

Informe Tablas Hash

Quintáns González, Iván
Novoa González, Christian

USC, Ingeniería Informática
Algoritmos y Estructura de Datos, 2022/2023
Fecha: 17 de noviembre de 2022

ÍNDICE

0. Introducción.	4
0.0. Tabla Hash	4
0.1. Definición de Colisión.	4
0.2. Método de Recolocación.	4
0.2.1. Recolocación simple	4
0.2.1. Recolocación lineal	5
0.2.1. Recolocación cuadrática	5
0.3. Método de Encadenamiento.	5
0.4. Implementación de contadores.	5
0.4.1. Método de encadenamiento	5
0.4.2. Método de recolocación	8
1. Influencia de N para la Inserción.	12
1.0 Tabla de Datos	12
1.1. N Primo vs. No-Primo.	12
1.2. Factor de Carga Óptimo vs. No Óptimo.	13
1.3. Conclusión.	13
2. Influencia de la Función Hash (1-2-3) y de la clave en la Inserción.	14
2.0 Explicacion Funcion Hash	14
2.0.0 Función Hash 1	14
2.0.1 Función Hash 2	14
2.1 Análisis de Resultados	15
2.0.0. Hash 1.	15
2.0.1. Análisis Clave = Nickname	15
2.0.2. Análisis Clave = Correo	16
3. Influencia de estrategia de recolocación en Inserción.	18
3.0. Explicación de Tamaño Óptimo.	18
3.1. Recolocación Lineal vs. Cuadrática	18
3.2. Análisis de Datos	18
4. Estimación de la Eficiencia en el acceso a datos (búsqueda).	19
4.0. Métodos de Búsqueda en Colisiones.	19
4.0.0. Método de Recolocación	19
4.0.1. Método de Encadenamiento	19
4.1. Análisis de Datos	19
4.2. Conclusión	20

0. Introducción.

0.0. Tabla Hash

En primer lugar debemos tratar una explicación general sobre qué es una **Tabla hash** para poder introducirnos en el tema del informe.

Una tabla hash es una estructura de datos que nos facilita las operaciones de inserción, búsqueda y borrado. Estas operaciones se realizan en nuestra Tabla Hash en un tiempo constante en media, independientemente del número de datos.

A cada dato añadido se le aplica una **función Hash** y se le asigna una posición única en una tabla, con lo que la búsqueda, la inserción y el borrado son inmediatos: $O(1)$.

0.1. Definición de Colisión.

Una **colisión** es producto de, en una **Tabla Hash**, la correspondencia de dos o más claves con la misma componente de un vector.

Por ejemplo: consideremos que queremos almacenar en una **Tabla Hash** todos los trabajadores de una determinado oficina para lo cual usamos como *clave hash* la letra de su DNI. En esta situación puede ocurrir (y es muy probable que ocurra) que varias personas de la oficina compartan la misma letra del DNI dando a lugar a una **colisión**.

La resolución de colisiones es muy costosa por lo que siempre se deben de intentar preveer y/o minimizar en lo posible. Por esta razón es de vital importancia elegir un **tamaño**, **clave** y **función hash** óptimos para cada escenario.

Para resolver las colisiones, la complejidad de búsqueda, inserción y borrado deja de ser $O(1)$ y se le deben de aplicar uno de los siguientes métodos: **recolocación** o **encadenamiento**.

0.2. Método de Recolocación.

Aplicando el método de recolocación, cuando existe una colisión entre dos claves, se resuelve colocando el elemento en otra posición libre que encontremos. Para este método es recomendable mantener un **factor de carga** inferior a 0.5.

Dependiendo de la forma en la que coloquemos este elemento encontramos diferentes tipos.

0.2.1. Recolocación simple

INSERCIÓN: Al insertar una clave c , si la posición $h(c)$ está ocupada, se comprueban las posiciones siguientes: $h(c)+1(mod N)$, $h(c)+2(mod N)$, $h(c)+3(mod N)$,... hasta encontrar una libre, en la que se inserta c

BÚSQUEDA: Se repite el mismo proceso, es decir, se busca en $h(c)+1(mod N)$, $h(c)+2(mod N)$, $h(c)+3(mod N)$,... hasta encontrar la clave o llegar a una posición vacía, en cuyo caso se entiende que la clave buscada no está.

0.2.1. Recolocación lineal

Si h es la función hash, y a un número fijo entre **1 y $N-1$** , la recolocación lineal (para inserción o búsqueda) consiste en:

Al insertar una clave c , si la posición $h(c)$ está ocupada, se comprueban las posiciones siguientes: $h(c)+a(mod\ N), h(c)+2a(mod\ N), \dots$ hasta encontrar una libre, en la que se inserta c

0.2.1. Recolocación cuadrática

Si h es la función hash, la recolocación cuadrática (para inserción o búsqueda) consiste en:

Al insertar una clave c , si la posición $h(c)$ está ocupada, se comprueban las posiciones siguientes: $h(c)+1(mod\ N), h(c)+4(mod\ N), h(c)+9(mod\ N), \dots, h(c)+i^2(mod\ N), \dots$ hasta encontrar una libre, en la que se inserta c

0.3. Método de Encadenamiento.

Aplicando el método de encadenamiento, la función hash distribuye uniformemente las claves insertadas de forma que las colisión se resuelve colocando una **lista**¹ de longitud media n/N en cada posición del vector con todos los elementos que se deban de situar en esa dirección (*elementos colisionados*).

