar la suma completa, sino en cada iteración, aumenta la variedad en la generación de números.

Al contrario de Hash 1, Hash 2 sí es una función hash viable para los requisitos especificados.

3.1.3. Hash 3

Hash 3 utiliza el mismo principio que Hash 2, con diferentes valores para la base. Las bases que sean coprimas con el tamaño de la tabla usado deberían dar una mayor optimización del espacio, pues realizar el módulo repartirá valores aleatorios equitativamente en las posiciones de la tabla. Eligiendo números primos no múltiplos entre sí tanto para Tam como para k se puede asegurar

que sean coprimos.

3.2. Recolocación - Comparación de los resultados

Estudiemos ahora las estadísticas de ejecuciones con Hash 1, Hash 2 y Hash 3.

3.2.1. Hash 1

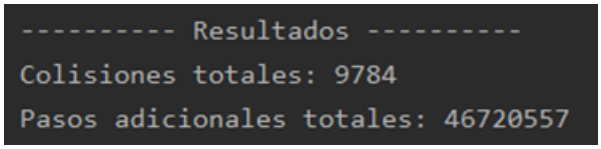


Figura 3: Resultado de todas las pruebas para Hash 1 con tamaños mayores de 10000

El comportamiento de la función deja mucho que desear. Por lo anteriormente expuesto, prácticamente todos los elementos provocan colisiones, con tan solo 216 que no lo hacen.

El número de pasos adicionales es igualmente exorbitado. Debido a todas las colisiones que hay, se irá formando un gran bloque que, para colocar un nuevo jugador, tendrá que ser recorrido en su completitud. De este modo, una vez todos los elementos empiecen a dar colisión, su número de pasos adicionales será igual al del elemento anterior + 1, pues se tendrá que avanzar un espacio más en el bloque. De aquí se deduce que las operaciones adicionales llegarán a tener una magnitud colosal.

3.2.2. Hash 2

Se remite a los resultados del apartado 2.

3.2.3. Hash 3

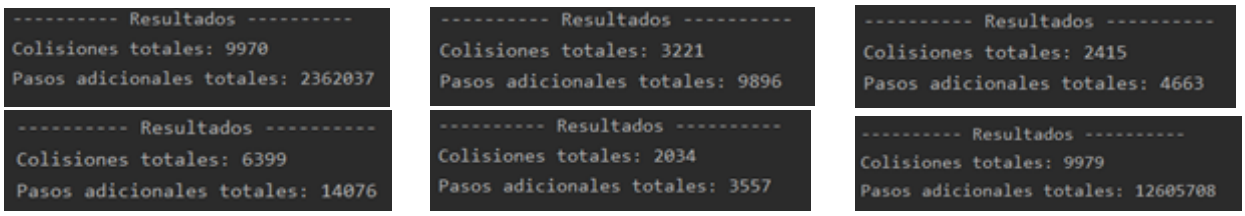


Figura 4: De izquierda a derecha, resultados para tamaños 15000, 15013, 19997, 20005, 24989 y 25000, con k = 500

Tam	Colisiones	Pasos adicionales
15000	9970	2362037
15013	3221	9896
19997	2415	4663
20005	6399	14076
24989	2034	35557
25000	9979	12605708

Cuadro 3: k = 500

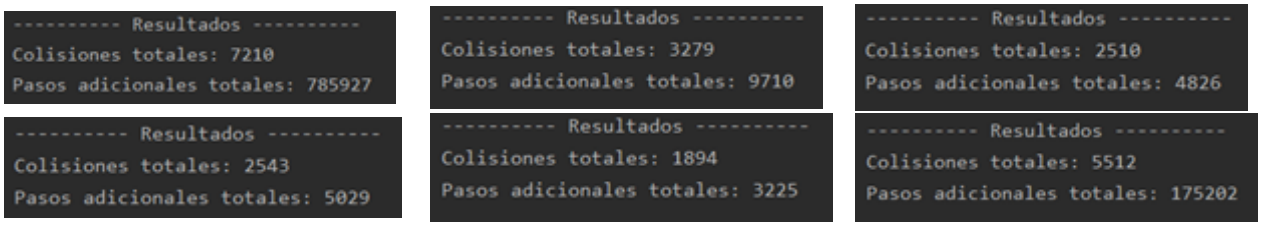


Figura 5: De izquierda a derecha, resultados para tamaños 15000, 15013, 19997, 20005, 24989 y 25000, con k = 499

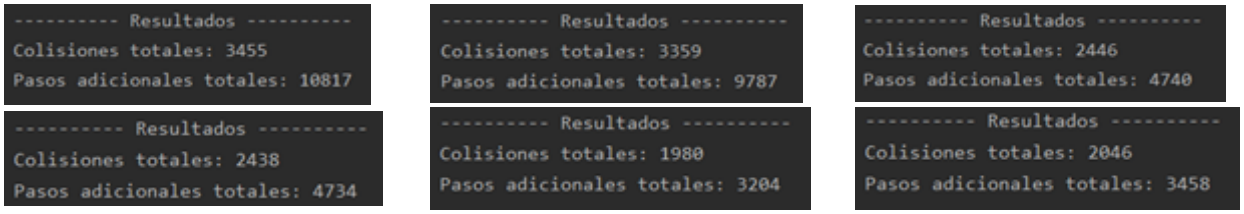


Figura 6: De izquierda a derecha, resultados para tamaños 15000, 15013, 19997, 20005, 24989 y 25000, con k = 1009

Tam	Colisiones	Pasos adicionales
15000	7210	785927
15013	3279	9710
19997	2510	4826
20005	2543	5029
24989	1894	3225
25000	5512	175202

Cuadro 4: k = 499

Tam	Colisiones	Pasos adicionales
15000	3455	10817
15013	3359	9787
19997	2446	4740
20005	2438	4734
24989	1980	3204
25000	2046	3458

Cuadro 5: k = 1009

De las figuras se deduce que el peor valor de k de entre los 3 elegidos es 500, y que entre 499 y 1009, 1009 es algo mejor.

Esta pauta se debe a que 499 y 1009 son primos, coprimos con todos los valores elegidos para el tamaño de la tabla, de forma que Hash 3 da valores separados entre sí. En cambio, 500 no solo no es primo, sino que tiene la siguiente descomposición en primos:

$$500 = 2^2 * 5^3, \tag{6}$$

muy similar a la de 15000 y 25000, que son los tamaños que peores resultados ofrecen. 20005, aunque también es divisible por 5, difiere más en su descomposición, por lo que no debe resultar sorprendente que no sea tan mala elección como los dos anteriores.

El motivo es que las potencias de 500 irán siendo canceladas al realizar los módulos en Hash 3, dejando de aportar diversidad a la generación, y las posiciones que se vayan calculando tendrán

una similitud mayor de la que cabría esperarse de no tener en cuenta este hecho.

Las estadísticas de 499 y 1009 no se alejan demasiado la una de la otra, pero al ser 1009 mayor que el primero, es comprensible que dé resultados algo mejores. Como base, generará números mucho mayores, por lo que habrá que efectuar el módulo en más ocasiones y lo obtenido con este será más dispar.

Otras diferencias en los resultados tienen que ver con los razonamientos desarrollados en el apartado 2.

3.3. Encadenamiento - Comparación de los resultados

3.3.1. Hash 1

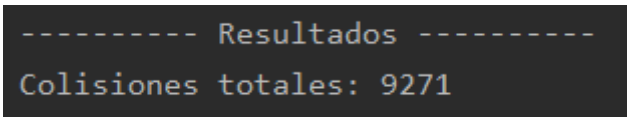


Figura 7: Resultado de todas las pruebas para Hash 1 con cualquier tamaño superior a 1650, aproximadamente

El total de colisiones para encadenamiento es algo menor que en recolocación, porque en el último, todos los elementos se colocan en una posición de la tabla, sea la que les correspondía originalmente o no, reduciendo la posibilidad de que el índice devuelto por Hash 1 para un dato posterior no se haya utilizado ya. En cambio, con encadenamiento, como todos los elementos van en una lista correspondiente al verdadero resultado de Hash 1 para los mismos, no ocurre este problema. Una lista va a permanecer vacía hasta que su ubicación corresponda con una clave dada, por lo esta no tendrá una colisiones. Tal fenómeno implica que hay un número ligeramente menor de colisiones.

3.3.2. Hash 2

Se remite a los resultados del apartado 2.

3.3.3. Hash 3

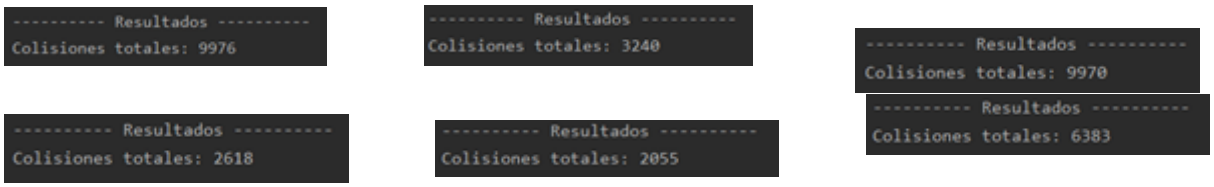


Figura 8: De izquierda a derecha, resultados para tamaños 12000, 12007, 15000, 15013, 19997 y 20005, con k = 500

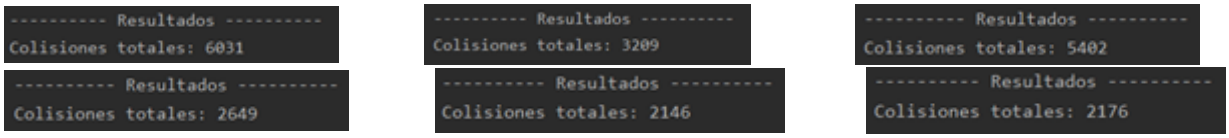


Figura 9: De izquierda a derecha, resultados para tamaños 12000, 12007, 15000, 15013, 19997 y 20005, con k = 499

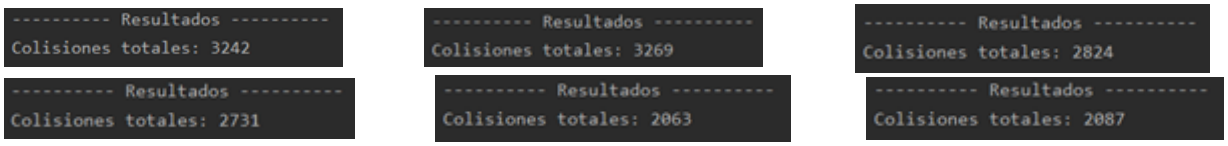


Figura 10: De izquierda a derecha, resultados para tamaños 12000, 12007, 15000, 15013, 19997 y 20005, con k = 1009

Tam	**Colisiones**
12000	9976
12007	3240
15000	9970
15013	2618
19997	2055
20005	6383

Cuadro 6: k = 500

Tam	**Colisiones**
12000	6031
12007	3209
15000	5402
15013	2649
19997	2146
20005	2176

Cuadro 7: k = 499

Tam	**Colisiones**
Tam	Colisiones
12000	3242
12007	3269
15000	2824
15013	2731
19997	2063
20005	2087

Cuadro 8: k = 1009

Se puede razonar de manera análoga a la expuesta para recolocación, teniendo en cuenta que habrá menos colisiones.

3.4. Variación en la clave utilizada

Queremos examinar ahora qué alteración en la eficiencia supone tomar otro campo como clave. En concreto, se empleará el correo electrónico de los usuarios, que será lo que se utilizará en las funciones hash para calcular las posiciones donde asignar los datos.

3.4.1. Recolocación simple con correo electrónico

A continuación aparecen las nuevas estadísticas:

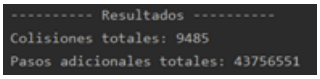


Figura 11: Resultado para todos los tamaños mayores de 10000 con Hash 1

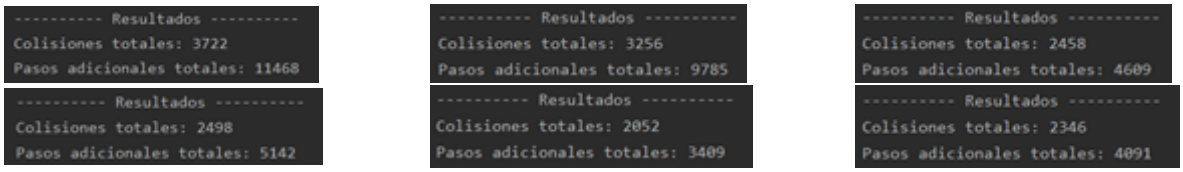


Figura 12: De izquierda a derecha, resultados para tamaños 15000, 15013, 19997, 20005, 24989 y 25000, con Hash 2

Tam	Colisiones	Pasos adicionales
15000	3722	11468
15013	3256	9785
19997	2458	4609
20005	2498	5142
24989	2052	3409
25000	2346	4091

Cuadro 9: Hash 2

Ya con Hash 1 se observa una leve mejoría en el número de colisiones, pero sin duda donde verdaderamente se nota el efecto del cambio es en Hash 2. Si antes Tam = 15000 y Tam = 25000 daban 8136 y 7010 colisiones, respectivamente, ahora les corresponden 3722 y 2346. De igual forma, 24989, el mejor tamaño probado, reduce sus conflictos de 2499 a 2052, y el resto se encuentran en tesituras análogas.

La pregunta que se plantea, pues, es cuál es la causa de dicha diferencia. Es decir, en qué aspectos es mejor el correo que el nick como clave.