Mediante este método, la inserción en la *Tabla Hash* sigue teniendo complejidad constante porque los elementos se insertan al comienzo de la lista en la dirección que les corresponda a su *clave*. Para la búsqueda es necesario aplicar alguna estrategia algorítmica para recorrer la lista de *elementos colisionados*. En el peor de los casos esto implicaría recorrer $L (=n/N)$ elementos y en el mejor $(L+1)/2$ elementos por lo que para $n \leq N$ resultaría en tiempo constante.

Para aplicar un método de encadenamiento óptimo es imprescindible tener una función hash que reparta de la forma más equilibrada posible las claves ya que existe la posibilidad que todos los elementos se almacenarán en una única lista llevando a una búsqueda y eliminación de complejidad N . Por esta misma razón es recomendable mantener un **factor de carga** inferior a 0.75.

Es posible implementar el método de encadenamiento con árboles equilibrados (o incluso arrays dinámicos) pero es recomendable usar listas enlazadas por su flexibilidad y porque para el factor de carga recomendado, aplicar un árbol equilibrado no tiene una mejoría sustancial en la eficiencia del código pero sí en su complejidad a la hora de implementarlo.

0.4. Implementación de contadores.

Para realizar la implementación de los contadores en el código proporcionado por la coordinadora de la asignatura realizamos los siguientes cambios

0.4.1. Método de encadenamiento

El método de encadenamiento, en la **inserción**, cuando se produce una colisión tenemos una variable llamada ***nColisiones*** pasada por referencia que nos permite ir llevando la cuenta del número de colisiones llevadas a la hora de insertar.

```

void insercionArchivo(FILE *fp, TablaHash *t, unsigned int tipoFH, unsigned int K, int *nColisionesI) {
    TIPOELEMENTO paciente;
    if (fp) {
        fscanf( stream: fp, format: " %[^-] - %s - %s", paciente.nombre, paciente.alias, paciente.correo);
        while (!feof( stream: fp)) {
            //Modificar la función InsertarHash para que devuelva: 1 si hubo colisión,
            //0 en caso contrario e ir acumulando los valores en un contador nColisionesI
            //acumulo las colisiones que se producen en la función
            *nColisionesI+=InsertarHash(t, elemento: paciente, tipoFH, K);
            fscanf( stream: fp, format: " %[^-] - %s - %s", paciente.nombre, paciente.alias, paciente.correo);
        }
    } else {
        printf( format: "El archivo no ha podido abrirse\n");
    }
}

```

Como se puede apreciar la función **InsertarHash** nos devuelve 1 o 0 dependiendo de si se ha producido una colisión o no, y este valor lo vamos almacenando en nuestra variable **nColisiones** que será impresa por pantalla una vez acabada la inserción de datos a la tabla Hash.

En nuestra función InsertarHash:

```

int InsertarHash(TablaHash *t, TIPOELEMENTO elemento, unsigned int tipoFH, unsigned int K) {
    int pos = FuncionHash( cad: elemento.alias, tipoFH, K);
    int contador=0;
    ////////////////////////////////////////
    //Contador de colisiones, se producen cuando la lista de la posición pos NO está vacía
    ////////////////////////////////////////
    //si la lista con la posición que tenemos no es vacía, se produjo un colision
    if(!esListaVacia( t (*t)[pos])){
        //colision
        contador=1;
    }
    InsertarElementoLista( t &(*t)[pos], p: primeroLista( t (*t)[pos]), e: elemento);

    return contador;
}

```

Como podemos ver en el fragmento de código la función nos aumenta el contador a uno si la lista en esa posición no es vacía, es decir que al haberse producido una colisión el elemento fue añadido a la lista.

De esta forma es como contamos el número de colisiones producidas.

En el caso de encadenamiento como no debemos buscar el espacio libre para colocar una vez se hay producido la colisión (se inserta en el principio de la lista) no tenemos un contador que nos lleve la cuenta de los pasos extra en inserción, como si ocurriría en el caso de recolocación.

En el caso de la **búsqueda** tenemos una variable llamada **nPasosExtraB** que nos permite llevar la cuenta del número de pasos necesarios para encontrar la clave que queremos.

En primer lugar llamamos a la función **busquedaArchivo** que tiene pasado como parámetro por referencia los pasos extras de la búsqueda.

```
void busquedaArchivo(FILE *fp, TablaHash t, unsigned int tipoFH, unsigned int K, int *nPasosExtraB) {
    TIPOELEMENTO paciente;
    if (fp) {
        fscanf(stream: fp, format: "%[^-] - %s - %s", paciente.nombre, paciente.alias, paciente.correo);
        while (!feof(stream: fp)) {
            //Modificar la función BuscarHash() para que devuelva el número de pasos adicionales en la búsqueda
            BuscarHash(t, clavebuscar: paciente.alias, e: &paciente, tipoFH, K, nPasosExtraB);
            fscanf(stream: fp, format: "%[^-] - %s - %s", paciente.nombre, paciente.alias, paciente.correo);
        }
    } else {
        printf(format: "El archivo no ha podido abrirse\n");
    }
}
```

Dentro de esta llamamos a la función *BuscarHash*:

```
int BuscarHash(TablaHash t, char *clavebuscar, TIPOELEMENTO *e, unsigned int tipoFH, unsigned int K, int *nPasosExtraB) {
    TPOSICION p;
    unsigned int encontrado = 0;
    TIPOELEMENTO ele;

    int pos = FuncionHash(cad: clavebuscar, tipoFH, K);

    p = primeroLista(t[pos]);
    while (p != finLista(t[pos]) && !encontrado) {
        recuperarElementoLista(t[pos], p, e: &ele);
        if (strcmp(ele.alias, clavebuscar) == 0) {
            encontrado = 1;
            *e = ele;
        } else {
            p = siguienteLista(t[pos], p);
            //contamos el numero de veces que nos desplazamos para buscar el siguiente de la lista
            *nPasosExtraB+=1;
            //UN PASO ADICIONAL PARA BUSCAR la clave DENTRO DE LA LISTA t[pos]
        }
    }
    return encontrado;
}
```

Como podemos ver en el fragmento de código la función nos aumenta el contador a uno si tiene que desplazarse al siguiente de la lista, es decir, que no encontró el elemento que buscamos. El contador va aumentando y al final de la ejecución como lo tenemos almacenado por referencia podemos mostrarlo por pantalla desde nuestro main.

0.4.2. Método de recolocación

Para la implementación de contadores en el método de recolocación tenemos 3 contadores principales: *nColisiones*, *nPasosExtraI*, *nPasosExtraB*.

Para el cálculo de *nColisiones* modificamos la función *InsertarHash* de forma que esta devuelve 1 o 0 en caso de que ocurriera o no una colisión.

```
void insercionArchivo(FILE *fp, TablaHash t, unsigned int tipoFH, unsigned int K, unsigned int tipoR, unsigned int nPasosExtraI, unsigned int nPasosExtraB, unsigned int nColisiones)
{
    TIPOELEMENTO paciente;
    if (fp) {
        fscanf( stream: fp, format: "%[^-] - %s - %s", paciente.nombre, paciente.alias, paciente.correo);
        while (!feof( stream: fp)) {
            //Modificar la función InsertarHash para que devuelve: 1 si colisión, 0 en caso contrario
            //y acumular estos valores en nColisionesI
            //Añadir a InsertarHash parámetro nPasosExtraI por referencia
            (*nColisiones) += InsertarHash(t, e: paciente, tipoFH, K, tipoR, a, nPasos);
            fscanf( stream: fp, format: "%[^-] - %s - %s", paciente.nombre, paciente.alias, paciente.correo);
        }
    } else {
        printf( format: "El archivo no ha podido abrirse\n");
    }
}
```

Dentro de la propia función *InsertarHash* el número de colisiones es calculado en la función privada *_PosicionInsertar*. Esta función lo que hace es calcular en qué posición hay que insertar el elemento. Para ello realiza un bucle *for* desde *i = 0* hasta *N*.

```
int posicion;
//Inicializar hayColisionI a 0, y pasarla como parámetro
int hayColisionI = 0;
//Enviar a _PosicionInsertar &hayColisionI y nPasosExtraI
posicion = _PosicionInsertar(t, cad: e.alias, tipoFH, K, tipoR, a, haiColision: &hayColisionI, nPasos);

if (t[posicion].alias[0] == VACIO || t[posicion].alias[0] == BORRADO) {
    t[posicion] = e;
}

//Devolver hayColisionI
return hayColisionI;
```

Gracias al funcionamiento de esta función, podemos saber si ha habido una colisión siempre y cuando se realice más de una iteración en el bucle (ya que esto implica que está recolocando el elemento).

```
int _PosicionInsertar(TablaHash t, char *cad, unsigned int tipoFH, unsigned int K, unsigned int N)
// Devuelve el sitio donde podriamos poner el elemento de clave cad
int posicion;
int ini = FuncionHash(cad, tipoFH, K); //calculo la posición mediante la función h

for (int i = 0; i < N; i++) {
    //cuando i no es 0, hay colisión
    if (i > 0) (*haiColision) = 1;
    //Intento recolocar en aux. Cuando i=0, aux=ini, pruebo en la posición dada por
    switch (tipoR) {
        case 1: //recolocación lineal
            posicion = (ini + a * i) % N;
            break;
        case 2: //Recolocación cuadrática
            posicion = (ini + i * i) % N;
            break;
    }
}
```

El cálculo de nPasosExtral se realiza en las mismas funciones que nColisiones y se pasa por referencia como argumento a InserciónHash y _PosicionInsertar.