Bien, ante todo, tenemos ya una ampliación en el tamaño de la clave. El nick tenía como máximo 20 caracteres de longitud, de los que decíamos que habitualmente eran desaprovechados varios. En comparación, el correo tiene el triple de longitud máxima. De nuevo, casi la mitad de esos caracteres no se utilizan, pero calibrando los jugadores del fichero de texto, podemos aventurar una media de 30 caracteres, muy superior a los 10 que suponíamos para los nicks.

Este hecho afecta positivamente a las 3 funciones hash, dando más diversidad a los resultados generados.

En el caso de Hash 1, hay más valores ASCII para sumar. No obstante, aunque es cierto que esto influye positivamente en el resultado (se alcanzan valores más altos y más diferentes entre sí), al no calcularse el módulo por el tamaño después de cada suma, sino solo después de completarse todas, resta relevancia a los sumandos añadidos. Puede haber más irregularidad, es cierto, pero no podemos asegurar que esta aumente demasiado, porque el módulo puede devolver el mismo valor para números que difieren ampliamente entre sí. Esto es consecuente con lo observado en los experimentos: las colisiones y los pasos adicionales de Hash 1 han disminuido, pero siguen manteniendo

un orden similar al que había con el nick.

En cuanto a Hash 2, aquí sí cobra importancia que se realice el módulo después de cada suma. Como hay un mayor número de operaciones, las progresivas sucesiones que se vayan formando irán divergiendo cada vez más entre sí. Esto mismo es aplicable a Hash 3 (comprobaremos ahora si ocurre lo esperado).

Por otro lado, es cierto que no todo es perfecto para esta opción de clave. Hay muchos más caracteres repetidos: puntos, la '@' y el dominio del correo electrónico, lo que reduce el peso del mayor tamaño. Sin embargo, que aparezca el nombre y los dos apellidos completos no es en ningún caso despreciable, como ya hemos visto por los anteriores resultados.

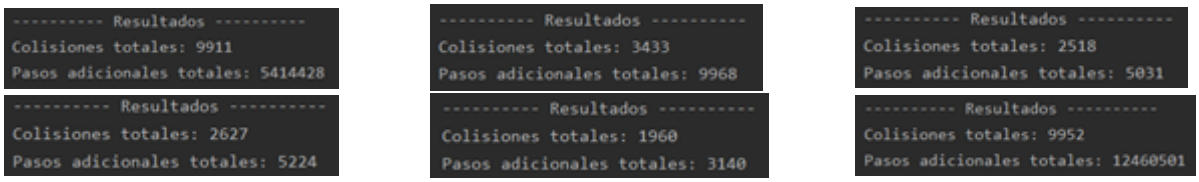


Figura 13: De izquierda a derecha, resultados para tamaños 15000, 15013, 19997, 20005, 24989 y 25000, con Hash 3 y k = 500

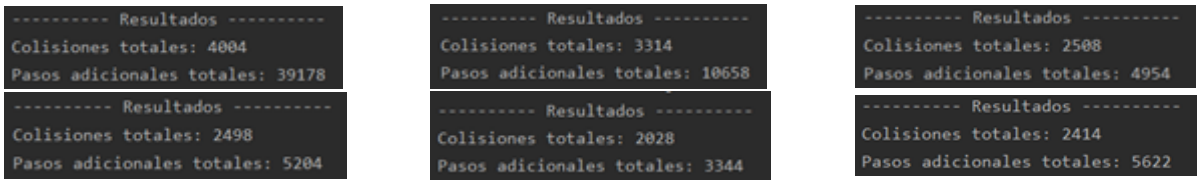


Figura 14: De izquierda a derecha, resultados para tamaños 15000, 15013, 19997, 20005, 24989 y 25000, con Hash 3 y k = 499

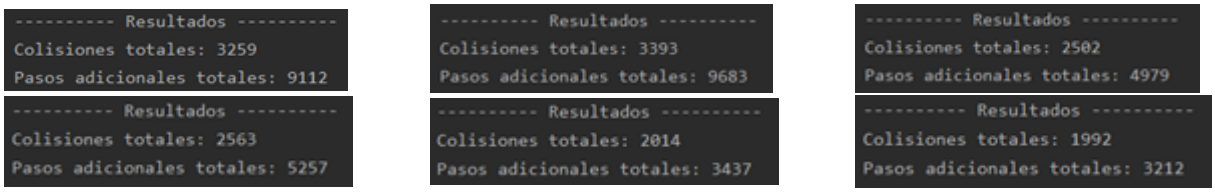


Figura 15: De izquierda a derecha, resultados para tamaños 15000, 15013, 19997, 20005, 24989 y 25000, con Hash 3 y k = 1009

Tam	Colisiones	Pasos adicionales
15000	9911	5414428
15013	3433	9968
19997	2518	5031
20005	2627	5224
24989	1960	3140
25000	9952	1246051

Cuadro 10: Hash 3, k = 500

Confirmamos así la hipótesis de que el correo también iba a funcionar bien con Hash 3 (por el mismo principio que Hash 2). Obsérvense el bajo número de colisiones y pasos adicionales incluso

Tam	Colisiones	Pasos adicionales
15000	4004	39178
15013	3314	10658
19997	2508	4954
20005	2498	5204
24989	2028	3344
25000	2414	5622

Cuadro 11: Hash 2

Tam	Colisiones	Pasos adicionales
15000	3259	9112
15013	3393	9683
19997	2502	4979
20005	2563	5257
24989	2014	3437
25000	1992	3212

Cuadro 12: Hash 2

en casos de tamaños redondos como 15000 o 20005. Es cierto que para k = 1009 los resultados son similares (si bien el correo sigue mejorando al nick en tamaños como 25000 o 15000), probablemente porque al ser un valor de k tan grande los efectos de la elección de la clave empiezan a notarse menos. Aun así, sí debemos considerar el hecho de que el correo requiere más espacio de almacenamiento que el nick y, por tanto, llegados a este punto podemos plantearnos utilizar este último para ahorrar memoria.

3.4.2. Encadenamiento con correo electrónico

La explicación de los resultados es análoga al caso de recolocación. Se muestran a continuación:

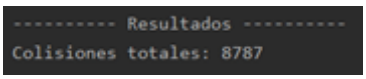


Figura 16: Resultado con Hash 1 para todos los tamaños superiores a 1650, aproximadamente

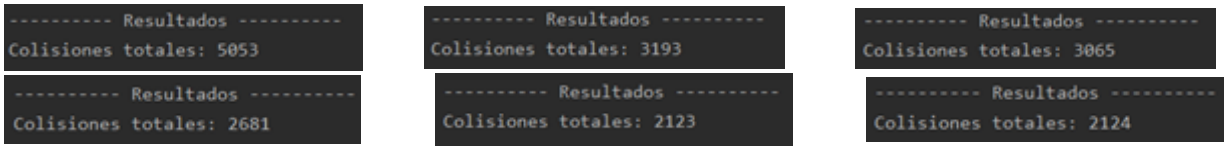


Figura 17: De izquierda a derecha, resultados para tamaños 12000, 12007, 15000, 15013, 19997 y 20005, con Hash 2