Aquí observamos las definiciones de las funciones *InsertarHash* y *_PosicionInsertar*

```
int InsertarHash(TablaHash t, TIPOELEMENTO e, unsigned int tipoFH,
                unsigned int K, unsigned int tipoR, unsigned int a,
                unsigned int *nPasos);
```

```
int _PosicionInsertar(TablaHash t, char *cad, unsigned int tipoFH,
                    unsigned int K, unsigned int tipoR, unsigned int a,
                    int *haiColision, unsigned int *nPasos)
```


Dentro de `_PosicionInsertar` se incrementa la variable `nPasosExtraI` con el valor de el número de iteraciones realizadas (`i`) cuando se encuentra un espacio borrado, vacío o se encuentra el propio elemento.

```
//Busco hueco en aux
if (t[posicion].alias[0] == VACIO || t[posicion].alias[0] == BORRADO) {
    //Hueco encontrado, se han necesitado i intentos para ubicar el dato
    //Incrementar en i la variable nPasosExtraI
    (*nPasos) += i;
    return posicion;
}
//Si el elemento a insertar ya estaba en la tabla
if (!strcmp(t[posicion].alias, cad)) {
    //Ya está, se han necesitado i intentos para encontrar el dato
    //Incrementar en i la variable nPasosExtraI
    (*nPasos) += i;
    return posicion;
}
```

El cálculo de `nPasosExtraB` es equivalente al cálculo de `nPasosExtraI` pero sustituyendo las funciones `InsertarHash` y `_PosicionInsertar` por `BuscarHash` y `_PosicionBuscar`. La única diferencia es que cuando se encuentra un valor 'BORRADO' la función de inserción retorna la posición mientras que la función de búsqueda continúa su proceso (un valor que no se ve reflejado en nuestras tablas de análisis de datos porque en ningún momento realizamos la eliminación de ningún paciente).

Aquí podemos observar las definiciones de funciones que toman el argumento `nPasosExtraB`:

```
int EsMiembroHash(TablaHash t, char *cad, unsigned int tipoFH,
                  unsigned int K, unsigned int tipoR,
                  unsigned int a, unsigned int *nPasos)
```

```
int _PosicionBuscar(TablaHash t, char *cad, unsigned int tipoFH,
                   unsigned int K, unsigned int tipoR, unsigned int a,
                   unsigned int *nPasos)
```

A continuación se muestra el fragmento del código de la función *_PosicionBuscar* donde se incrementa la variable *nPasosExtraB* por el número de iteraciones si esa posición está (VACIO) o se corresponde consigo misma (*cad*):

```
if (t[posicion].alias[0] == VACIO) { //si está vacío, terminé de buscar
    //////////////////////////////////////
    //incremento en i el nPasosExtraB
    (*nPasos) += i;
    //////////////////////////////////////
    return posicion;
}

if (!strcmp(t[posicion].alias, cad)) { //si encontré cad, terminé de buscar
    //////////////////////////////////////
    //incremento en i el nPasosExtraB
    (*nPasos) += i;
    //////////////////////////////////////
    return posicion;
}
```

1. Influencia de N para la Inserción.

1.0 Tabla de Datos

A la hora de elegir los tamaños con los que vamos a realizar la experimentación, debemos tener en cuenta que los rangos de factor de carga óptimos son, para recolocación $L \leq 0,5$ y para encadenamiento $L \leq 0,75$.

Para probar esto experimentalmente, decidimos probar con valores de factor de carga 1 y su primo mayor más cercano, 0,5 y su primo mayor más cercano y 0,75 y su primo mayor más cercano. De esta forma obtenemos valores en los que conocemos que deberían dar resultados óptimos y otros en los que no, de esta forma nos permite contrastar la información y saber qué factor de carga es mejor.

TABLA 1. Influencia de N para el proceso de inserción mediante la Función Hash2

Encadenamiento	N = 10000	N = 10007	N = 13334	N = 13337	N = 20000	N = 20011
nColisionesI	9375	3705	4847	2977	9375	2200
Recolocación Simple (a = 1)	N = 10000	N = 10007	N = 13334	N = 13337	N = 20000	N = 20011
nColisionesI	9375	5026	5163	3749	9375	2587
nPasosExtral	672632	414876	17019	14417	79333	5540

NOTA: En el Apartado 1.2. se da una explicación a los valores aislados con N = 20000.

1.1. N Primo vs. No-Primo.

Técnicamente, que **N** pertenezca al conjunto de los *números primos* o no, se podría decir que es irrelevante ya que la condición importante en la creación de una *función hash* óptima y equilibrada es que **N** y **k** sean *primos relativos* o *coprimos*. Ahora bien, el método más sencillo de asegurar esta relación es hacer que **N** sea un valor primo (sin divisores) y **k** menor que **N**. En general, siempre que se está aplicando *aritmética modular*, es más sencillo y da mejores resultados trabajar con valores *primos* por el hecho de no tener divisores distintos del 1 y el mismo.

Los datos de la *Tabla 1* han sido calculados mediante el uso de la *función hash 2* la cuál sigue el algoritmo explicado en el *Apartado 2.0.1*. Para tamaños de tabla similares (pares de columnas 0-1, 2-3, 4-5) se puede observar un gran cambio en los resultados obtenidos, tanto en el número de *colisiones* como en el número de *pasos extra*. Estas variaciones en resultados son más notables con *factores de carga* más cercanos a 1 ya que la función está menos optimizada (e.g. reducción: $1 - (3705 / 9375) \approx 60\%$, $1 - (2977 / 4847) \approx 40\%$).