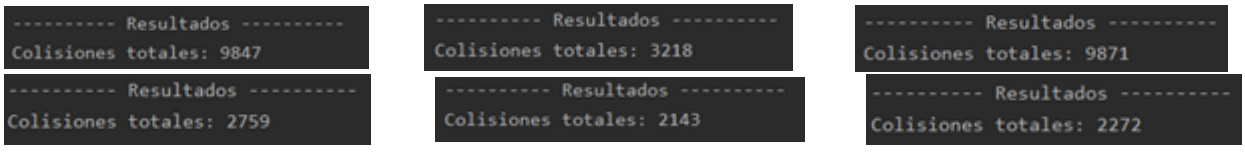


Figura 18: De izquierda a derecha, resultados para tamaños 12000, 12007, 15000, 15013, 19997 y 20005, con Hash 3 y k = 500

Tam	Colisiones
12000	5063
12007	3193
15000	3065
15013	2681
19997	2123
20005	2124

Cuadro 13: Hash 2

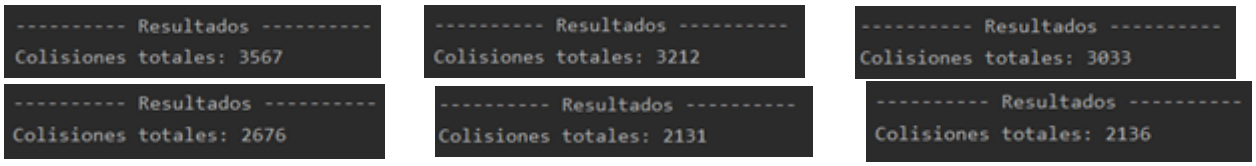


Figura 19: De izquierda a derecha, resultados para tamaños 12000, 12007, 15000, 15013, 19997 y 20005, con Hash 3 y k = 499

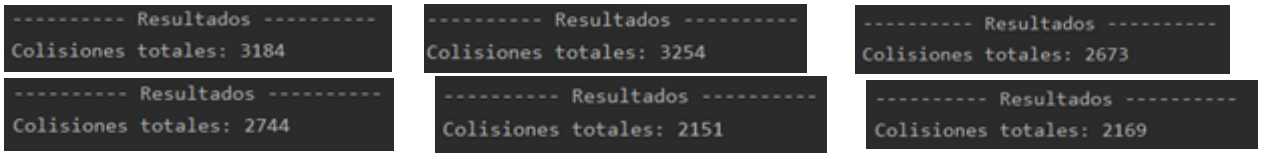


Figura 20: De izquierda a derecha, resultados para tamaños 12000, 12007, 15000, 15013, 19997 y 20005, con Hash 3 y k = 1009

Tam	Colisiones
12000	9847
12007	3218
15000	9871
15013	2759
19997	2143
20005	2272

Cuadro 14: Hash 3, k = 500

Tam	Colisiones
12000	3567
12007	3212
15000	3033
15013	2676
19997	2131
20005	2136

Cuadro 15: Hash 3, k = 499

De nuevo, se vuelve a notar significativamente el efecto en Hash 2 y en Hash 3 con k = 500 y k = 499, y de manera más suave en Hash 1 y en Hash 3 con k = 1009. La explicación es análoga a la de recolocación y, de nuevo, podemos volver a considerar lo planteado: utilizar el nick cuando el resto de elecciones nos aseguran un resultado óptimo para ahorrar memoria en el caso de k = 1009 con Hash 3, por ejemplo.

Tam	Colisiones
12000	3184
12007	3254
15000	2673
15013	2744
19997	2151
20005	2169

Cuadro 16: Hash 3, k = 1009

4. Influencia de la elección de la estrategia de recolocación

Para empezar, retornando al nick como clave (lo predeterminado en este proyecto), busquemos qué combinaciones de variables son óptimas.

Con respecto a la función hash, fijaremos Hash 3. De las 3 posibles, podemos descartar completamente Hash 1 por sus deplorables resultados. Entre Hash 2 y Hash 3 la elección es más delicada, pero si tomamos Hash 3 con un buen valor de k, podremos mejorar al primero.

Por lo concluido en el apartado 3, y comparando los resultados de Hash 2 y Hash 3, si tomamos k = 499 o k = 1009, nos pondremos en una mejor situación de partida. Como de entre los dos valores, k = 1009 es el más óptimo, utilizaremos este.

Asimismo, establezcamos un tamaño para la tabla. Volviendo al apartado 2, decíamos que aquellos Tam superiores o iguales a 20000 permitían que el factor de carga, L, valiese menos o lo mismo que 1/2, de forma que ya tenemos una primera restricción, que de cumplirse dará una media de número de ensayos constante. Veíamos también que los mejores valores para Tam eran los primos, porque realizar el módulo por ellos repartía los jugadores más dispersamente. Así, podríamos escoger Tam = 24989.

Para poder cotejar lo que obtengamos, realizaremos también las pruebas de este apartado con Tam = 20005, un valor no primo que, no obstante, daba resultados medianamente adecuados (si bien ampliamente mejorables) por las características de su descomposición.

4.1. Variación en la estrategia

Dependiendo del valor a que utilicemos, estaremos empleando un tipo de estrategia de recolocación u otra.

Así, $a = 1$ es la que venimos aplicando hasta el momento: recolocación simple. Cuando una posición está ocupada, se comprueba la inmediatamente siguiente, y se continúa hasta encontrar alguna vacía o borrada. El problema que acarrea este método es que si hay un gran número de colisiones, al colocarse los elementos conflictivos seguidos, se formarán grandes bloques que provocarán un elevado número de pasos adicionales para la inserción en esas posiciones.

Si a es un número entre 1 y Tam - 1, hablamos de recolocación lineal (la simple es un caso particular de esta). Tomando $a \neq 1$, nos estaremos saltando lugares de la tabla. Por ejemplo, si $a = 3$, solo se pueden mirar las posiciones congruentes en módulo 3 para un dato dato.

El problema que surge en la recolocación lineal es que, al restringirnos a valores congruentes en módulo a, no vamos a poder comprobar toda la tabla, y podríamos no llegar a encontrar un hueco libre, cuando en realidad este sí existe. Además, de forma similar al caso simple, si a es pequeño es probable que se formen bloques de jugadores a lo largo de las ubicaciones congruentes entre sí.

Otra opción es la recolocación cuadrática, en la que se comprueban posiciones siguiendo una sucesión de números cuadrados: 1, 4, 9, 16, 25... Este método no requiere de la variable a, sino que únicamente utiliza un contador que registra el número del intento actual, i. Dado i, su valor

asociado en la sucesión anterior será i^2.

Con recolocación cuadrática también se puede llegar a dar una situación en la que no se encuentre espacio libre, cuando en realidad este sí existe, pero manteniendo un factor de carga menor o igual que 1/2 esto debería darse solamente si la tabla está ya muy llena. Además, la recolocación cuadrática disminuye notablemente la probabilidad de que se formen bloques.

Empecemos pues con las pruebas.

4.1.1. Recolocación simple

Se remite a los resultados del apartado 3.

4.1.2. Recolocación lineal

Pongamos en primer lugar a = 5, a = 4001 y a = 11. Podemos aventurar que 20005 ofrecerá malos resultados con los dos primeros, al ser divisible por ellos. En efecto,

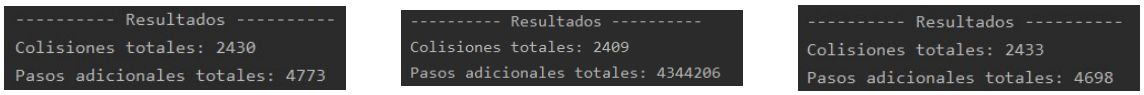


Figura 21: Con Tam = 20005, resultados para a = 5, a = 40001 y a = 11

a	Colisiones	Pasos adicionales
5	2430	4773
4001	2409	4344206
11	2433	4698

Cuadro 17: Tam = 20005, recolocación lineal

Antes de nada, es importante hacer notar que la diferencia entre los a tomados se debe buscar en el número de pasos adicionales, y no en las colisiones, pues la decisión sobre a afecta en el caso de que se haya una colisión y se tenga que aplicar recolocamiento. Si bien es cierto que un mayor número de recolocaciones formará bloques y estos a su vez darán lugar a nuevas colisiones, no es este el verdadero objetivo del apartado. Lo que debemos buscar son desemejanzas en la resolución de conflictos, esto es, en las operaciones adicionales y en el hecho de que se encuentre o no posición libre.

Entre los 3 valores, el que peor ha funcionado es, sin duda, 4001. El motivo es que, por un lado, es divisor de 20005 y, por otro, se trata de un valor muy grande, que tiene pocos espacios posibles en la tabla para cada uno de los 4001 módulos. Esto explica que durante la ejecución haya bastantes valores para los que no se ha encontrado un sitio libre, lo cual implica a su vez que, por la forma en la que se han establecido los contadores, el número de pasos adicionales será considerable.

a = 5, a pesar de que también es divisor de 20005, ofrece resultados bastante mejores, principalmente porque el hecho de que sea tan pequeño le da el suficiente margen de ubicaciones para cada módulo en cuanto a posiciones disponibles. Es más, para esta opción, se ha encontrado un lugar para todos los datos.

Queda por analizar a = 11, primo y coprimo con 20005. Supera ligeramente a a = 5, presumiblemente por su coprimidad. Es esperable que hubiera más disparidad en el caso de que para algún dato no se encontrase espacio con a = 5. En cualquier caso, queda claro que, a pesar de la cercanía de 5 y 11, la relación matemática con el tamaño es relevante.

Estudiemos ahora qué ocurre con el tamaño que consideramos "óptimo", 24989.

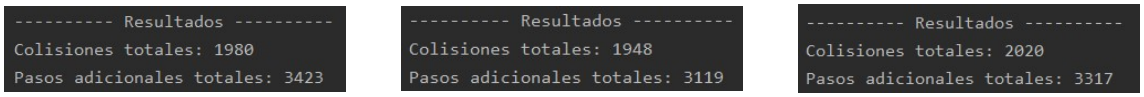


Figura 22: Con Tam = 24989, resultados para a = 3, a = 29 y a = 4001

a	Colisiones	Pasos adicionales
3	1980	3423
29	1948	3119
4001	2020	3317

Cuadro 18: Tam = 24989, recolocación lineal

Ahora, los valores de a son 3, 29 y 4001, todos coprimos con la longitud de la tabla. Se puede ver la contraposición con 20005: ninguno de ellos es significativamente mejor que los otros, sino que sus colisiones y pasos adicionales son similares. Incluso en el caso de un a grande, como es 4001, este no se desvía de los resultados esperados. Tampoco un a pequeño como 3 lo hace, y tan solo da unos 100 pasos adicionales más que 4001. En ningún momento de las 3 pruebas ha quedado un dato sin colocar en la tabla.

En conclusión, deducimos que las mejores opciones para a son aquellos valores coprimos con el tamaño y lo suficientemente grandes como para que el número de módulos diferentes permita repartir ecuánimemente los jugadores.

4.1.3. Recolocación cuadrática

Figura 23: Recolocación cuadrática con Tam = 20005 y Tam = 24989, respectivamente

a	Colisiones	Pasos adicionales
20005	2423	4083
24989	1976	2945

Cuadro 19: Recolocación cuadrática

Utilizar recolocación cuadrática ha superado todo lo probado hasta el momento para cada uno de los dos tamaños en cuanto a pasos adicionales. Utilizar este proceso puede considerarse más ”aleatorio” que seguir una estrategia lineal, donde se agrupan los valores congruentes en conjuntos aislados. No obstante, no deberíamos olvidar que estamos trabajando con lo que en anteriores apartados nos dio resultados mínimos. Con otros peores, arriesgaríamos que hubiera jugadores sin colocar cuando la tabla está casi llena (es lo que ocurre probando, por ejemplo, Tam = 24000 y k = 500).

Tampoco se debe ignorar la mayor complejidad algorítmica de la estrategia cuadrática, que se hará patente especialmente en aquellos pasos en los que deba probar repetidas veces para encontrar ubicaciones libres, o donde no las llegue a encontrar.

Inferimos, por tanto, que es preferible usar recolocación cuadrática únicamente cuando estamos seguros de que el resto de variables tienen una buena sinergia entre sí, y que no provocarán demasiadas colisiones u operaciones extra de por sí.

5. Estimación de la eficiencia en el acceso a los datos (búsqueda)

En este apartado mediremos el rendimiento de la búsqueda de datos en la tabla. Para ello, una vez insertados los 10000 jugadores, se buscarán todos ellos. Para los resultados se tendrán en cuenta tanto el tiempo necesario para la búsqueda, como el número de operaciones realizadas en la misma.

Se utilizarán diferentes implementaciones, variando el tamaño de la tabla, la estrategia de búsqueda y la función hash. El proceso mediante el cual se inserten los datos se verá determinado por el que se esté utilizando para la búsqueda (deben coincidir porque además, en caso contrario, podría no encontrarse el jugador que se quiera localizar).

El procedimiento a seguir será el siguiente: tras modificar todos los valores necesarios respectivos a la prueba, se realizará una ejecución en la que únicamente se cronometrará el tiempo de búsqueda, con el objetivo de que otros controles no afecten. Después, en una segunda ejecución, se constatará el número de operaciones totales.