1.2. Factor de Carga Óptimo vs. No Óptimo.

En el caso de recolocación tenemos un **factor de carga** menor, es decir, que la tabla necesita un tamaño más grande. Esto es debido a que la *recolocación* necesita colocar en una nueva posición el elemento, una vez se produce una *colisión*. Esto conlleva que necesitemos tamaños de tablas más grandes para reducir el número de colisiones. Esto no ocurre en *encadenamiento* ya que no tiene que recolocar el elemento en una nueva posición de la tabla (colocado en el principio de la lista) permitiendo así tener un **factor de carga** mayor y no tener límite del número de elementos que insertamos (listas).

Generalmente cuanto menor sea el **factor de carga** obtendremos menos *colisiones* y menos pasos extra debido al tamaño de la tabla tan elevado. Sin embargo lo que estamos ganando en eficiencia lo estamos perdiendo en memoria por lo que no es del todo óptimo utilizar valores de carga tan pequeños.

No obstante cabe destacar que no siempre el factor de carga menor reduce las colisiones ya que, como podemos observar, tanto para recolocación como para encadenamiento en los factores de carga 0.5 y 0.75 obtenemos valores de colisiones muy similares. Esto es debido a que en Hash 2 nuestra k es 256 y nuestro mod es n , que en nuestro caso es 10000. Como 20000 es el doble de 10000 tienen los mismos factores, los únicos que lo distinguen es un dos a mayores en 20000. Esto produce que al dividir entre el módulo 10000 pues nos da resultados similares en el número de colisiones.

Sin embargo nos dan muchísimos menos pasos extra ya que al tener una tabla de mayor tamaño hay más huecos vacíos y tenemos que desplazarnos menos a la hora de encontrar un nuevo hueco.

Recolocación funciona mejor con 0.5 con número primo (explicación de los módulos anterior), como se aprecia en la tabla .

Encadenamiento funciona mejor con 0.75, como dijimos, cuanto menor sea el factor de carga menor colisiones como norma general; sin embargo, con factor 0.75 primo nos da 2977 y con 0.5 primo nos da 2200. Tenemos una cantidad notoria, menos de colisiones pero tenemos que aumentar en 7000 el tamaño de la memoria por lo que consideramos que es mejor 0.75 en relación colisiones/memoria.

1.3. Conclusión.

Finalmente, podemos concluir que para obtener resultados más óptimos es imprescindible tener un tamaño de *tabla hash* N perteneciente al conjunto de los primos o por lo menos que sea coprimo con k .

En cuanto al tamaño de N , para optimizar el funcionamiento de la *tabla hash* este debería de ser el más grande posible pero nos encontramos con un problema, el espacio en memoria que ocupa. Por esta razón no se debe de establecer un valor de N demasiado elevado si no que es recomendable mantenerlo con valores que produzcan un factor de carga inferior a 0.75 ($N = 13337$), para el método de encadenamiento y de 0.5 ($N = 20011$), para el método de recolocación.

2. Influencia de la Función Hash (1-2-3) y de la clave en la Inserción.

2.0 Explicacion Funcion Hash

2.0.0 Función Hash 1

La función *Hash 1* consiste en sumar todos los valores ascii de los caracteres de la clave que sea elegida y hacer el mod del número de elementos que queramos meter en la tabla.

No funciona bien si N es demasiado grande, porque tendríamos la tabla cargada al principio y muchos espacios vacíos al final lo que supone mayor número de colisiones

En nuestro idioma al usar letras muy parecidas puede ocurrir que muchos de estos valores se repitan y se produzcan un mayor número de colisiones.

2.0.1 Función Hash 2

La función *Hash 2* consiste en la suma ponderada de una cadena de caracteres podemos apreciar su funcionamiento según el siguiente algoritmo:

```
algoritmo h(c: cadena[maxTam]) return 0..maxTam-1;  
  i,suma:entero;  
  principio  
    suma:=0;  
    para i:=maxTam-1 hasta 0 hacer  
      suma:=(suma*256+ord(c[i])) mod maxTam;  
    finpara  
    devuelve(suma);  
  fin
```

Como se puede apreciar en el algoritmo se recorren los caracteres de nuestra cadena y se realiza la multiplicación de la suma que tiene un valor diferente (incrementa considerablemente) por la k (valor que contiene el espectro de todos los valores ascii) más el valor ascii del carácter en módulo n. Una vez recorrido el último carácter devolvemos la suma que será la clave de nuestro elemento.

2.0.2 Función Hash 3

La función *Hash 3* consiste en lo mismo que la función Hash 2 pero con la diferencia de que el valor de k es el que se le quiera asignar.

2.1 Análisis de Resultados

2.0.0. Hash 1.

En el caso de la función hash 1, podemos apreciar que en todas las tablas nos da el mismo resultado aunque estemos cambiando el tamaño de la tabla.