5.1. Recolocación simple

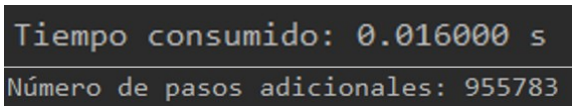


Figura 24: Recolocación simple, Hash 1, todos los tamaños a partir de 10000. El tiempo consumido a veces puede ser de 0.015000 s

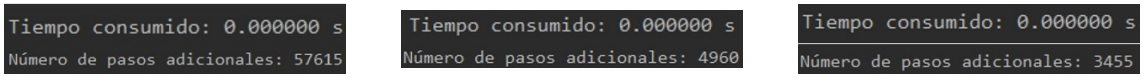


Figura 25: Recolocación simple, Hash 2. De izquierda a derecha, tamaños 15000, 20005, 24989

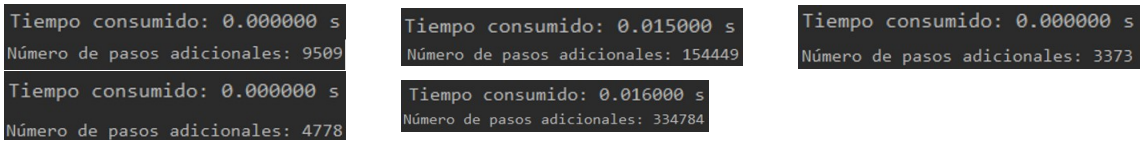


Figura 26: Recolocación simple, Hash 3. De izquierda a derecha, las dos primeras usan k = 499 y tienen Tam = 15013 y Tam = 11000, respectivamente. La última de la primera línea usa k = 1009 y tamaño 25000. Las de la segunda línea tienen k = 500 y Tam = 19997 y 20000

5.2. Recolocación cuadrática

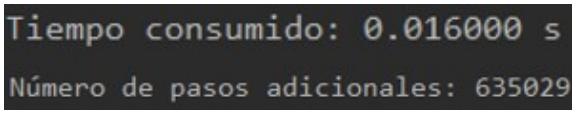


Figura 27: Recolocación cuadrática, Hash 1, todos los tamaños a partir de 10000

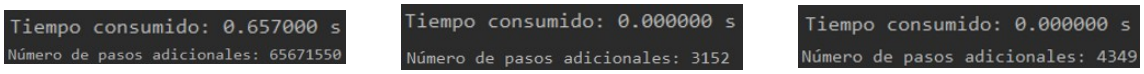


Figura 28: Recolocación cuadrática, Hash 2. De izquierda a derecha, tamaños 15000, 20005, 24989. En el primer caso, muchos elementos quedaron sin colocar

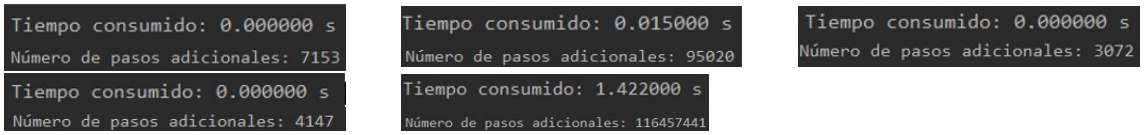


Figura 29: Recolocación cuadrática, Hash 3. De izquierda a derecha, las dos primeras usan k = 499 y tienen Tam = 15013 y Tam = 11000, respectivamente. La última de la primera línea usa k = 1009 y tamaño 25000. Las de la segunda línea tienen k = 500 y Tam = 19997 y 20000. En esta última prueba, muchos elementos quedaron sin colocar

5.3. Encadenamiento

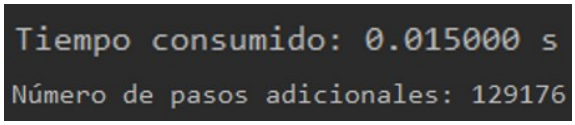


Figura 30: Encadenamiento, Hash 1, todos los tamaños superiores a 1650, aproximadamente

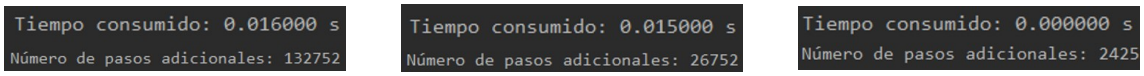


Figura 31: Encadenamiento, Hash 2. De izquierda a derecha, tamaños 1200, 15000, 19997

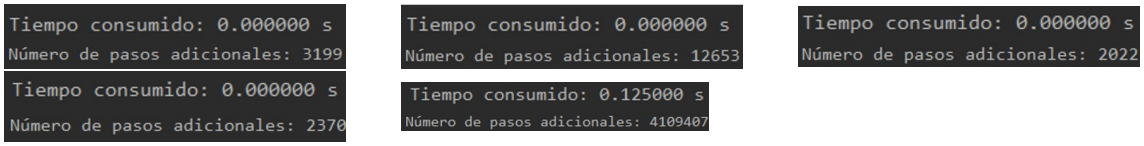


Figura 32: Encadenamiento, Hash 3. De izquierda a derecha, las dos primeras usan k = 499 y tienen Tam = 15013 y Tam = 11000, respectivamente. La última de la primera línea usa k = 1009 y tamaño 25000. Las de la segunda línea tienen k = 500 y Tam = 19997 y 25000

5.4. Análisis

En general, la estrategia de encadenamiento es la que obtiene un mayor rendimiento. Es comprensible: el hecho de restringir el espacio de búsqueda a un subconjunto concreto, una lista, permite reducir el número de elementos a comprobar. Incluso con Hash 1 o una mala elección para Hash 3 (por ejemplo, k = 1009 y Tam = 25000), los tiempos y pasos adicionales no se disparan demasiado.

Sin embargo, no se debe despreciar la importancia del factor de carga. Si este es demasiado elevado, los datos no se estarán repartiendo adecuadamente, con lo que la longitud de las listas será demasiado elevada. Ello ocasionará problemas al recorrerlas, pues están conectadas a través de punteros. Es decir, que una mala gestión del espacio implica una eficiencia todavía peor.

Por otro lado, la recolocación cuadrática puede no parecer un opción ineficiente y, efectivamente, en la mayoría de los casos supera a la recoloción simple ampliamente. Se llega a alcanzar un número de pasos adicionales muy bajo, del orden de 3000 o 4000, y en bastantes ocasiones el algoritmo tarda un tiempo tan reducido que se presenta como 0 segundos. Aun así, existe un riesgo con respecto a aquellos casos en los que, sencillamente, el funcionamiento no es correcto. Así, nos encontramos con que Hash 2 con Tam = 15000 no encuentra a todos los jugadores, y esto ocasiona un incremento relevante en la duración de la búsqueda: se llega hasta los 0.657 segundos. La misma situación se da para un valor que desde un principio se sabía que no iba a ofrecer buenos resultados: 20000. En el caso de Hash 3, incluso a pesar de que se usa k = 1009, se obtiene un número de operaciones extra exorbitado, con el peor tiempo registrado: 1.422 segundos.

En cuanto a la recolocación simple, esta tiene un comportamiento consistente a lo largo de todos los experimentos. No es demasiado eficaz, porque incluso en las mejores coyunturas suele quedar en peor lugar que los otros dos métodos, pero sí es preferible en casos en los que recolocación cuadrática se dispara por no encontrar elementos (con la simple no se nos presentaría este problema).

En cualquiera de las estrategias se puede comprobar un punto común: los incrementos en los tiempos están estrechamente relacionados con el número de pasos adicionales. Estos, a su vez, dependen de la cantidad de colisiones que hayan tenido lugar. Por consiguiente, lo que verdaderamente marca la eficiencia de una tabla hash es la cantidad media de colisiones que provoca, pues este es el factor origen que determina el resto de parámetros que resultan interesantes en las mediciones.

Es preferible, por tanto, priorizar aquellas tácticas que garanticen una buena sinergia entre la función hash y el tamaño de la tabla, antes siquiera de empezar a plantear qué se utilizará en la resolución de conflictos.

Respecto a esta, es necesario llegar a una solución de compromiso. ¿Hasta qué punto merece la pena el gasto extra de memoria en las listas del encadenamiento con respecto a la mayor lentitud general de la recolocación? En principio, se tiene que dar preferencia al encadenamiento, que no solo soporta factores de carga mayores (hasta 0.75, en contraposición al 0.5 óptimo de la recolocación), sino que su lentitud en función del mismo crece más despacio, al solo buscarse en una lista, en lugar de agrupar todas los datos juntos. De igual manera, para L altos métodos como la relocación cuadrática no garantizan siquiera que se encuentren e inserten todos los elementos.

Además, con encadenamiento se puede llegar a tener $L > 1$, pues las listas tienen un tamaño indefinido, no dependiente de la longitud de la tabla. En cambio, con recolocación resultaría imposible colocar más elementos que Tam.

Aun así, hay un aspecto en el que sí puede resultar preferible la recolocación: si el peso de los datos que se guardan en la tabla es grande, las listas llegarían a ocupar un espacio excesivo en memoria. Entonces sí merece la pena sacrificar una pérdida de velocidad a favor de aligerar el gasto en almacenamiento.

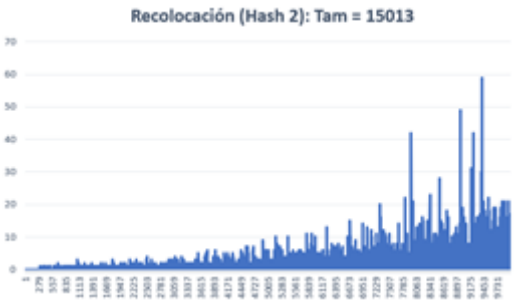
Para el problema que nos ocupa, llevado casos extremos, podríamos preferir encadenamiento si guardamos solo la clave como el nick y el nombre de cada jugador, y recolocación si tomamos nombre, los dos apellidos, la clave y el correo. No obstante, la diferencia sería mayor en situaciones con aumento mayor de una situación a otra.

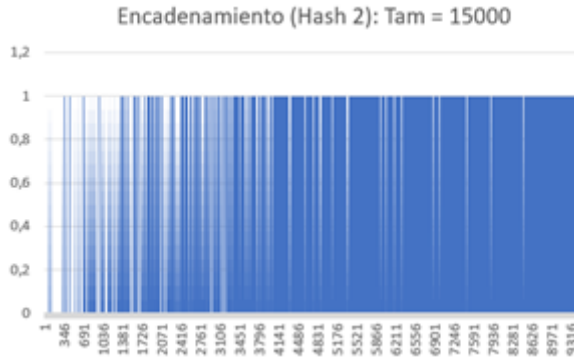
6. Anexo

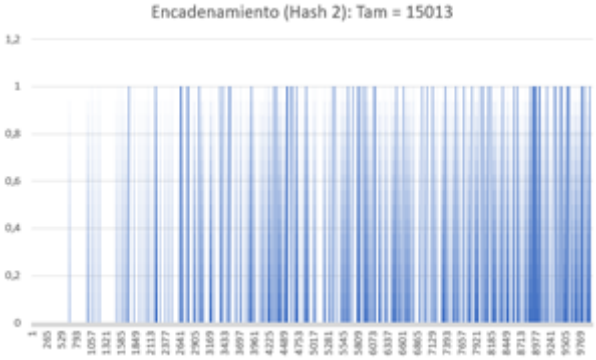
A continuación aparecen algunas gráficas que resultaron ilustrativas a la hora de comprender la evolución de los pasos adicionales con respecto a la inserción de sucesivos elementos para distintas combinaciones.

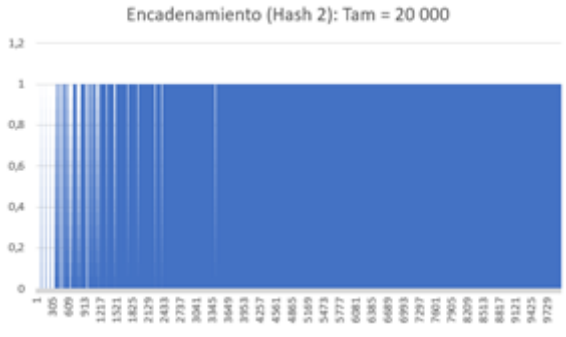
En el eje X aparece el índice del elemento colocado y en el eje Y, sus correspondientes pasos adicionales, en el caso de recolocación, o su correspondiente colisión (que puede valer 0 o 1), en el caso de encadenamiento.

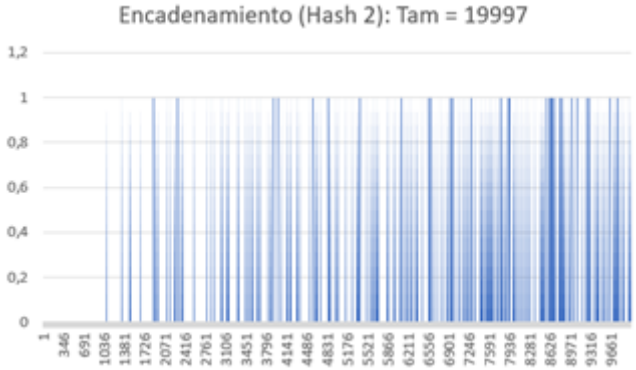
Todas las gráficas se han creado utilizado el nick como clave.