Este método (*Apartado 2.0.0*) es la suma de los valores *ASCII* de la cadena de caracteres que elijamos como clave. El problema que tenemos es que la magnitud de esta suma es muy inferior al tamaño de la tabla que estemos usando (10000 como mínimo en nuestro caso) por lo que vamos a tener las mayorías de las claves al principio de la tabla, a mayores hay que tener en cuenta que las claves que estamos usando son en función de nombres españoles (alias, correos) por lo que las letras que se utilizan van a ser similares (debido a que el español, como muchos otros idiomas, usa un subconjunto de las letras posibles en la gran mayoría de situaciones), y la suma también.

A razón de este comportamiento, las codificaciones hash de las claves se concentran en el principio de la tabla y con valores muy similares, dando lugar a una gran cantidad de colisiones como se puede observar en la *Tabla 2*.

En las siguientes tablas de datos se han resaltado ciertos valores según las siguientes reglas: *Rojo* para resultados extremadamente malos, *Verde* para los resultados óptimos y *Amarillo* para resultados extremadamente buenos en comparación con el resto de métodos para ese valor de *N* (casos especiales). Estos casos especiales serán explícitamente tratados.

2.0.1. Análisis Clave = Nickname

**Tabla 2. Influencia de la función hash en el proceso de inserción en Encadenamiento.
Clave=nickname**

	N = 10000	N = 10007	N = 13334	N = 13337	N = 20000	N = 20011
nColisiones! Hash1	9271	9271	9271	9271	9271	9271
nColisiones! Hash2	9375	3705	4847	2977	9375	2200
nColisiones! Hash3 K = 500	9982	3645	4753	2984	9976	2125
nColisiones! Hash3 K = 313	3755	3646	2951	2944	2253	2101

En esta tabla podemos observar lo comentado en la introducción de este apartado en relación a la *función hash 1*, independientemente de aumentar o no *N*, el número de colisiones no disminuye. El resto de los datos de la tabla sigue el comportamiento definido en el *Apartado 1* y el *Apartado 2.0.0*.

Es notable destacar el buen funcionamiento de la función hash 3 con $k = 313$ y $N = 10000$ o 20000 . Estos resultados se explican debido a que 313 es un número primo por lo

que k y N sean coprimos. Esta relación de *coprimeidad* no se da lugar con los otros valores de k (256 y 500) produciendo peores resultados en el número de colisiones, excepcionalmente $k = 500$, ya que 500 es divisor de 10000 y 20000.

Tabla 3. Influencia de la función hash en el proceso de inserción en Recolocación Lineal ($a = 1$). Clave=nickname

	N = 10000	N = 10007	N = 13334	N = 13337	N = 20000	N = 20011
nColisionesl Hash1	9784	9784	9784	9784	9784	9784
nPasosExtral Hash1	46720557	46720557	46720557	46720557	46720557	46720557
nColisionesl Hash2	9375	5026	5163	3749	9375	2587
nPasosExtral Hash2	672632	414876	17019	14417	79333	5546
nColisionesl Hash3 K = 500	9982	4942	5077	3810	9976	2506
nPasosExtral Hash3 K = 500	16904000	294282	16225	15300	5481017	4991
nColisionesl Hash3 K = 313	5054	4962	3719	3685	2659	2474
nPasosExtral Hash3 K = 313	508373	942599	14913	13313	5373	4813

La *Tabla 3* sigue el mismo comportamiento general que los datos de la *Tabla 2*, esto es, pésimo rendimiento de la *función hash 1*, óptimo número de colisiones cuando N y k son coprimos (datos verdes y amarillos). Cabe destacar que, aplicando el método de recolocación, también hay que tener en cuenta el número de pasos de búsqueda para la inserción. Este número, aunque también influido de forma substancial por la relación de coprimos, depende del tamaño de N , más concretamente, del factor de carga como bien se explicó en el *Apartado 1.2*. Por esta razón podemos observar un número de colisiones válido en $N = 10007$ y 10000 (valor amarillo) pero el número de pasos de búsqueda es todavía demasiado elevado debido al factor de carga proximo a 1.

2.0.2. Análisis Clave = Correo

En este apartado, como podemos comprobar en las tablas 4 y 5, la principal diferencia con los resultados del apartado anterior es la magnitud del número de colisiones. Al emplear el correo como clave para la función hash, debido a la naturaleza del correo, este consta de más caracteres (por normal general) que un alias. Esto implica que la clave calculada

mediante la función hash sea más única para cada correo (a excepción de la *función hash 1*, cuyo funcionamiento ya explicamos previamente) originando menos colisiones.

**Tabla 4. Influencia de la función hash en el proceso de inserción en Encadenamiento.
Clave=correo**

	N = 10000	N = 10007	N = 13334	N = 13337	N = 20000	N = 20011
nColisionesl Hash1	8787	8787	8787	8787	8787	8787
nColisionsl Hash2	4569	3683	3007	3025	3838	2132
nColisionesl Hash3 K = 500	9951	3692	2973	2994	9939	2082
nColisionesl Hash3 K = 313	3663	3606	2992	2946	2109	2173

Tabla 5. Influencia de la función hash en el proceso de inserción en Recolocación Lineal (a = 1). Clave=correo

	N = 10000	N = 10007	N = 13334	N = 13337	N = 20000	N = 20011
nColisionesl Hash1	9485	9485	9485	9485	9485	9485
nPasosExtral Hash1	43756551	43756551	43756551	43756551	43756551	43756551
nColisionsl Hash2	5913	5020	3830	3801	4607	2509
nPasosExtral Hash2	837630	543727	15206	14804	18840	4977
nColisionesl Hash3 K = 500	9956	5006	3809	3762	9945	2427
nPasosExtral Hash3 K = 500	20765418	493389	14578	14762	7571573	5046
nColisionesl Hash3 K = 313	4994	4958	3818	3738	2488	2541
nPasosExtral Hash3 K = 313	649625	730640	14834	14983	4925	5204

3. Influencia de estrategia de recolocación en Inserción.

3.0. Explicación de Tamaño Óptimo.

Como hemos observado en el apartado anterior, los mejores resultados han sido obtenidos cuando se cumplían las condiciones óptimas especificadas en el *Apartado 1*. De esta forma, no sorprende que el tamaño que ofrezca mejores resultados sea $N = 20011$. De la misma manera, la función hash 1 fué completamente descartada debido a su naturaleza.

Finalmente, optamos por escoger $N = 20011$ debido a su excelente funcionamiento con recolocación y encadenamiento al igual que su casi necesidad trabajando con *clave = correo* debido al mayor número de caracteres de este.

En cuanto a las funciones, nos decidimos por escoger la función hash 2 con $k = 256$ (el número de caracteres ASCII) y la función hash 3 con $k = 500$. Escogimos $k = 500$ y no $k = 313$ porque la relación de coprimidad entre k y N ya viene dada por N ser primo y obtuvimos mejores valores con valores de k mayores.

3.1. Recolocación Lineal vs. Cuadrática

Viendo los resultados anteriores, podemos apreciar que en normas generales la cuadrática mejora a la lineal (con $a=1$) cuando tenemos un tamaño más grande de tabla. Esto se debe a que con la cuadrática damos saltos más grandes a la hora de colocar elemento y puede ser que encontremos en menos pasos el hueco que necesitamos. Sin embargo en la lineal con $a=1$ puede ser que si la tabla es muy grande y las colisiones se encuentren al principio, le cueste más desplazarse hasta los huecos vacíos finales de la tabla a la hora de recolocar.

No obstante, si vemos los resultados posteriores, en la *tabla 6*, si utilizamos un valor ' a ' lo suficientemente grande, podemos incluso obtener mejores resultados que la cuadrática en cuanto a colisiones. No obstante, a la hora de pasos extra la cuadrática funciona bastante mejor que la lineal.

En el método de recolocación, puede darse la posibilidad de que no se encuentre sitio para colocar la clave, según la experimentación realizada, cuanto mayor sea el paso entre un elemento y otro mayor es la posibilidad de que no se encuentre sitio. Esto ocurre en un mayor cantidad con factores de carga mayores ya que tenemos un tamaño de tabla pequeño.

3.2. Análisis de Datos

Tabla 6. Influencia de la estrategia de recolocación en el proceso de inserción

Casos	Variables	Simple (lineal con $a = 1$)	Lineal ($a = 800$)	Lineal ($a = 809$)	Cuadrática
Función Hash 2 N = 20011	nColisionesl	2587	2580	2574	2587
	nPasosExtral	5540	5078	4982	4582
Función Hash 3 (k=500) N = 20011	nColisionesl	2506	2495	2474	2518
	nPasosExtral	4991	5053	5005	4341

En estos casos que hemos elegido, al ser los mejores casos entre funciones hash y tamaños podemos ver poca diferencia entre colisiones y pasos extra en general.

Podemos destacar que hay una mejora notoria en cuanto a pasos entre la recolocación lineal y la cuadrática ya que la segunda es más óptima.

En este caso no vemos una diferencia notoria entre elegir un valor de a que sea prima o no, por lo que suponemos que da un número menor de pasos extra y de colisiones debido a que si el valor de a es mayor tenemos más diferencia entre elemento y elemento por lo que tenemos menos colisiones y pasos.

4. Estimación de la Eficiencia en el acceso a datos (búsqueda).

4.0. Análisis de Datos

Debido a la similitud entre los métodos de inserción y búsqueda, no se encuentran muchas diferencias entre esta tabla y las analizadas en el *Apartado 3* en cuanto a los métodos de recolocación simple y encadenamiento. Seguimos observando los valores óptimos en los factores de carga menores y con N y k coprimos, las excepciones con $k = 500$ y $N = 10000$ y 20000 . Este comportamiento general también se extiende al método de recolocación cuadrática.