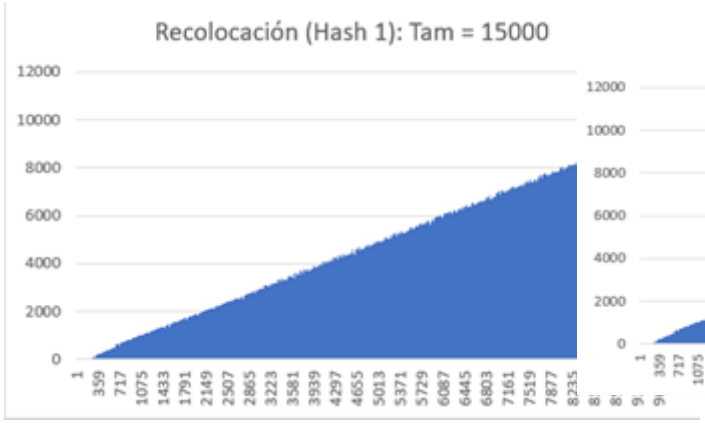


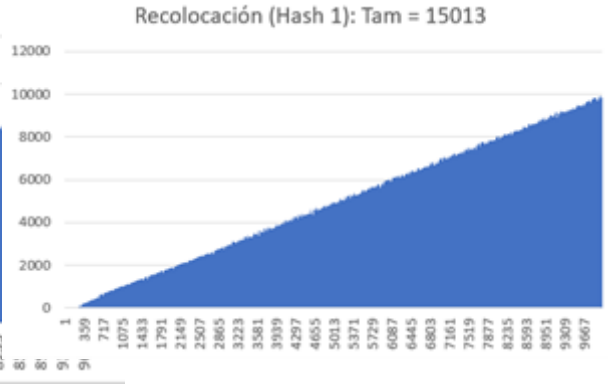


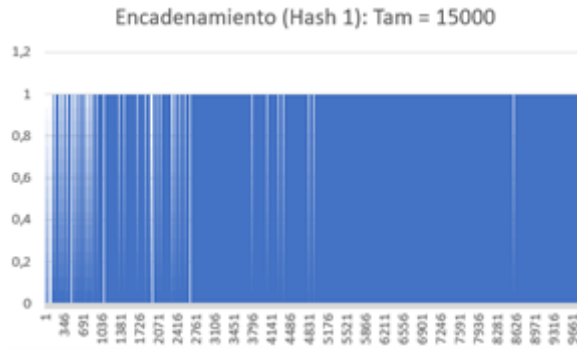


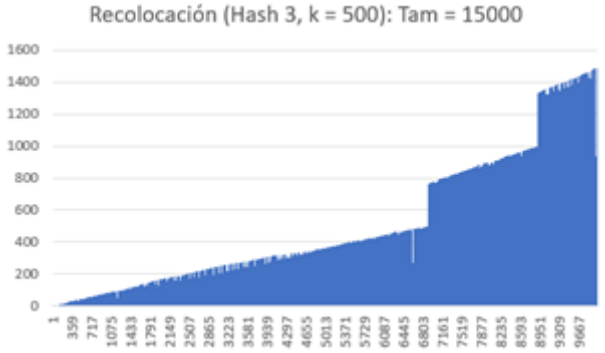


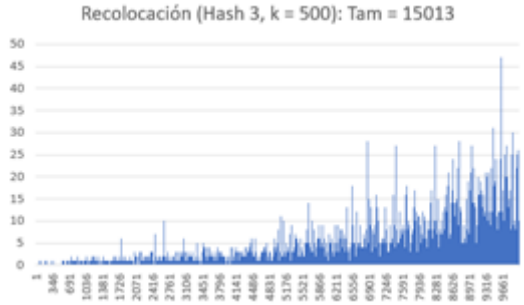


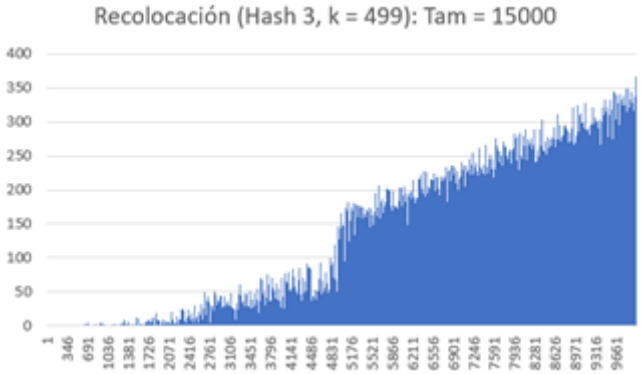

Recolocación (Hash 1): Tam = 15000

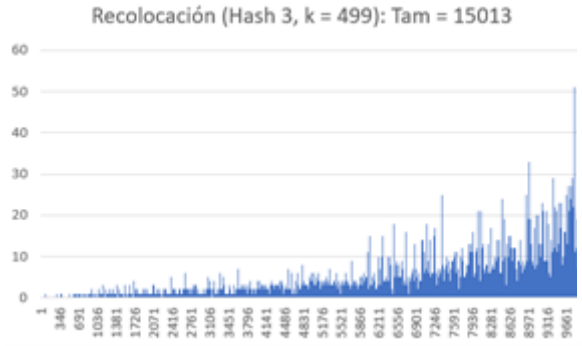

Recolocación (Hash 1): Tam = 15013

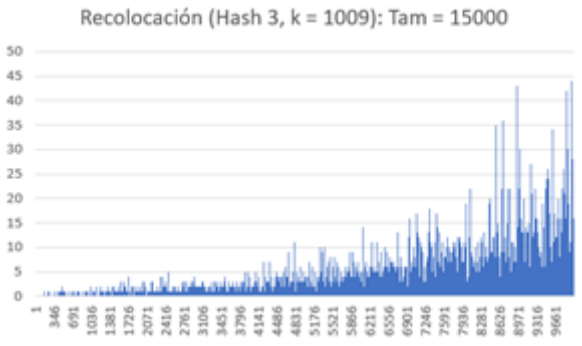

Encadenamiento (Hash 1): Tam = 15000

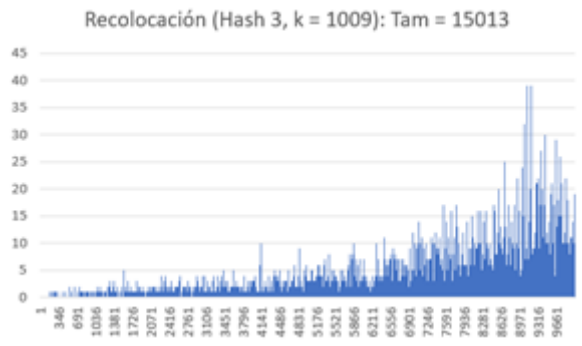

Recolocación (Hash 3, k = 500): Tam = 15000

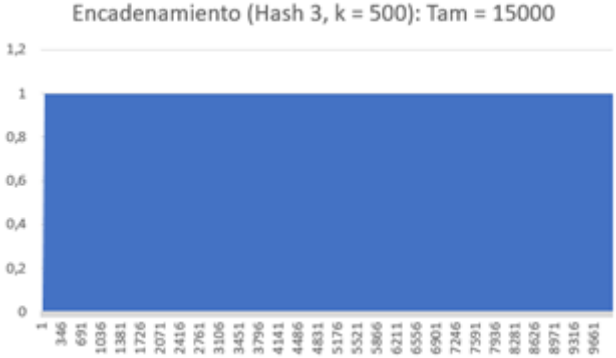

Recolocación (Hash 3, k = 500): Tam = 15013


Recolocación (Hash 3, k = 499): Tam = 15000


Recolocación (Hash 3, k = 499): Tam = 15013


Recolocación (Hash 3, k = 1009): Tam = 15000


Recolocación (Hash 3, k = 1009): Tam = 15013


Encadenamiento (Hash 3, k = 500): Tam = 15000