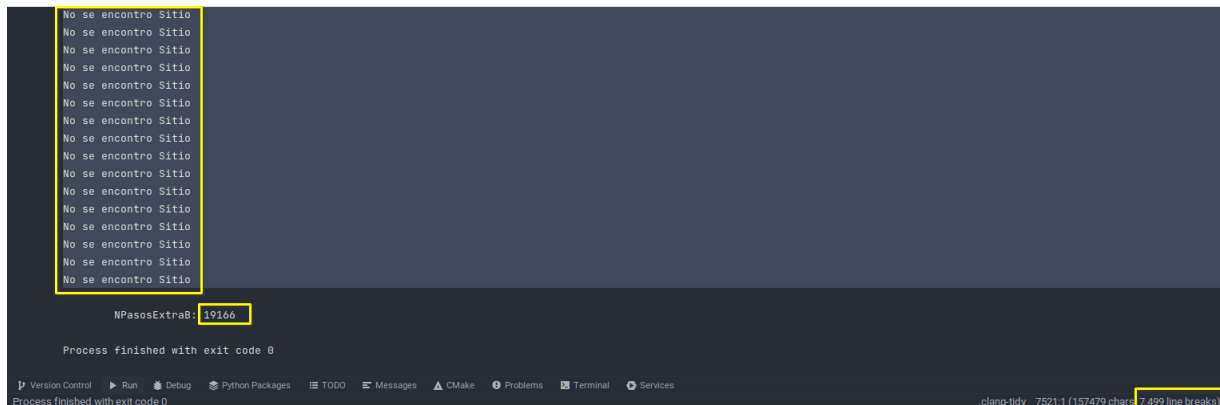
Sin embargo, cabe destacar el contraste en pasos extra de búsqueda entre el método de encadenamiento y los de recolocación (simple y cuadrática) especialmente con factores de carga cercanos a 1. Esto se debe al buen funcionamiento de la función hash que reparte de forma equitativa las claves, por lo que las listas de elementos en el método de encadenamiento tienen una longitud media por lo que se recorren rápidamente.

Tabla 7. Influencia de N en el proceso de búsqueda

	N = 10000	N = 10007	N = 13334	N = 13337	N = 20000	N = 20011
Encadenamiento Hash1	129176	129176	129176	129176	129176	129176
Encadenamiento Hash2	79333	5037	7597	3820	79333	2605
Encadenamiento Hash3 K = 500	4397778	4905	7179	3763	3594373	2465
Encadenamiento Hash3 K = 313	5202	4892	3705	3671	2684	2463
Recolocacion Simple Hash1	46720557	46720557	46720557	46720557	46720557	46720557
Recolocacion Simple Hash2	672632	414876	17019	14417	79333	5540
Recolocacion Simple Hash3 K = 500	16904000	294282	16625	9686	5981017	4991
Recolocacion Simple Hash3 K = 313	608373	942599	14913	9535	5375	4813
Recolocación Cuadrática Hash1	681312	681312	681312	681312	681312	681312
Recolocación Cuadrática Hash2	19166*	66106	12513	9507	46630*	4582
Recolocación Cuadrática Hash 3 K = 500	342852	63666	12115	9686	1243202*	4341
Recolocación Cuadrática Hash 3 K = 313	63632	69022	9735	9535	4435	4183

A mayores, cabe destacar los valores azules marcados con un asterisco, ya que estos no son los valores reales de pasos realizados. Esto se debe a que de la manera que está implementado el algoritmo, la variable *nPasosExtraB* solo se actualiza cuando encuentra el valor que busca por lo que si no se encuentra el valor, no se incrementa el contador de pasos. Este fenómeno es exactamente lo que ocurre en los valores azules y para realizar la

comprobación, añadimos un *printf* cada vez que no se encuentre el valor buscado y posteriormente contamos todos estos *printf*'s seleccionandolos en terminal.



```
No se encontro Sitio
No se encontro Sitio
No se encontro Sitio
No se encontro Sitio
No se encontro Sitio
No se encontro Sitio
No se encontro Sitio
No se encontro Sitio
No se encontro Sitio
No se encontro Sitio
No se encontro Sitio
No se encontro Sitio
No se encontro Sitio
No se encontro Sitio
No se encontro Sitio
No se encontro Sitio
No se encontro Sitio
No se encontro Sitio
NPasosExtraB: 19166
Process finished with exit code 0
```

Este suceso solo se da lugar cuando se aplica la recolocación cuadrática con valores no óptimos para N y k.

4.1. Conclusión

Después de nuestra experimentación sobre las tablas Hash podemos obtenemos las siguientes conclusiones:

En primer lugar el método más óptimo en cuanto a tiempo es el método de encadenamiento, como se puede observar en la tabla 7, ya que obtiene los valores de pasos extras más bajos.

No obstante cabe destacar que este método ocupa más memoria que el método de recolocación, ya que además de almacenar la tabla, en cada posición tenemos una lista en la que se guardan las colisiones.

Estos datos tan óptimos son para encadenamiento con factor de carga 0.5, en cambio podemos reducir la memoria utilizada empleando un factor de carga de 0.75. Reducimos significativamente la memoria aunque perdamos eficiencia en ejecución.

Por lo que concluimos con que la mejor relación entre tamaño y memoria para la búsqueda es encadenamiento con factor de carga 0.75 y con tamaño de la tabla 13337.

Perdemos en eficiencia para ganamos en memoria significativamente.